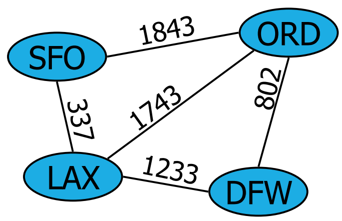
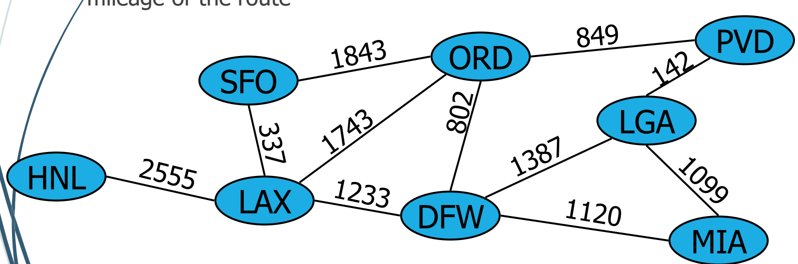
Graphs



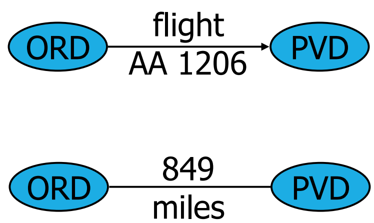
Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



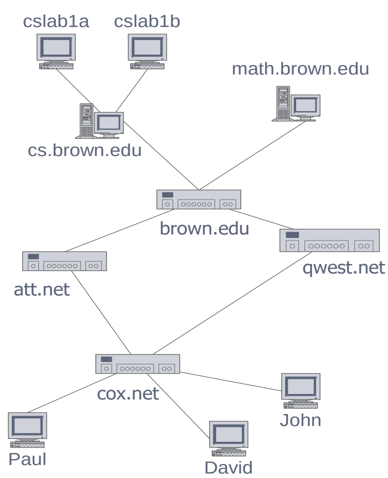
Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network



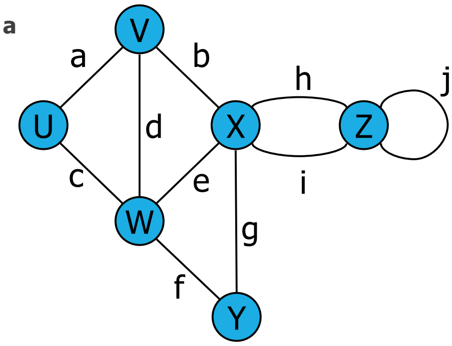
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



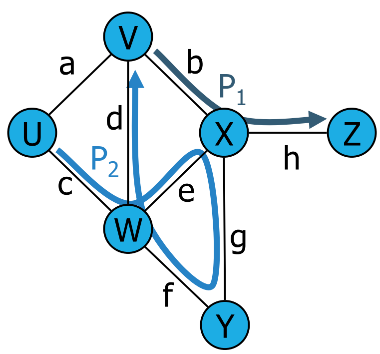
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the **endpoints** of **a**
- Edges incident on a vertex
  - a, d, and b are **incident** on V
- Adjacent vertices
  - U and V are **adjacent**
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



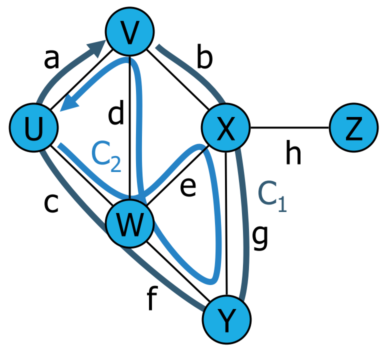
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



Properties of Graphs

Notation

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

Property 1

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

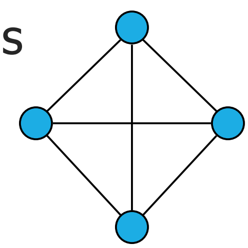
Proof: each edge is counted twice

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 $(n-1)$



Example

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

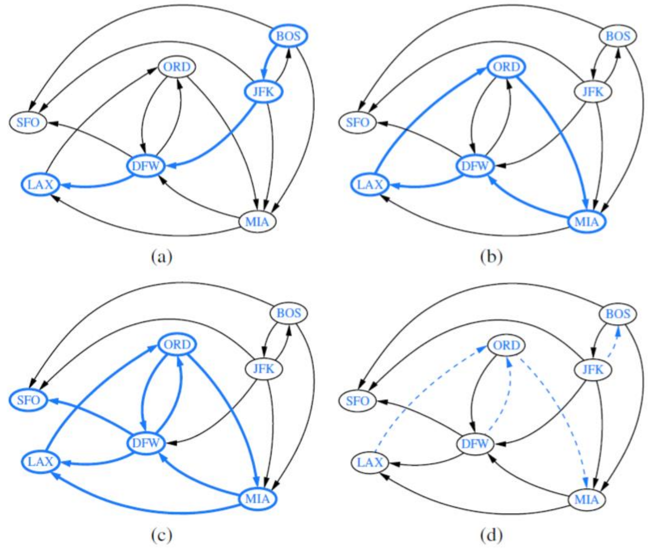


Figure 14.3: Examples of reachability in a directed graph: (a) a directed path from BOS to LAX is highlighted; (b) a directed cycle (ORD, MIA, DFW, LAX, ORD) is highlighted; its vertices induce a strongly connected subgraph; (c) the subgraph of the vertices and edges reachable from ORD is highlighted; (d) the removal of the dashed edges results in a directed acyclic graph.

Vertices and Edges

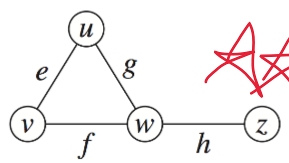
- A **graph** is a collection of **vertices** and **edges**.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.

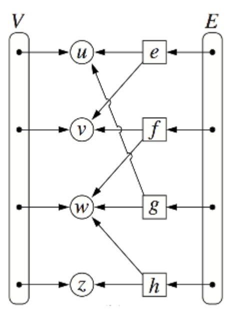
Graph ADT

- numVertices( ): Returns the number of vertices of the graph.
- vertices( ): Returns an iteration of all the vertices of the graph.
- numEdges( ): Returns the number of edges of the graph.
- edges( ): Returns an iteration of all the edges of the graph.
- getEdge(u, v): Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
- endVertices(e): Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
- opposite(v, e): For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
- outDegree(v): Returns the number of outgoing edges from vertex v.
- inDegree(v): Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
- outgoingEdges(v): Returns an iteration of all outgoing edges from vertex v.
- incomingEdges(v): Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
- insertVertex(x): Creates and returns a new Vertex storing element x.
- insertEdge(u, v, x): Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
- removeVertex(v): Removes vertex v and all its incident edges from the graph.
- removeEdge(e): Removes edge e from the graph.

Edge List Structure

- Vertex object (stored in an unordered list V)
  - element
  - reference to position in vertex sequence
- Edge object (stored in an unordered list E)
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects

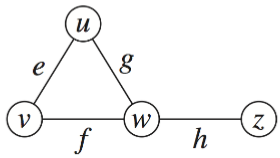


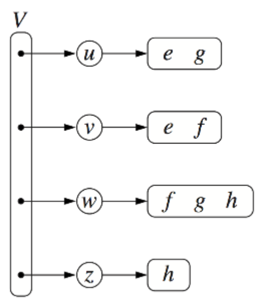


Schematic representation of the edge list structure for G. Notice that an edge object refers to the two vertex objects that correspond to its endpoints, but that vertices do not refer to incident edges.

Adjacency List Structure

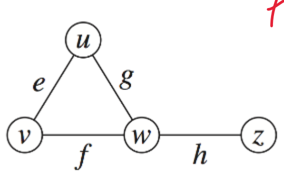
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices





Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge



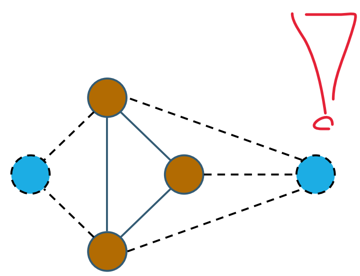
		0	1	2	3
$u \longrightarrow$	0		e	g	
$v \longrightarrow$	1	e		f	
$w \longrightarrow$	2	g	f		h
$z \longrightarrow$	3			h	

Performance

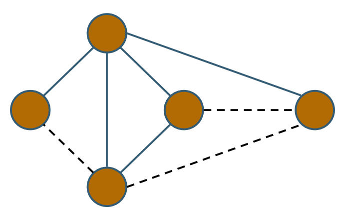
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	$\deg(v)$	n
areAdjacent (v, w)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	$\deg(v)$	n^2
removeEdge(e)	1	1	1

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 (not necessary the edges)



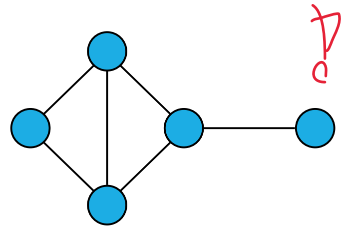
Subgraph



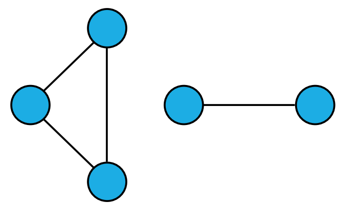
Spanning subgraph

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

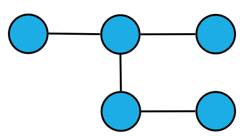


Non connected graph with two connected components

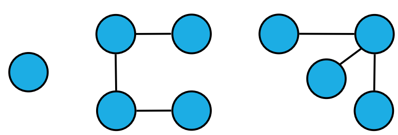
Trees and Forests

- 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



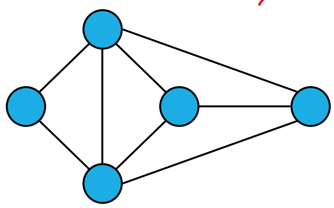
Tree



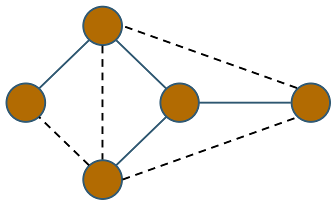
Forest

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

Graph Traversals

Graph Traversal

- A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.
- Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of reachability, that is, in determining how to travel from one vertex to another while following paths of a graph.

Interesting Problems of Reachability in an Undirected Graph G

- Computing a path from vertex u to vertex v, or reporting that no such path exists.
- Given a start vertex s of G, computing, for every vertex v of G, a path with the minimum number of edges between s and v, or reporting that no such path exists.
- Testing whether G is connected.
- Computing a spanning tree of G, if G is connected.
- Computing the connected components of G.
- Identifying a cycle in G, or reporting that G has no cycles.

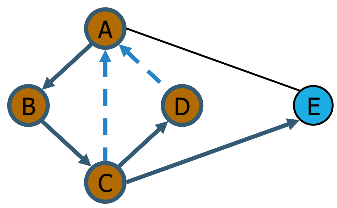
Interesting Problems of Reachability in an Directed Graph G

- Computing a directed path from vertex u to vertex v, or reporting that no such
- path exists.
- Finding all the vertices of G that are reachable from a given vertex s.
- Determine whether G is acyclic.
- Determine whether G is strongly connected.

Graph Traversal Algorithms

- There are two efficient graph traversal algorithms, called
  - Depth-first Search
  - Breadth-first Search

Depth-First Search



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

Depth-First Search

- Depth-first search in a graph G is analogous to wandering in a labyrinth with a string and a can of paint without getting lost. We begin at a specific starting vertex s in G, which we initialize by fixing one end of our string to s and painting s as "visited." The vertex s is now our "current" vertex. In general, if we call our current vertex u, we traverse G by considering **an arbitrary edge (u,v) incident to the current vertex u**. If the edge (u,v) leads us to a vertex v that is already visited (that is, painted), we ignore that edge. If, on the other hand, (u,v) leads to an unvisited vertex v, then we unroll our string, and go to v. We then paint v as "visited," and make it the current vertex, repeating the computation above.

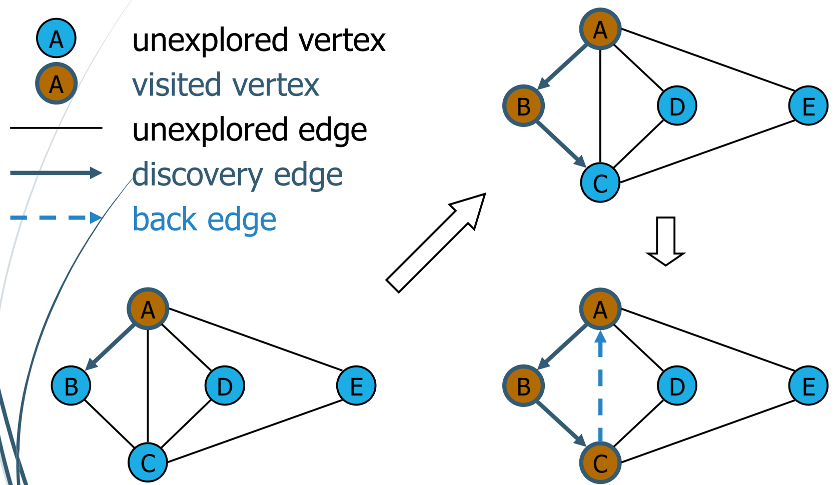
DFS Algorithm from a Vertex

```
Algorithm DFS(G, u):
  Input: A graph G and a vertex u of G
  Output: A collection of vertices reachable from u, with their discovery edges
   Mark vertex u as visited.
   for each of u's outgoing edges, e = (u,v) do
     if vertex v has not been visited then
       Record edge e as the discovery edge for vertex v.
       Recursively call DFS(G, v).
```

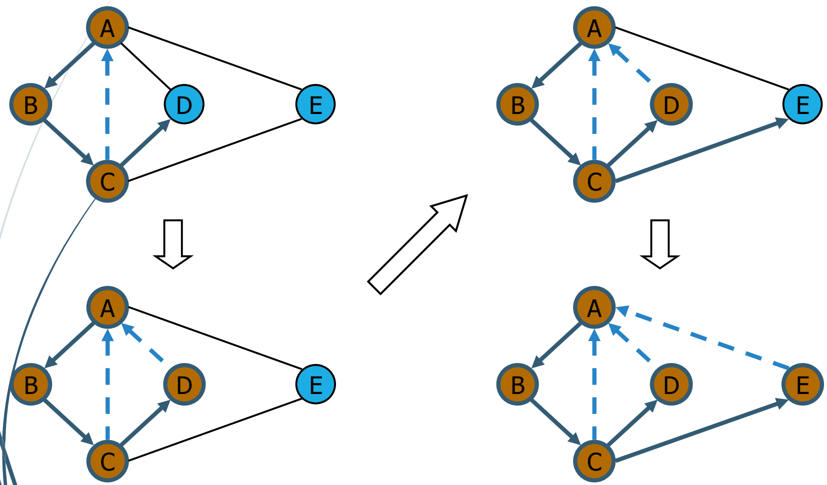
Java Implementation

```
/** Performs depth-first search of Graph g starting at Vertex u. */
public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
                    Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  known.add(u);                              // u has been discovered
  for (Edge<E> e : g.outgoingEdges(u)) {     // for every outgoing edge from u
    Vertex<V> v = g.opposite(u, e);
    if (!known.contains(v)) {
      forest.put(v, e);                      // e is the tree edge that discovered v
      DFS(g, v, known, forest);              // recursively explore from v
    }
  }
}
```

Example



Example (cont.)



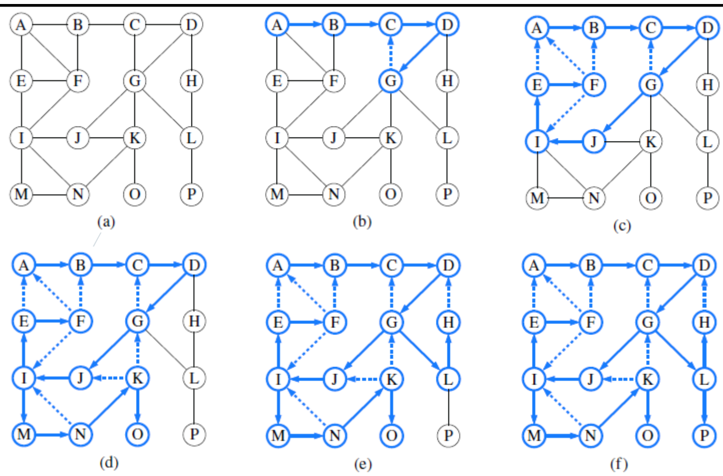
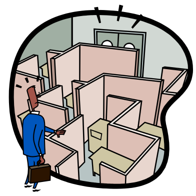
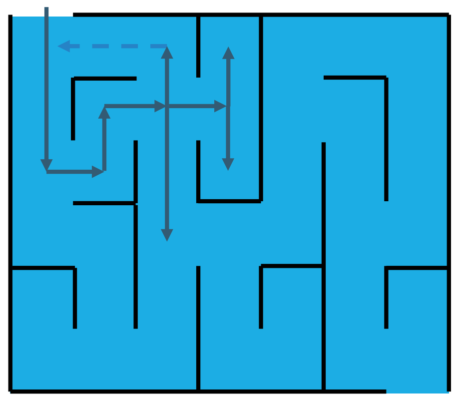


Figure 14.9: Example of depth-first search traversal on an undirected graph starting at vertex A. We assume that a vertex's adjacencies are considered in alphabetical order. Visited vertices and explored edges are highlighted, with discovery edges drawn as solid lines and nontree (back) edges as dashed lines: (a) input graph; (b) path of tree edges, traced from A until back edge (G,C) is examined; (c) reaching F, which is a dead end; (d) after backtracking to I, resuming with edge (I,M), and hitting another dead end at O; (e) after backtracking to G, continuing with edge (G,L), and hitting another dead end at H; (f) final result.

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)



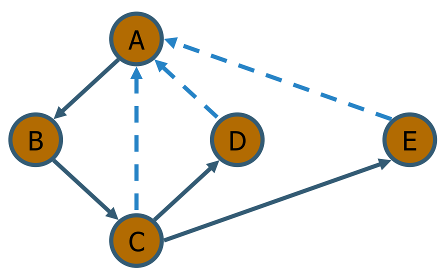
Properties of DFS

Property 1

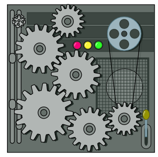
DFS*(*G*, *v*)** visits all the vertices and edges in the connected component of ***v

Property 2

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



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

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $DFS(G, u)$ with u 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(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

Path Finding in Java

```
/** Returns an ordered list of edges comprising the directed path from u to v. */
public static <V,E> PositionalList<Edge<E>>
constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
              Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Edge<E>> path = new LinkedPositionalList<>();
  if (forest.get(v) != null) {                  // v was discovered during the search
    Vertex<V> walk = v;                         // we construct the path from back to front
    while (walk != u) {
      Edge<E> edge = forest.get(walk);
      path.addFirst(edge);                      // add edge to *front* of path
      walk = g.opposite(walk, edge);            // repeat with opposite endpoint
    }
  }
  return path;
}
```


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, z)
  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)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```

DFS for an Entire Graph

- 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)
```

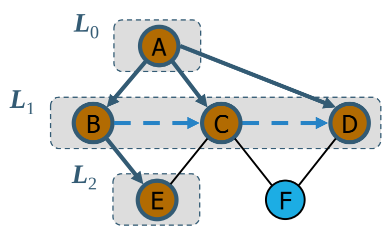
```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
     in the connected component of v
     as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w)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

All Connected Components

- Loop over all vertices, doing a DFS from each unvisted one.

```
/** Performs DFS for the entire graph and returns the DFS forest as a map. */
public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
  Set<Vertex<V>> known = new HashSet<>();
  Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
  for (Vertex<V> u : g.vertices())
    if (!known.contains(u))
      DFS(g, u, known, forest);            // (re)start the DFS process at u
  return forest;
}
```

Breadth-First Search



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

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)
  L_0 ← new empty sequence
  L_0.addLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬L_i.isEmpty()
    L_{i+1} ← new empty sequence
    for all v ∈ L_i.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)
            L_{i+1}.addLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

Breadth-First Search

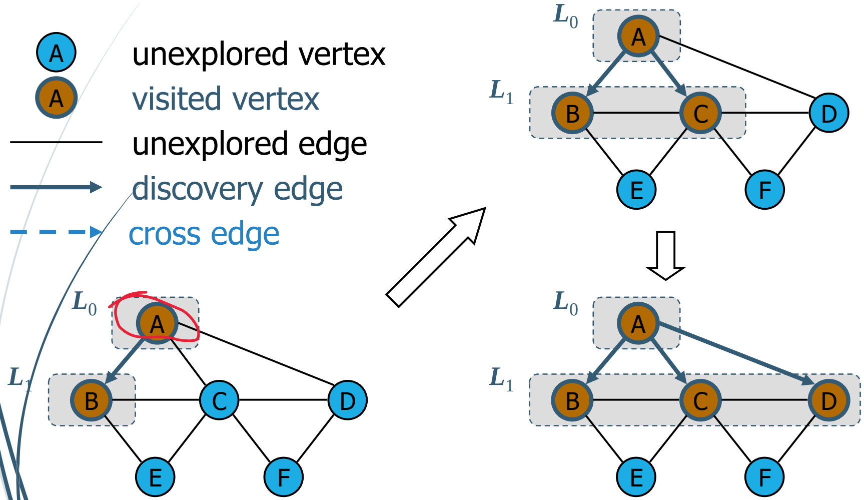
44

Java Implementation

```
/** Performs breadth-first search of Graph g starting at Vertex u. */
public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
                  Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
  known.add(s);
  level.addLast(s);                    // first level includes only s
  while (!level.isEmpty()) {
    PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
    for (Vertex<V> u : level)
      for (Edge<E> e : g.outgoingEdges(u)) {
        Vertex<V> v = g.opposite(u, e);
        if (!known.contains(v)) {
          known.add(v);
          forest.put(v, e);            // e is the tree edge that discovered v
          nextLevel.addLast(v);        // v will be further considered in next pass
        }
      }
    level = nextLevel;                 // relabel 'next' level to become the current
  }
}
```

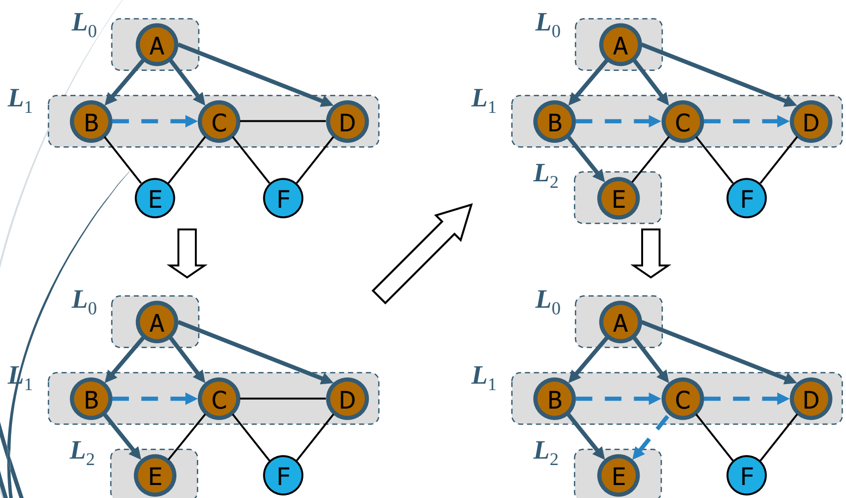
45

Example



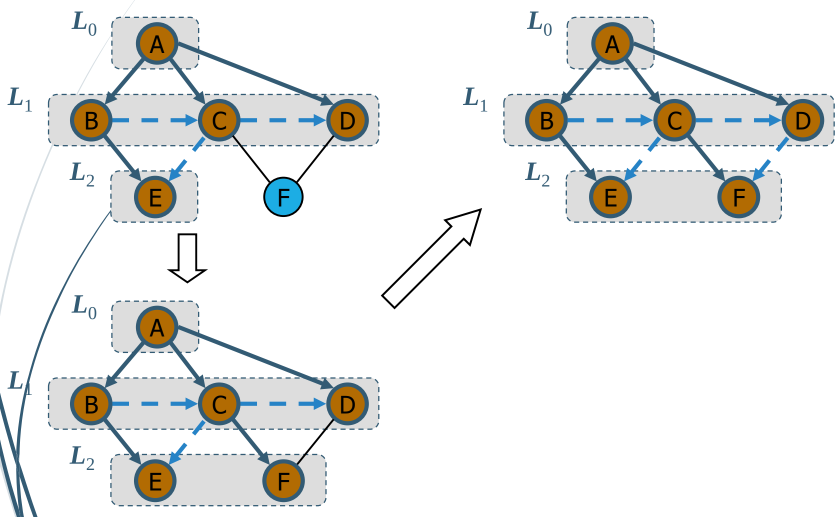
46

Example (cont.)



47

Example (cont.)



48

Properties

Notation

G_s: connected component of s

Property 1

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

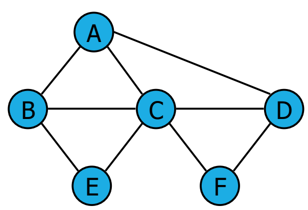
Property 2

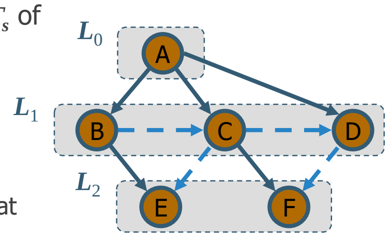
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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

50

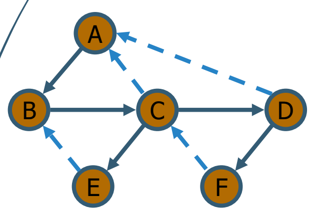
Applications

- 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

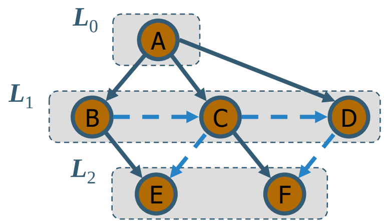
51

DFS vs. BFS

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



DFS



BFS

52

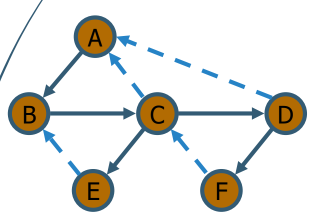
DFS vs. BFS (cont.)

Back edge (v,w)

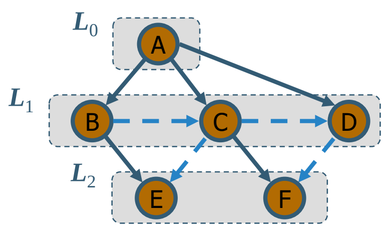
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

- w is in the same level as v or in the next level



DFS



BFS

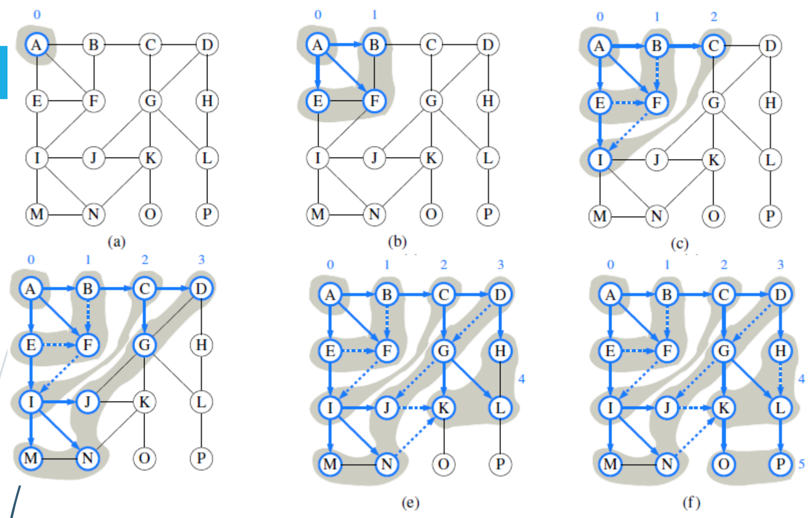
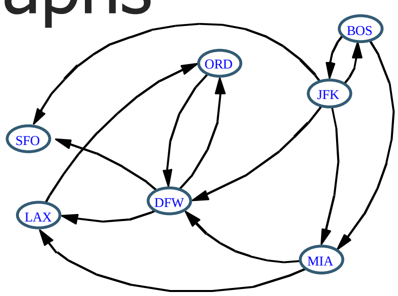


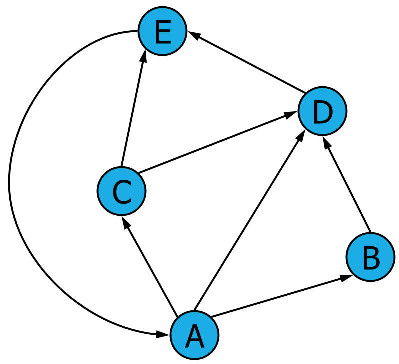
Figure 14.10: Example of breadth-first search traversal, where the edges incident to a vertex are considered in alphabetical order of the adjacent vertices. The discovery edges are shown with solid lines and the nontree (cross) edges are shown with dashed lines: (a) starting the search at A; (b) discovery of level 1; (c) discovery of level 2; (d) discovery of level 3; (e) discovery of level 4; (f) discovery of level 5.

Directed Graphs

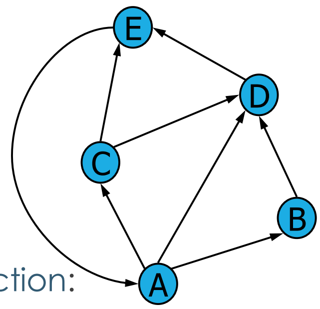


Digraphs

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



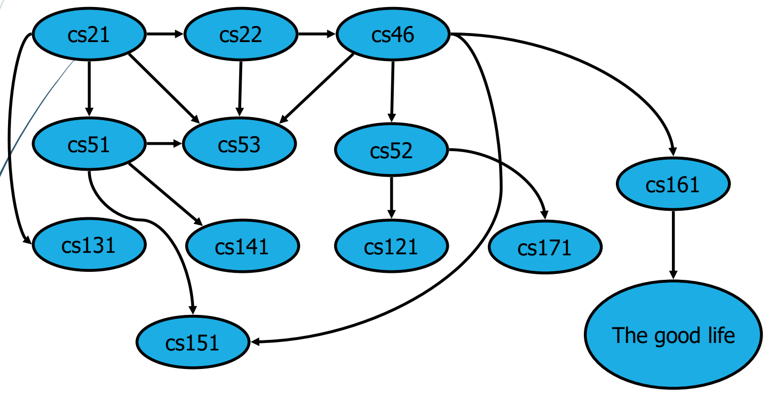
Digraph Properties



- 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, $m \leq n \cdot (n - 1)$
- 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

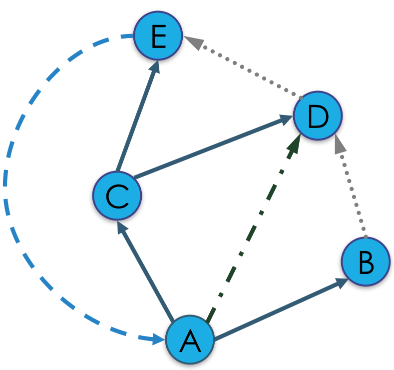
Digraph Application

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



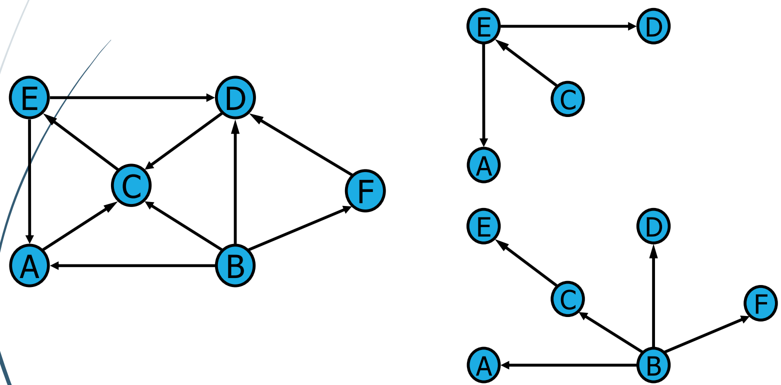
Directed DFS

- 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

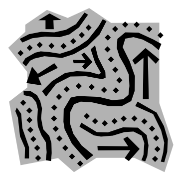


Reachability

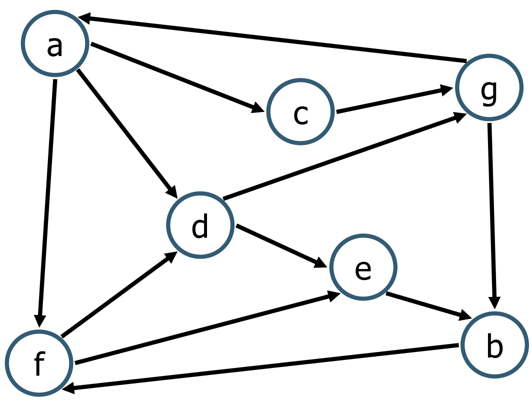
- DFS tree rooted at v: vertices reachable from v via directed paths



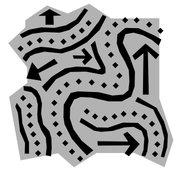
Strong Connectivity



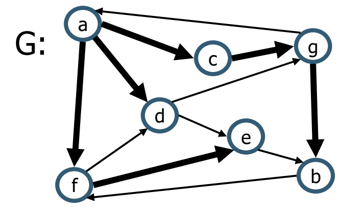
- Each vertex can reach all other vertices

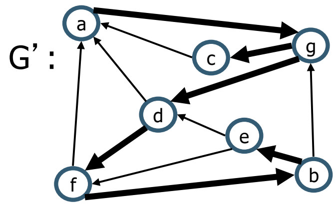


Strong Connectivity Algorithm

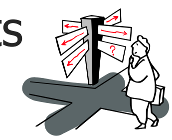


- Pick a vertex v in G
- Perform a DFS from v in G
  - If there's a w not visited, print "no"
- Let G′ be G with edges reversed
- Perform a DFS from v in G′
  - If there's a w not visited, print "no"
  - Else, print "yes"
- Running time: O(n+m)

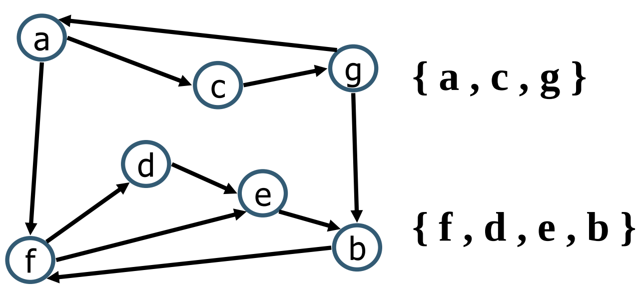




Strongly Connected Components

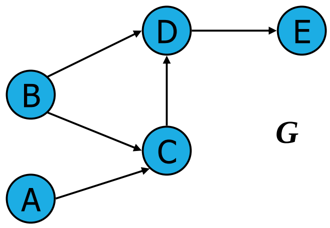


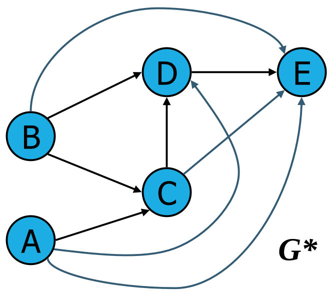
- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in O(n+m) time using DFS, but is more complicated (similar to biconnectivity).



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

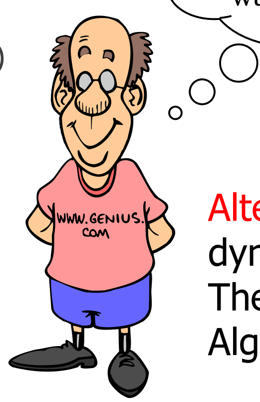




Computing the Transitive Closure

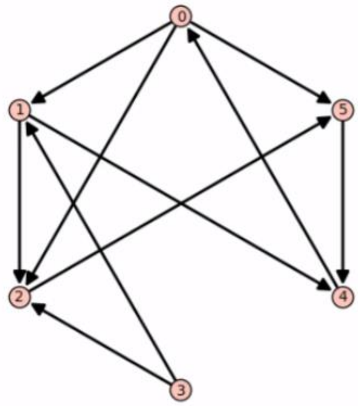
- We can perform DFS starting at each vertex
  - O(n(n+m))

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

Adjacency Matrix (Rem.)



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

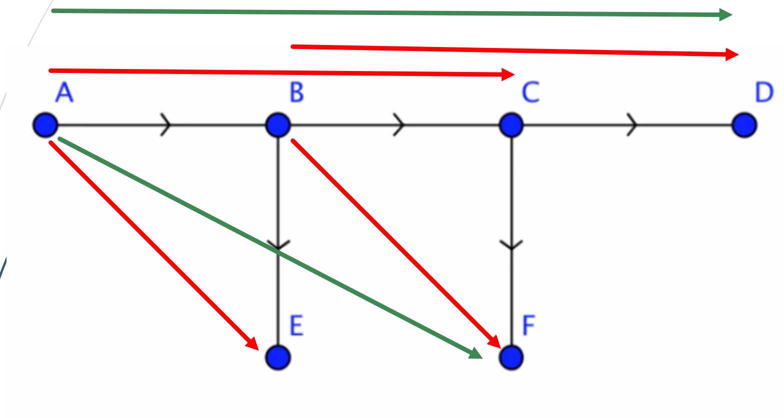
M[i][j] or M[i,j] =
Entry in row i, column j

Reminder: And / Or

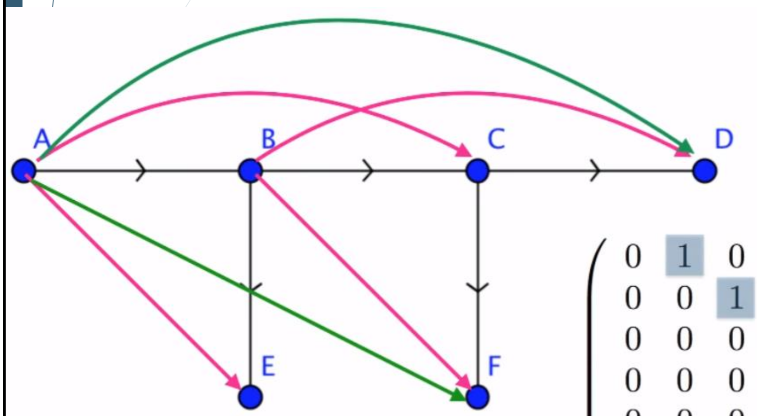
b1	b2	b1 AND b2
1	1	**1**
1	0	**0**
0	1	**0**
0	0	**0**

b1	b2	b1 OR b2
1	1	**1**
1	0	**1**
0	1	**1**
0	0	**0**

Warshall Algorithm - Concept

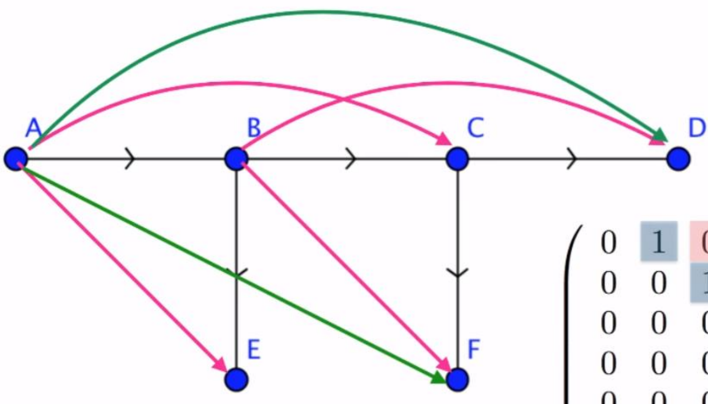


First Level Connections



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Second Level Connections

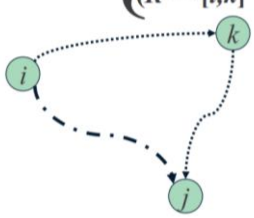


$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Warshall's Algorithm (Math)

- On the kth iteration the algorithm determines if a path exists between two vertices i,j using just vertices among 1,…..,k allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k\text{-}1) \\ \text{or} & \\ (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just } 1,\ldots,k\text{-}1) \end{cases}$$



k^{th} iteration

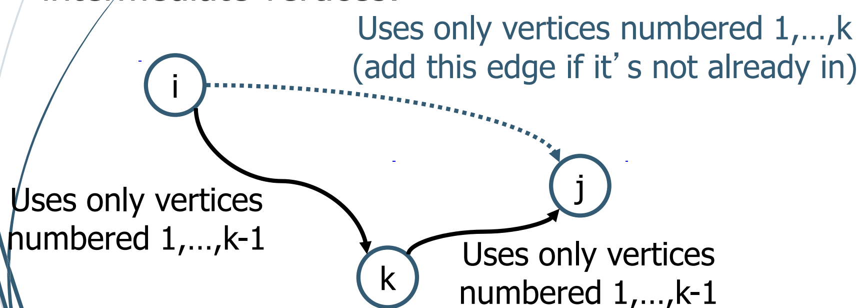
Warshall's Algorithm (Adjacency Matrix)

```
def warshall(M):
    n = M.nrows()
    W = M
    for k in range(n):
        for i in range(n):
            for j in range(n):
                W[i,j] = W[i,j] or (W[i,k] and W[k,j])
    return W
```

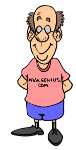
- Time Efficiency **$O(n^3)$**
- Space Efficiency: **Matrices can be written over their predecessors.**

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 (Generic – Adjacency List)



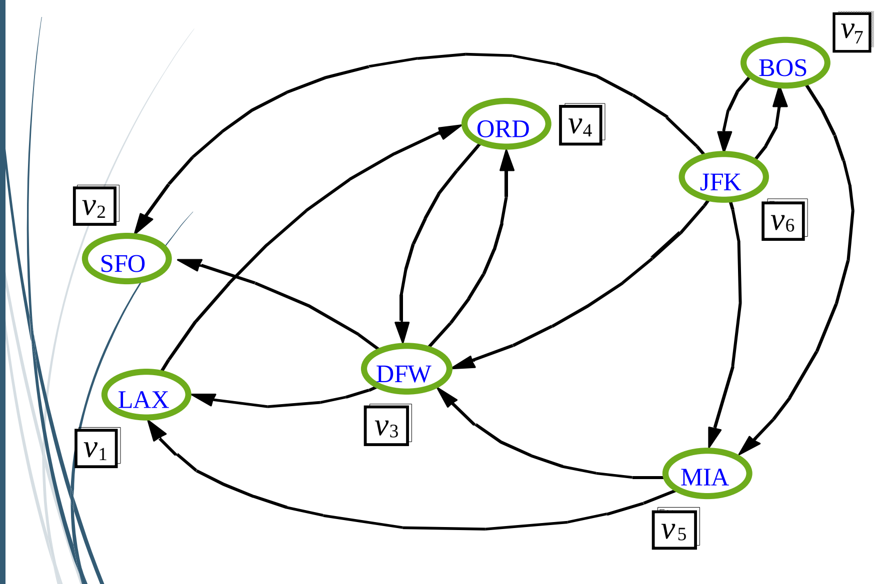
- Number vertices $v_1, \ldots, v_n$
- Compute digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is computed from G_{k-1}
- Running 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 ← 1
  for all v ∈ 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 ≠ k) do
       for j ← 1 to n (j ≠ i, k) do
         if G_{k-1}.areAdjacent(v_i, v_k) ∧
              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
```

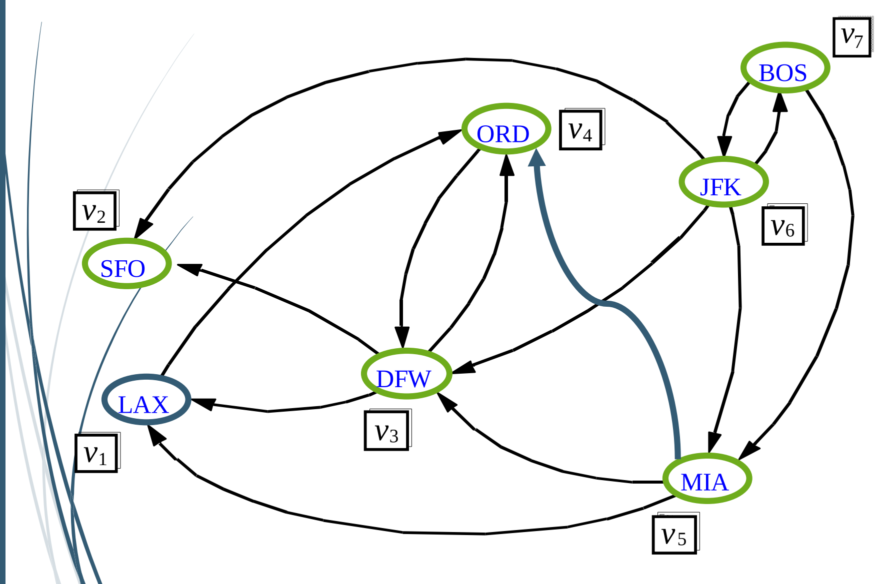
Java Implementation

```java
/** Converts graph g into its transitive closure. */
public static <V,E> void transitiveClosure(Graph<V,E> g) {
  for (Vertex<V> k : g.vertices())
    for (Vertex<V> i : g.vertices())
      // verify that edge (i,k) exists in the partial closure
      if (i != k && g.getEdge(i,k) != null)
        for (Vertex<V> j : g.vertices())
          // verify that edge (k,j) exists in the partial closure
          if (i != j && j != k && g.getEdge(k,j) != null)
            // if (i,j) not yet included, add it to the closure
            if (g.getEdge(i,j) == null)
              g.insertEdge(i, j, null);
}
```

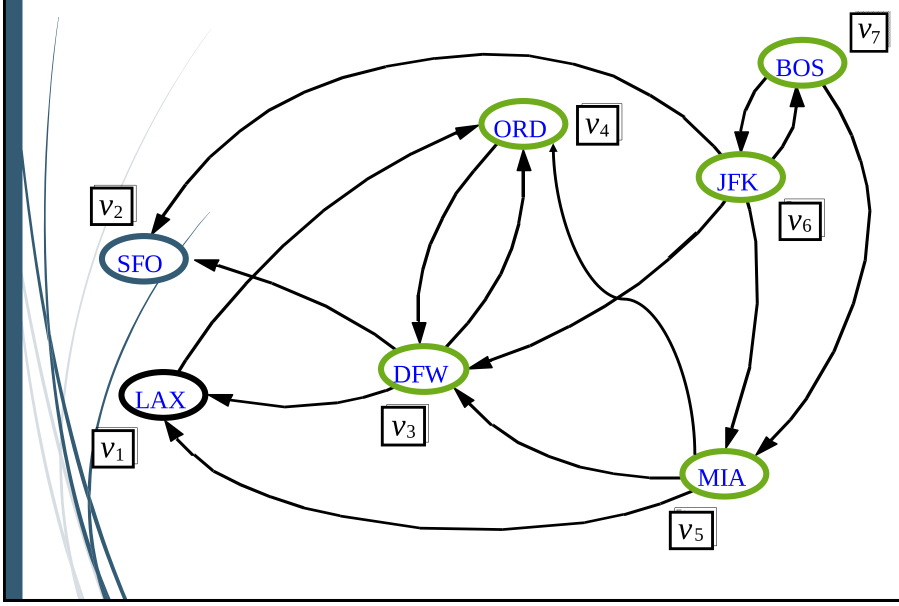
Floyd-Warshall Example



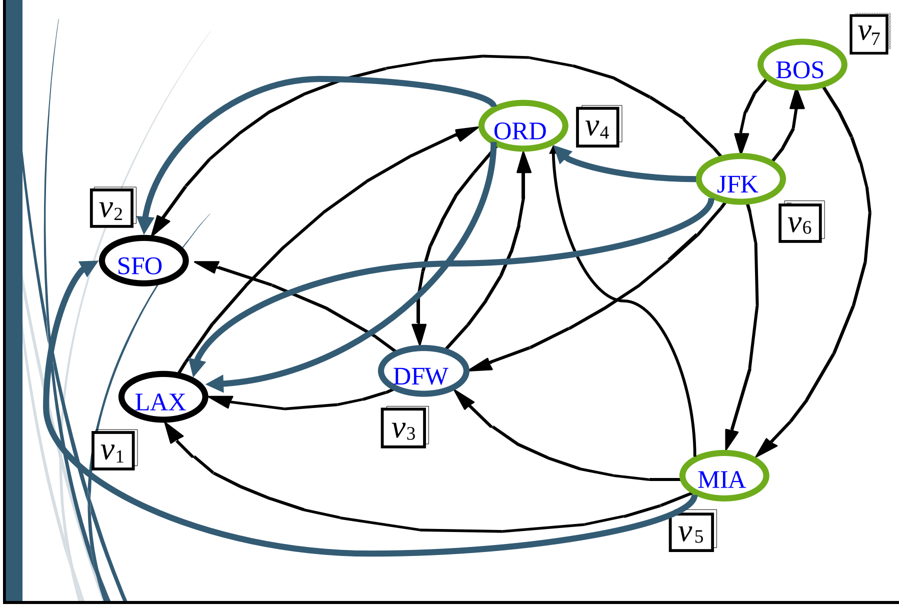
Floyd-Warshall, Iteration 1



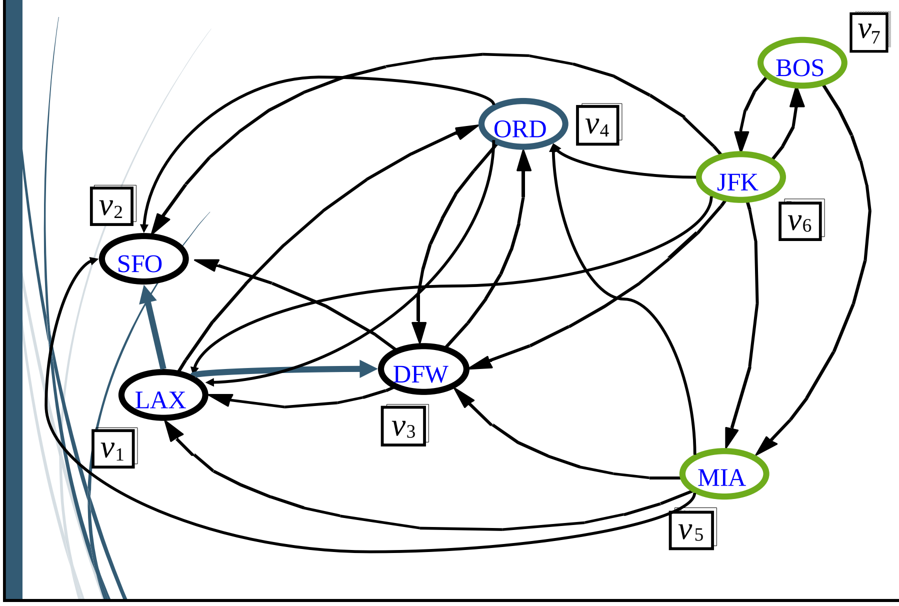
Floyd-Warshall, Iteration 2



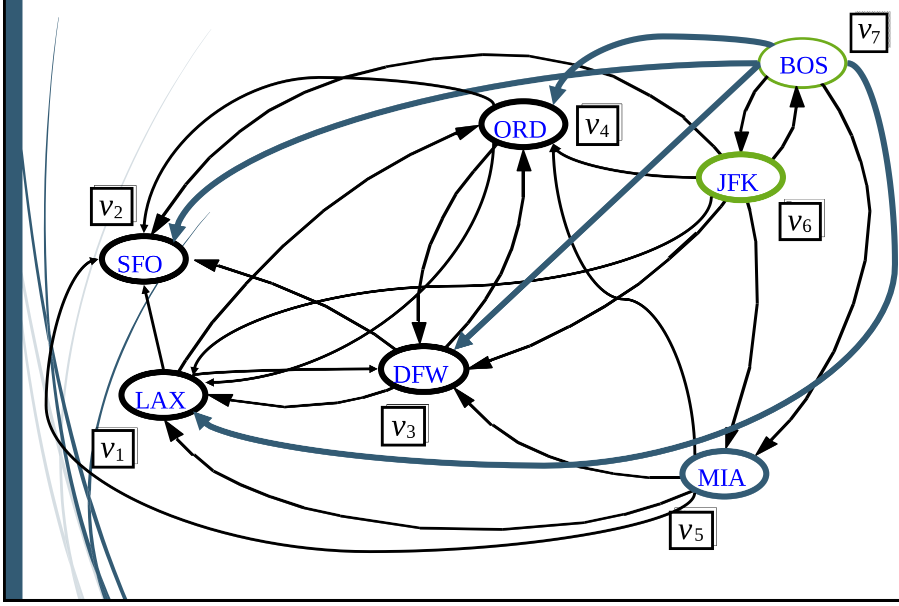
Floyd-Warshall, Iteration 3



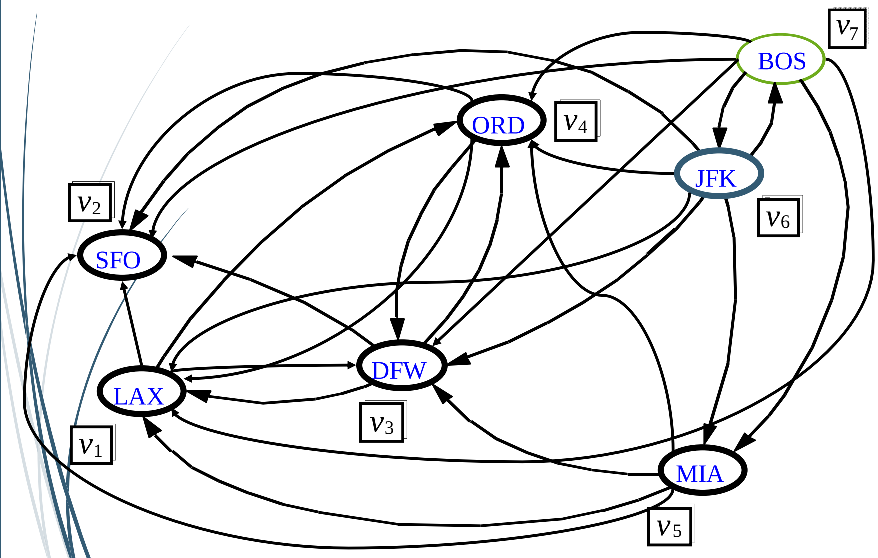
Floyd-Warshall, Iteration 4



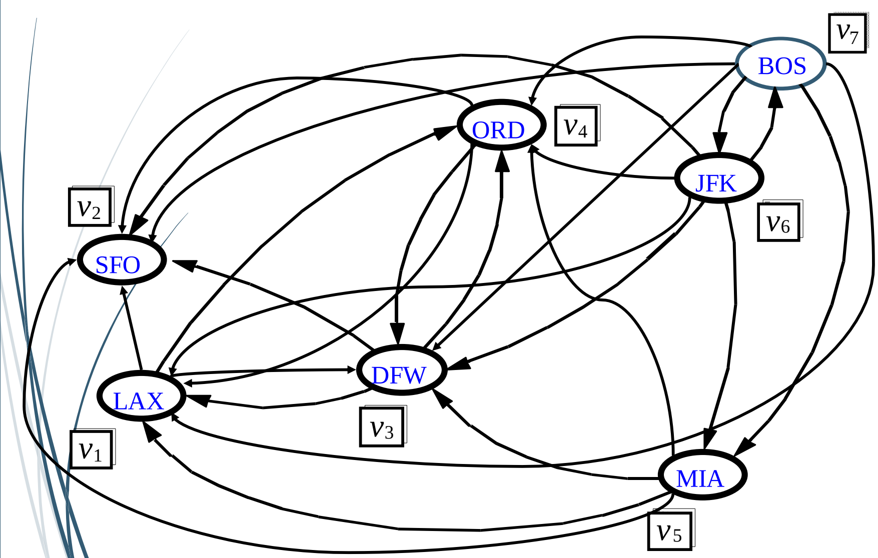
Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6



Floyd-Warshall, Conclusion



DAGs and Topological Ordering

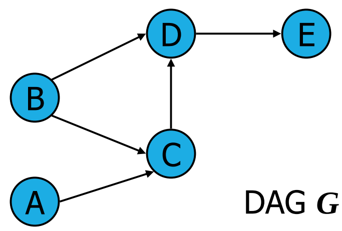
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $$v_1, \ldots, v_n$$

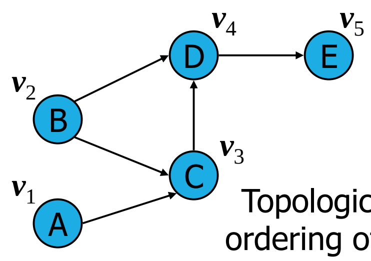
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

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

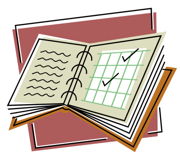


DAG G



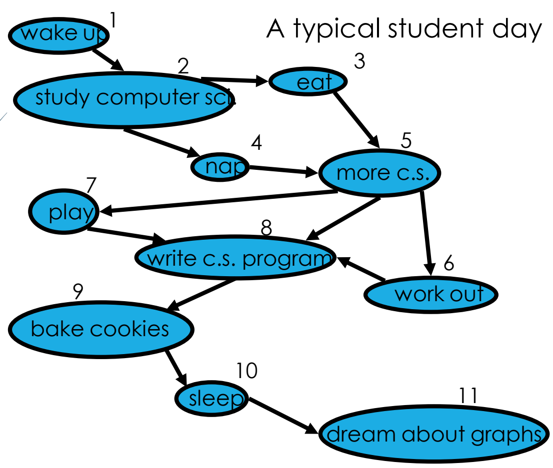
Topological ordering of G

Topological Sorting



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

A typical student day



Algorithm for Topological Sorting

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

```
Algorithm 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)

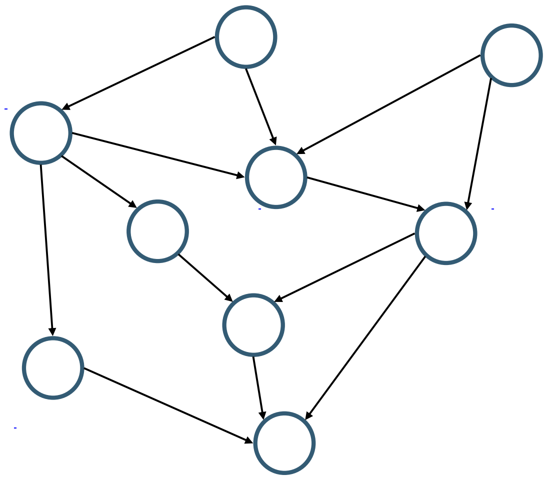
Implementation with DFS

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

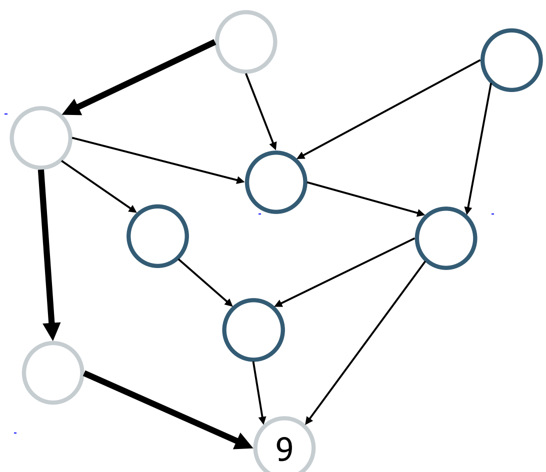
```
Algorithm topologicalDFS(G)
    Input dag G
    Output topological ordering of G
  n ← G.numVertices()
  for all  u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all  v ∈ G.vertices()
    if  getLabel(v) = UNEXPLORED
      topologicalDFS(G, v)
```

```
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
```

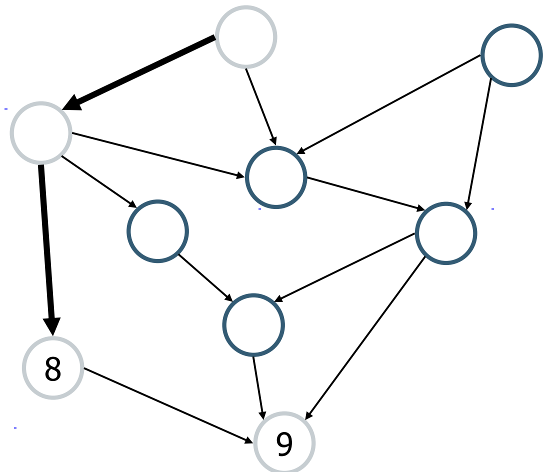
Topological Sorting Example



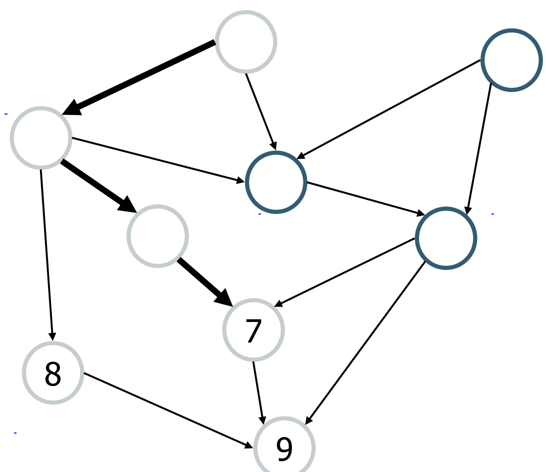
Topological Sorting Example



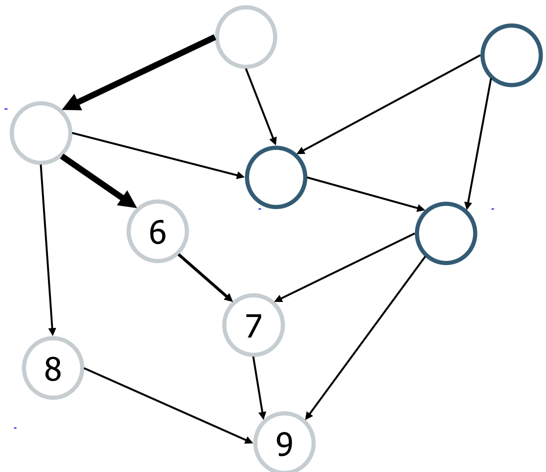
Topological Sorting Example



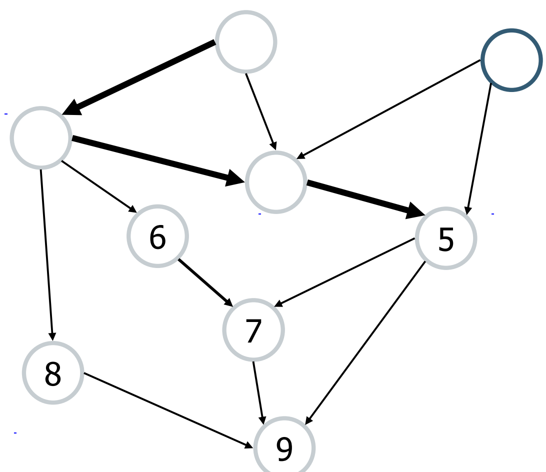
Topological Sorting Example



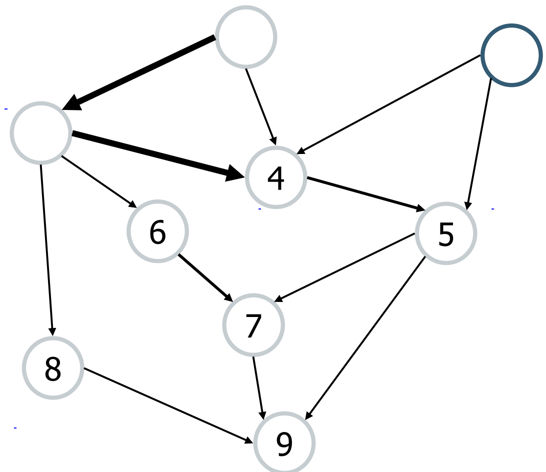
Topological Sorting Example



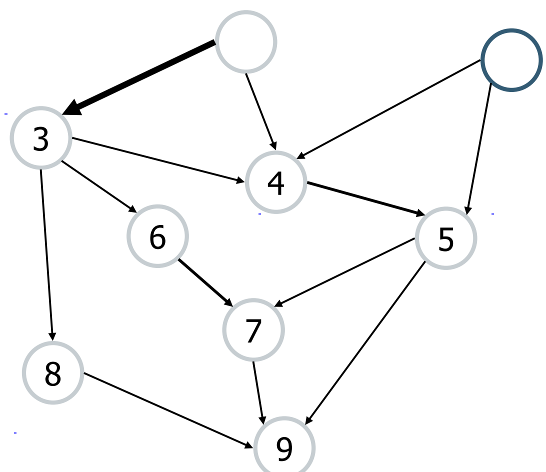
Topological Sorting Example



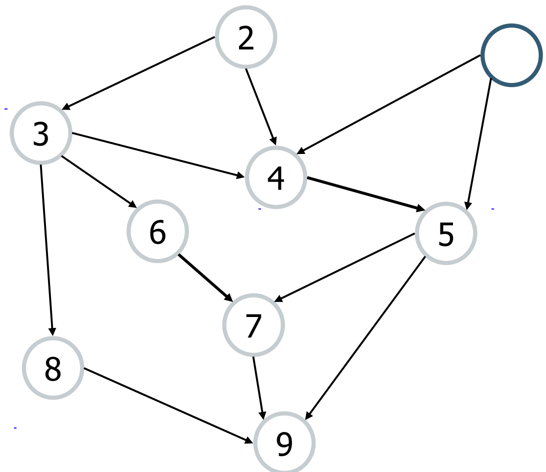
Topological Sorting Example



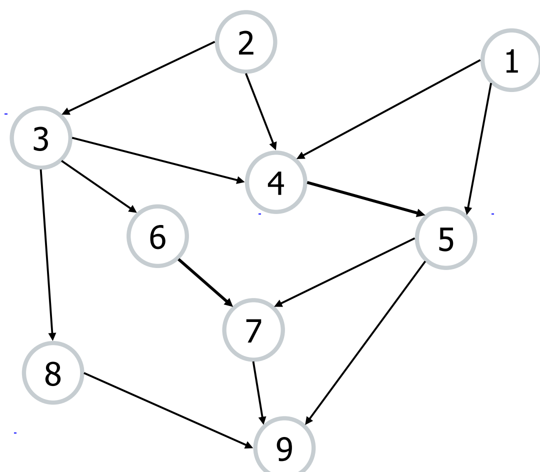
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example

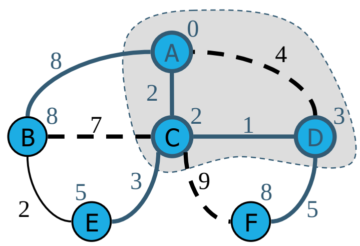


Java Implementation

```
/** Returns a list of verticies of directed acyclic graph g in topological order. */
public static <V,E> PositionalList<Vertex<V>> topologicalSort(Graph<V,E> g) {
  // list of vertices placed in topological order
  PositionalList<Vertex<V>> topo = new LinkedPositionalList<>();
  // container of vertices that have no remaining constraints
  Stack<Vertex<V>> ready = new LinkedStack<>();
  // map keeping track of remaining in-degree for each vertex
  Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>();
  for (Vertex<V> u : g.vertices()) {
    inCount.put(u, g.inDegree(u));            // initialize with actual in-degree
    if (inCount.get(u) == 0)                   // if u has no incoming edges,
      ready.push(u);                           // it is free of constraints
  }
  while (!ready.isEmpty()) {
    Vertex<V> u = ready.pop();
    topo.addLast(u);
    for (Edge<E> e : g.outgoingEdges(u)) {   // consider all outgoing neighbors of u
      Vertex<V> v = g.opposite(u, e);
      inCount.put(v, inCount.get(v) − 1);     // v has one less constraint without u
      if (inCount.get(v) == 0)
        ready.push(v);
    }
  }
  return topo;
}
```

Directed Graphs

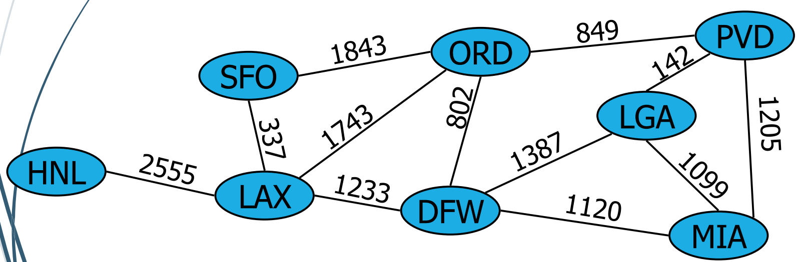
Shortest Paths



99

Weighted Graphs

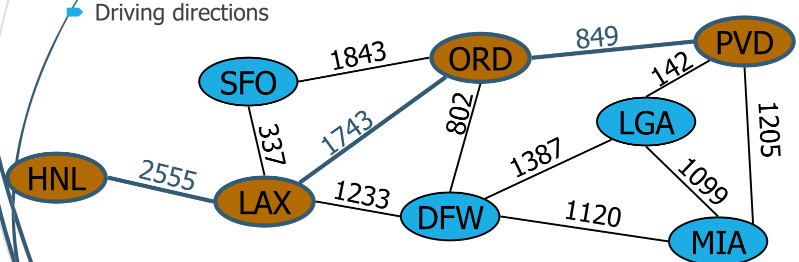
- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



100

Shortest Paths

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



101

Shortest Path Properties

Property 1:

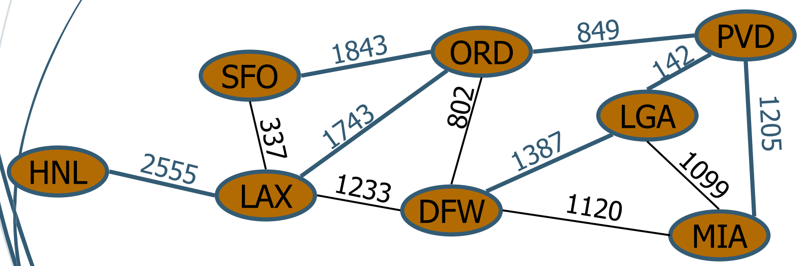
A subpath of a shortest path is itself a shortest path

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices

Example:

Tree of shortest paths from Providence



102

Dijkstra's Algorithm

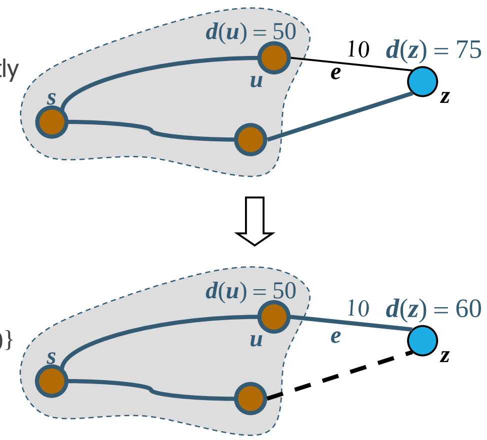
- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

103

Edge Relaxation

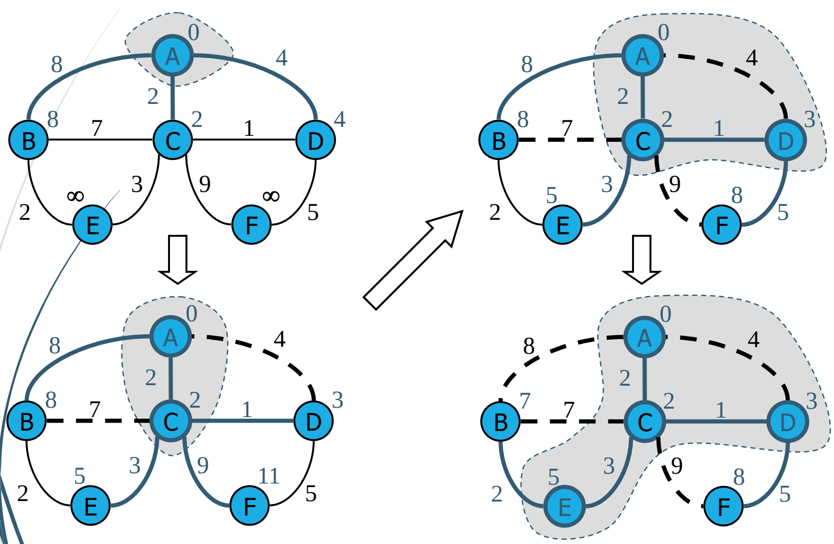
- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$



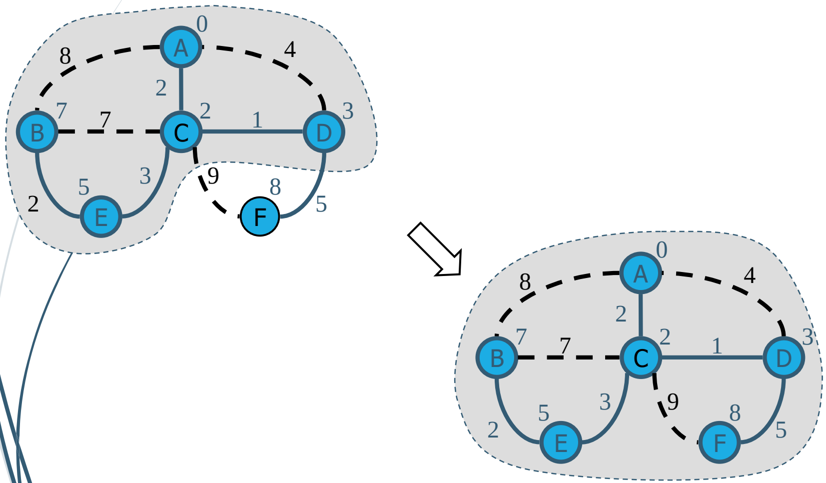
104

Example



105

Example (cont.)



Shortest Pahs

Dijkstra's Algorithm

Algorithm ShortestPath(G, s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

Output: The length of a shortest path from s to v for each vertex v of G.

```
Initialize D[s] = 0 and D[v] = ∞ for each vertex v ≠ s.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty do
    {pull a new vertex u into the cloud}
    u = value returned by Q.remove_min()
    for each vertex v adjacent to u such that v is in Q do
        {perform the relaxation procedure on edge (u,v)}
        if D[u] + w(u,v) < D[v] then
            D[v] = D[u] + w(u,v)
            Change to D[v] the key of vertex v in Q.
return the label D[v] of each vertex v
```

Analysis of Dijkstra's Algorithm

- Graph operations
  - We find all the incident edges once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list/map structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

Java Implementation

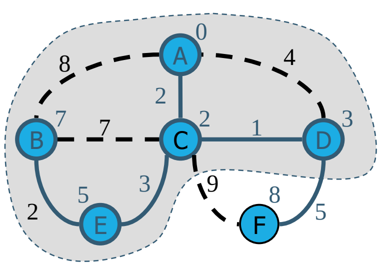
```
/** Computes shortest-path distances from src vertex to all reachable vertices of g. */
public static <V> Map<Vertex<V>, Integer>
shortestPathLengths(Graph<V,Integer> g, Vertex<V> src) {
  // d.get(v) is upper bound on distance from src to v
  Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
  // map reachable v to its d value
  Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
  // pq will have vertices as elements, with d.get(v) as key
  AdaptablePriorityQueue<Integer, Vertex<V>> pq;
  pq = new HeapAdaptablePriorityQueue<>();
  // maps from vertex to its pq locator
  Map<Vertex<V>, Entry<Integer,Vertex<V>>> pqTokens;
  pqTokens = new ProbeHashMap<>();

  // for each vertex v of the graph, add an entry to the priority queue, with
  // the source having distance 0 and all others having infinite distance
  for (Vertex<V> v : g.vertices()) {
    if (v == src)
      d.put(v,0);
    else
      d.put(v, Integer.MAX_VALUE);
    pqTokens.put(v, pq.insert(d.get(v), v));         // save entry for future updates
  }
```

Java Implementation, 2

```
// now begin adding reachable vertices to the cloud
while (!pq.isEmpty()) {
  Entry<Integer, Vertex<V>> entry = pq.removeMin();
  int key = entry.getKey();
  Vertex<V> u = entry.getValue();
  cloud.put(u, key);                                   // this is actual distance to u
  pqTokens.remove(u);                                  // u is no longer in pq
  for (Edge<Integer> e : g.outgoingEdges(u)) {
    Vertex<V> v = g.opposite(u,e);
    if (cloud.get(v) == null) {
      // perform relaxation step on edge (u,v)
      int wgt = e.getElement();
      if (d.get(u) + wgt < d.get(v)) {                 // better path to v?
        d.put(v, d.get(u) + wgt);                      // update the distance
        pq.replaceKey(pqTokens.get(v), d.get(v));      // update the pq entry
      }
    }
  }
}
return cloud;              // this only includes reachable vertices
}
```

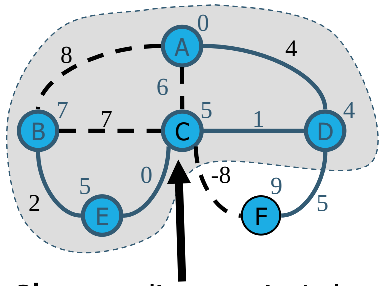
Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex



Why It Doesn't Work for Negative-Weight Edges

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!

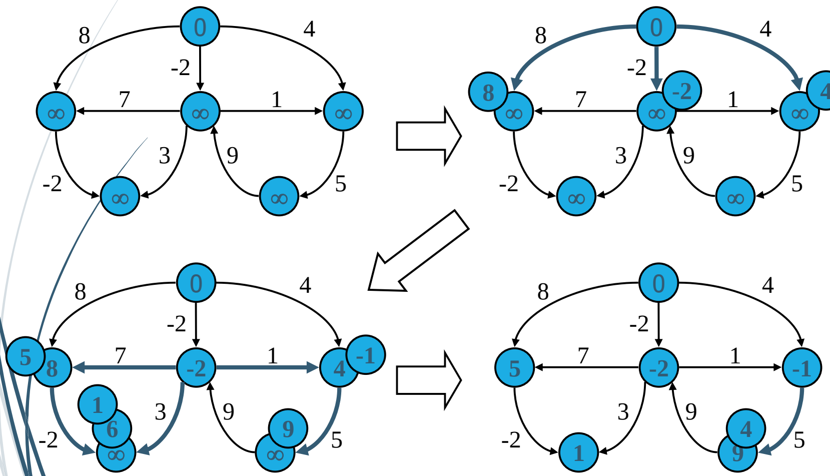
Bellman-Ford Algorithm (not in book)

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists
  - How?

```
Algorithm BellmanFord(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  for i ← 1 to n − 1 do
    for each e ∈ G.edges()
      { relax edge e }
      u ← G.origin(e)
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
```

Bellman-Ford Example

Nodes are labeled with their d(v) values



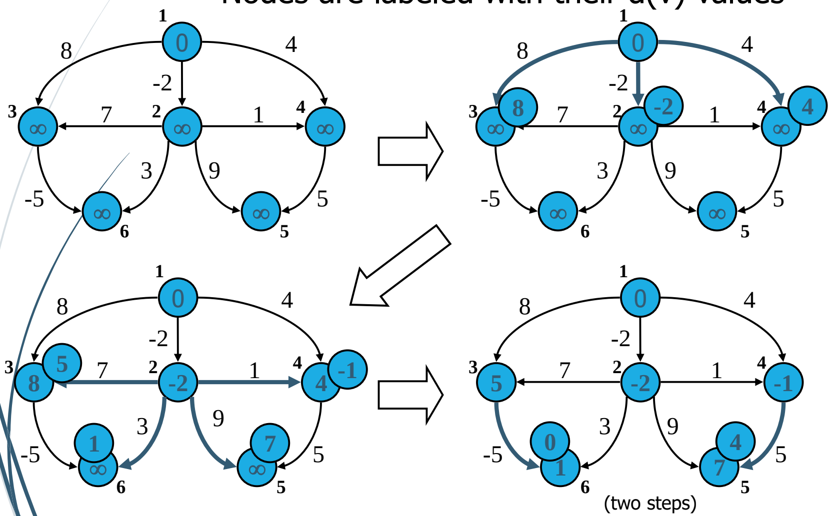
DAG-based Algorithm (not in book)

- Works even with negative-weight edges
- Uses topological order
- Doesn't use any fancy data structures
- Is much faster than Dijkstra's algorithm
- Running time: O(n+m).

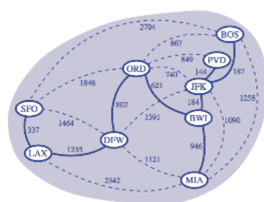
```
Algorithm DagDistances(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  { Perform a topological sort of the vertices }
  for u ← 1 to n do    {in topological order}
    for each  e ∈ G.outEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if  r < getDistance(z)
        setDistance(z,r)
```

DAG Example

Nodes are labeled with their d(v) values



Minimum Spanning Trees



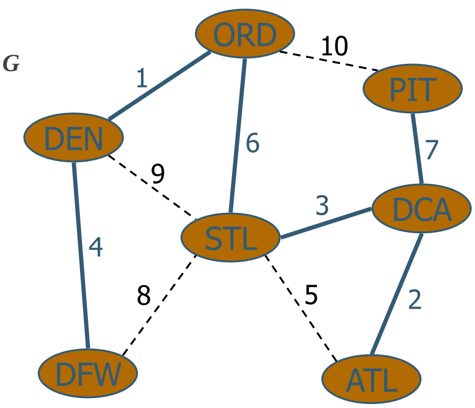
Minimum Spanning Trees

Spanning subgraph
- Subgraph of a graph G containing all the vertices of G

Spanning tree
- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)
- Spanning tree of a weighted graph with minimum total edge weight

- Applications
  - Communications networks
  - Transportation networks

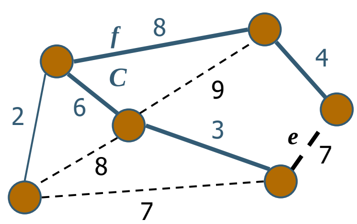


Minimum Spanning Trees

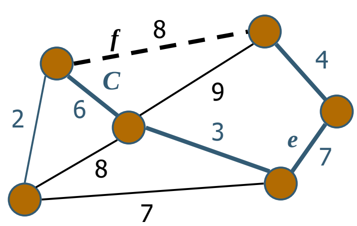
Cycle Property

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



Replacing f with e yields a better spanning tree



Minimum Spanning Trees

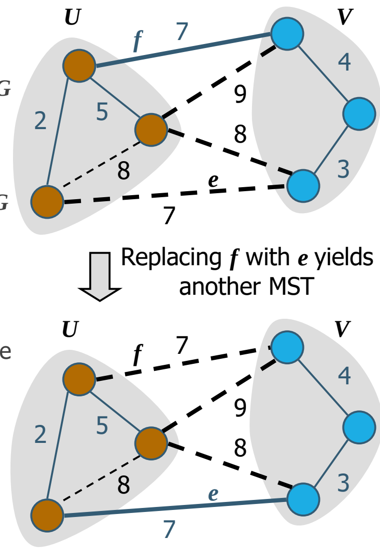
Partition Property

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***

Proof:

- Let ***T*** be an MST of ***G***
- If ***T*** does not contain ***e***, consider the cycle ***C*** formed by ***e*** with ***T*** and let ***f*** be an edge of ***C*** across the partition
- By the cycle property,
  weight(***f***) ≤ ***weight***(***e***)
- Thus, ***weight***(***f***) = ***weight***(***e***)
- We obtain another MST by replacing ***f*** with ***e***



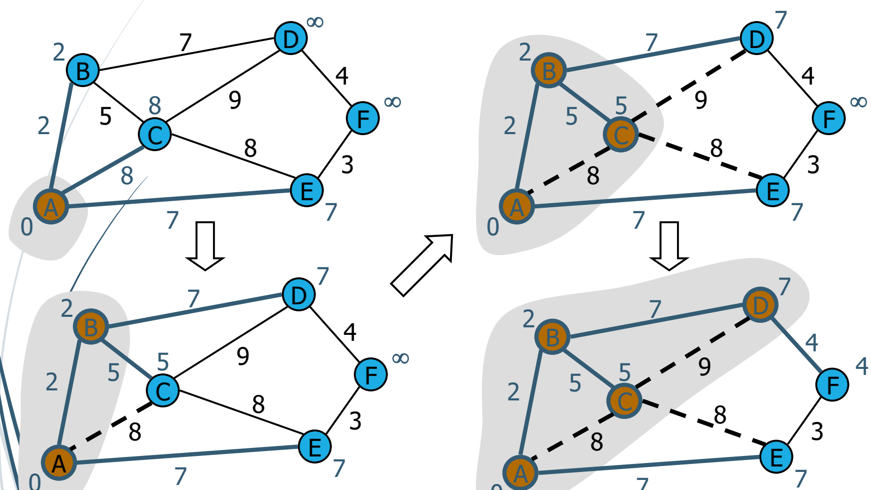
Prim-Jarnik's Algorithm

- Similar to Dijkstra'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 Pseudo-code

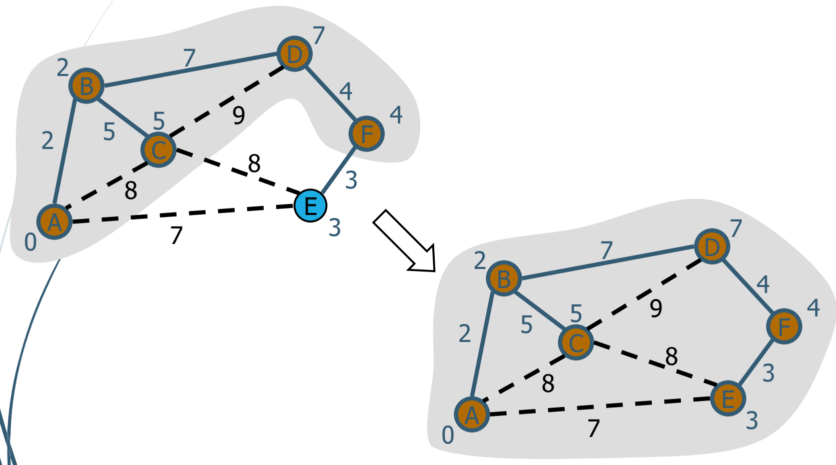
Algorithm PrimJarnik(G):
 Input: An undirected, weighted, connected graph G with n vertices and m edges
 Output: A minimum spanning tree T for G
Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
 $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
 $(u, e) =$ value returned by Q.remove_min()
 Connect vertex u to T using edge e.
 for each edge $e' = (u, v)$ such that v is in Q **do**
  {check if edge (u, v) better connects v to T}
  if $w(u, v) < D[v]$ **then**
   $D[v] = w(u, v)$
   Change the key of vertex v in Q to $D[v]$.
   Change the value of vertex v in Q to (v, e').
return the tree T

Example



Minimum Spanning Trees

Example (contd.)



Analysis

- Graph operations
  - We cycle through the incident edges once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

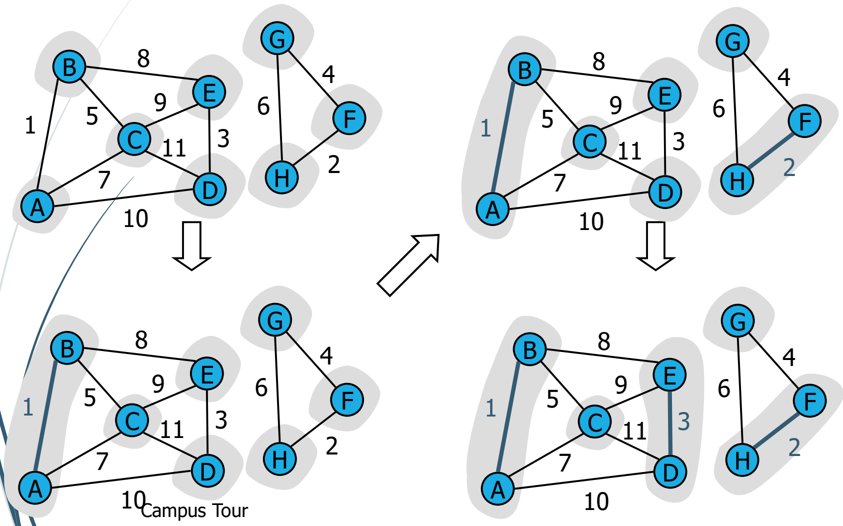
Kruskal's Approach

- 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

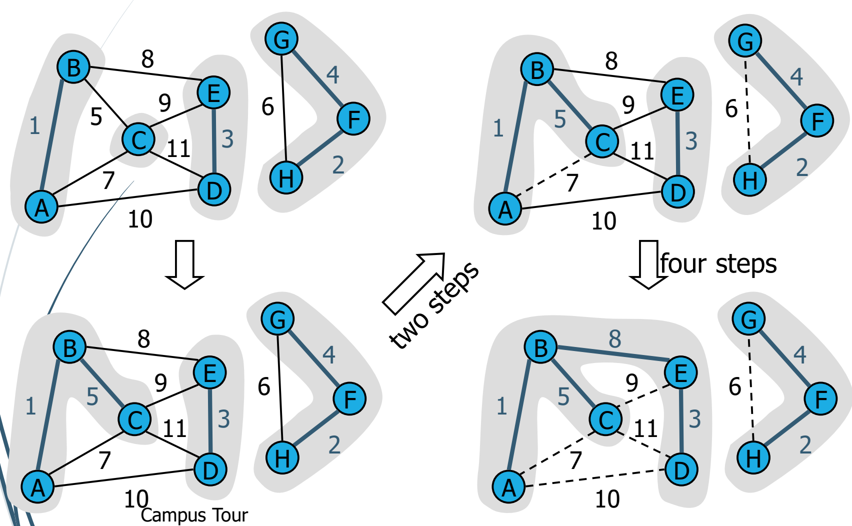
Kruskal's Algorithm

Algorithm Kruskal(G):
  Input: A simple connected weighted graph G with n vertices and m edges
  Output: A minimum spanning tree T for G
 for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
 Initialize a priority queue Q to contain all edges in G, using the weights as keys.
 $T = \emptyset$ $\{T$ will ultimately contain the edges of the MST$\}$
 while T has fewer than $n-1$ edges **do**
  (u,v) = value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
   Add edge (u,v) to T.
   Merge $C(u)$ and $C(v)$ into one cluster.
 return tree T

Example



Example (contd.)

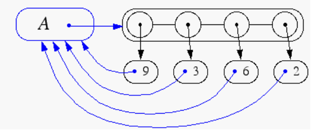


Data Structure for Kruskal's Algorithm

- The algorithm maintains a forest of trees
- A priority queue extracts the edges by increasing weight
- An edge is accepted it if connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with operations:
  - makeSet(u): create a set consisting of u
  - find(u): return the set storing u
  - union(A, B): replace sets A and B with their union

Minimum Spanning Trees

List-based Partition



- Each set is stored in a sequence
- Each element has a reference back to the set
  - operation find(u) takes O(1) time, and returns the set of which u is a member.
  - in operation union(A,B), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation union(A,B) is min(|A|, |B|)
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most log n times

Minimum Spanning Trees

Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
  - Cluster merges as unions
  - Cluster locations as finds
- Running time $O((n + m) \log n)$
  - Priority Queue operations: $O(m \log n)$
  - Union-Find operations: $O(n \log n)$

Minimum Spanning Trees

Java Implementation

```
/** Computes a minimum spanning tree of graph g using Kruskal's algorithm. */
public static <V> PositionalList<Edge<Integer>> MST(Graph<V,Integer> g) {
  // tree is where we will store result as it is computed
  PositionalList<Edge<Integer>> tree = new LinkedPositionalList<>();
  // pq entries are edges of graph, with weights as keys
  PriorityQueue<Integer, Edge<Integer>> pq = new HeapPriorityQueue<>();
  // union-find forest of components of the graph
  Partition<Vertex<V>> forest = new Partition<>();
  // map each vertex to the forest position
  Map<Vertex<V>,Position<Vertex<V>>> positions = new ProbeHashMap<>();

  for (Vertex<V> v : g.vertices())
    positions.put(v, forest.makeGroup(v));

  for (Edge<Integer> e : g.edges())
    pq.insert(e.getElement(), e);

```

Minimum Spanning Trees

Java Implementation, 2

```java
  int size = g.numVertices();
  // while tree not spanning and unprocessed edges remain...
  while (tree.size() != size − 1 && !pq.isEmpty()) {
    Entry<Integer, Edge<Integer>> entry = pq.removeMin();
    Edge<Integer> edge = entry.getValue();
    Vertex<V>[ ] endpoints = g.endVertices(edge);
    Position<Vertex<V>> a = forest.find(positions.get(endpoints[0]));
    Position<Vertex<V>> b = forest.find(positions.get(endpoints[1]));
    if (a != b) {
      tree.addLast(edge);
      forest.union(a,b);
    }
  }

  return tree;
}
```

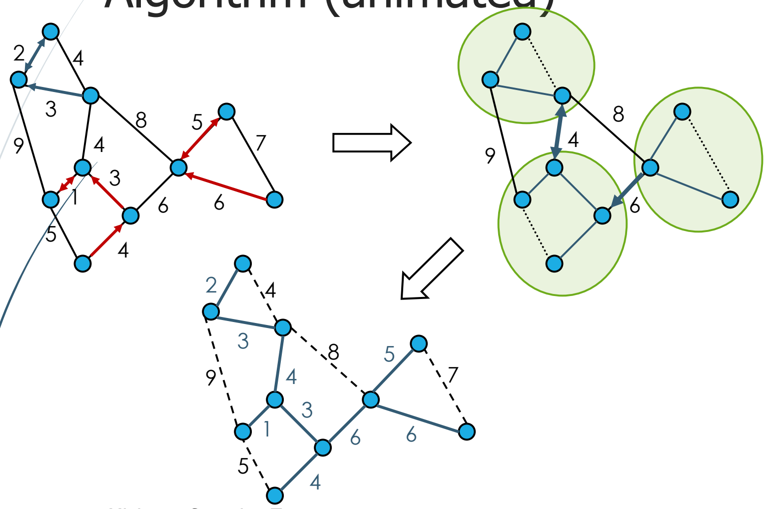

Baruvka's Algorithm (Exercise)

- 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

135

Example of Baruvka's Algorithm (animated)



Minimum Spanning Trees