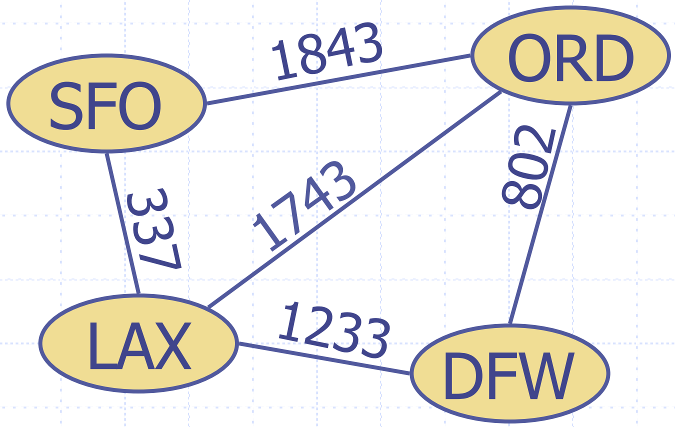
Graphs



What is a Graph?

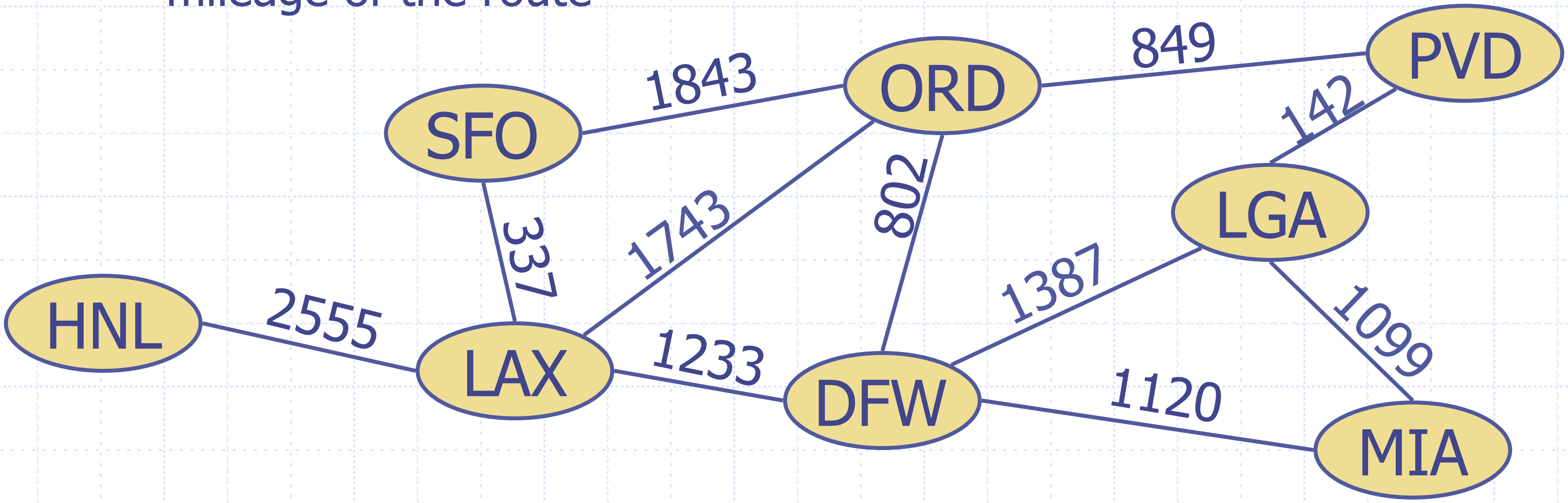
(in computer science, it's not a data plot)

General structure for representing positions with an arbitrary connectivity structure

- Collection of *vertices* (nodes) and *edges* (arcs)
  - Edge is a pair of vertices - it connects the two vertices, making them *adjacent*
- A tree is a special type of graph!

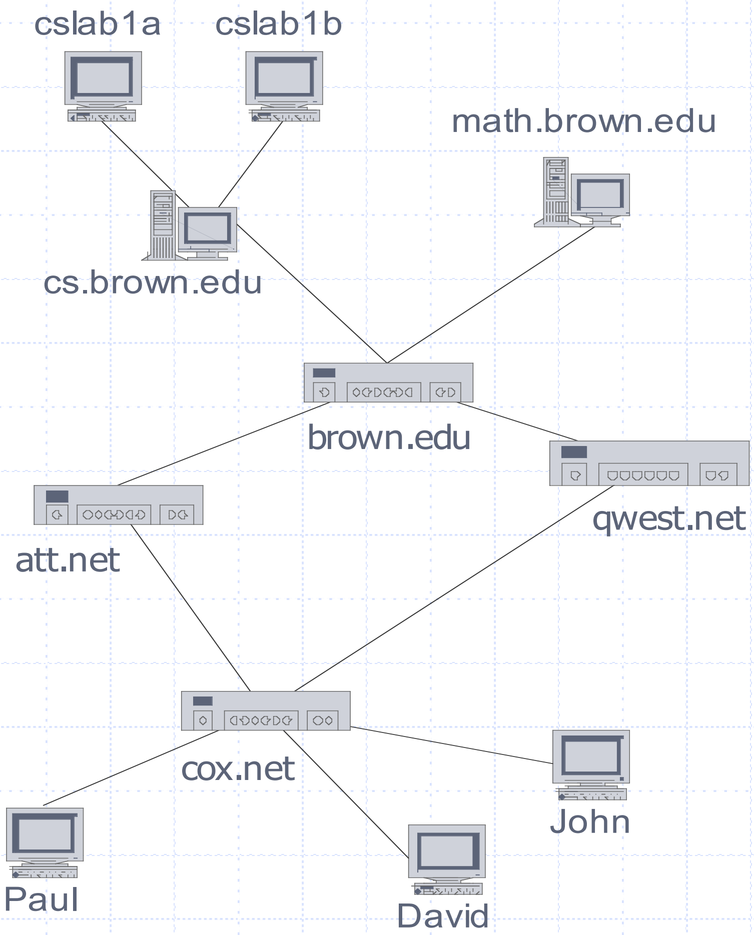
Graphs

- 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



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



What can we do with graphs?

Find a *path* from one place to another

Determine connectivity

Find the *shortest path* from one place to another

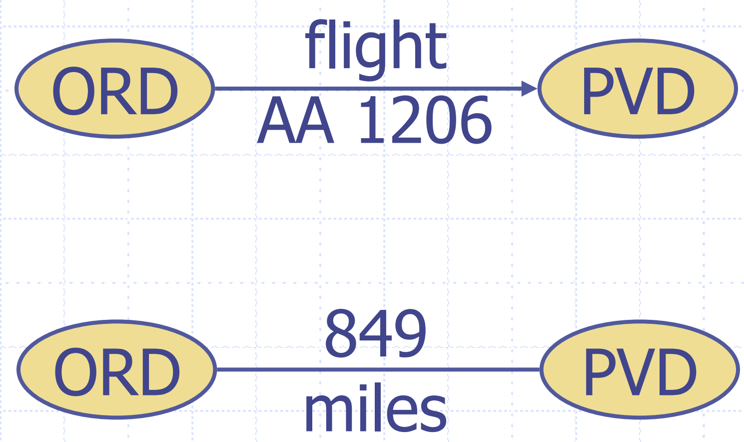
Find the "weakest link" (min cut)

- check amount of redundancy in case of failures

Find the amount of flow that will go through them

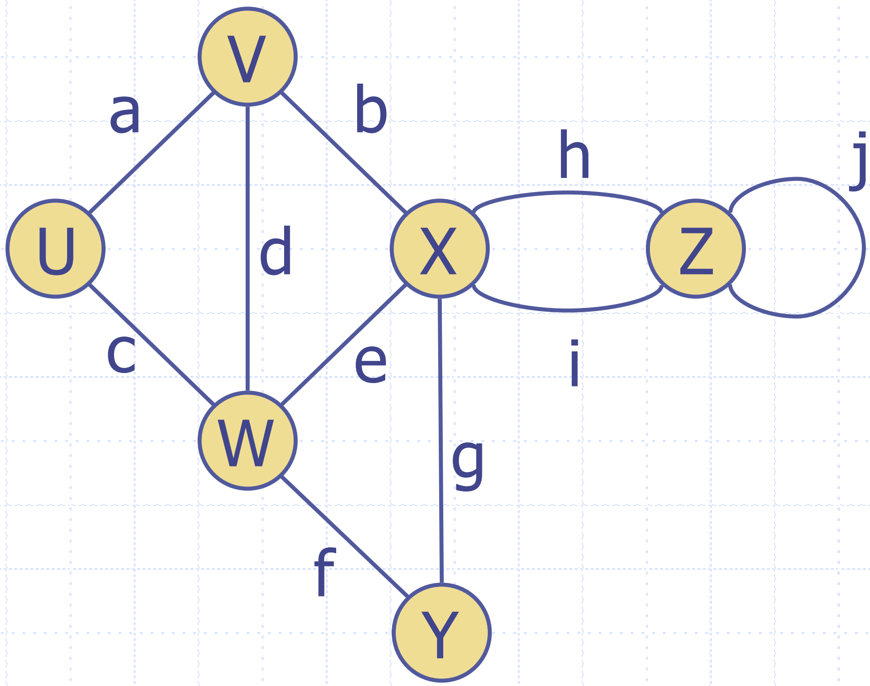
Edge Types

- 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



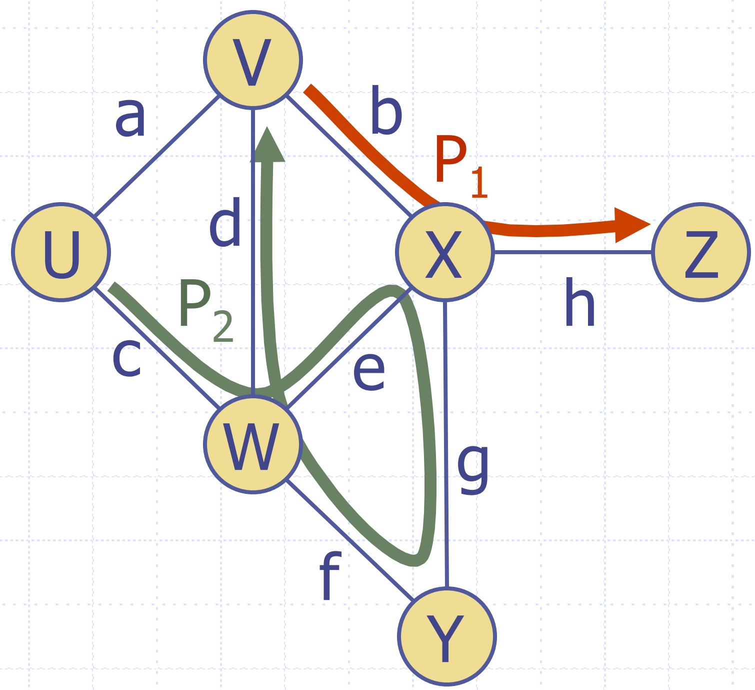
Terminology

- 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
- Simple Graph
  - No self-loops or parallel edges



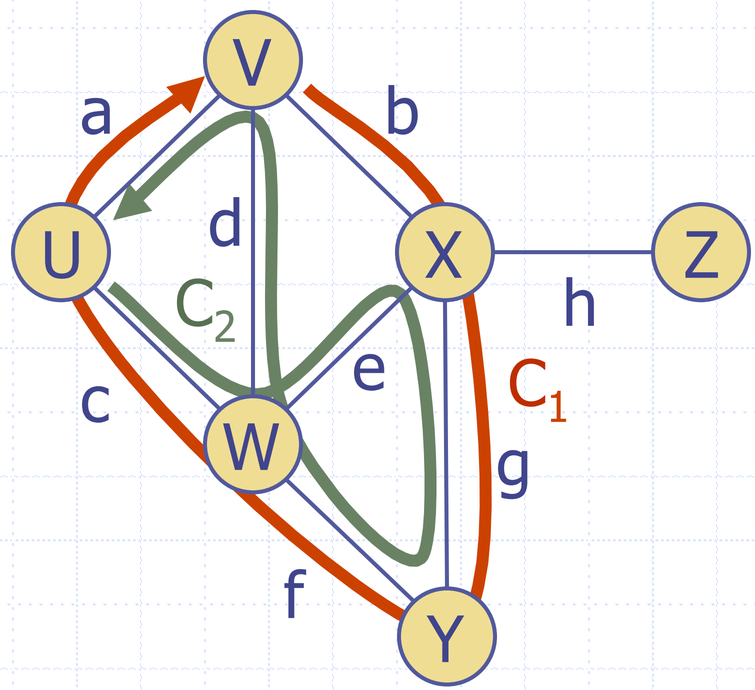
Terminology (cont.)

- 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



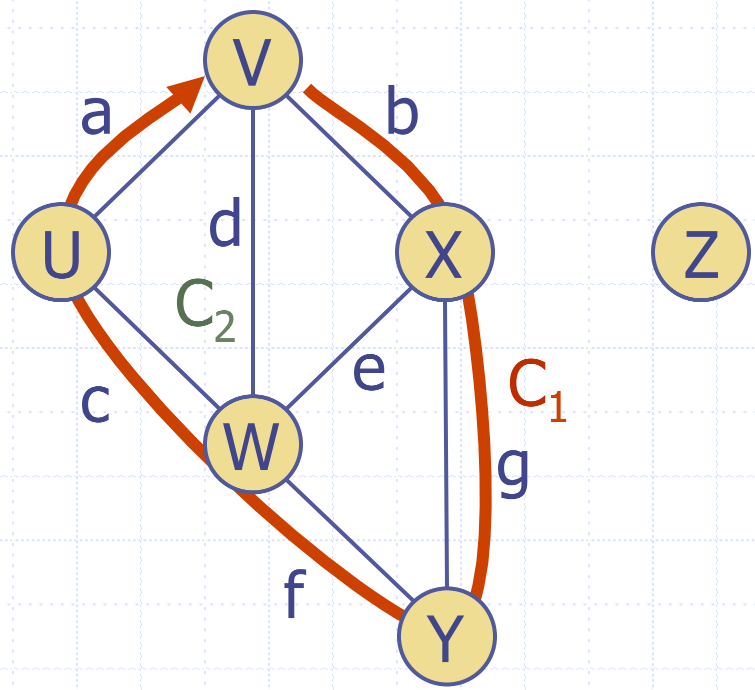
Terminology (cont.)

- 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



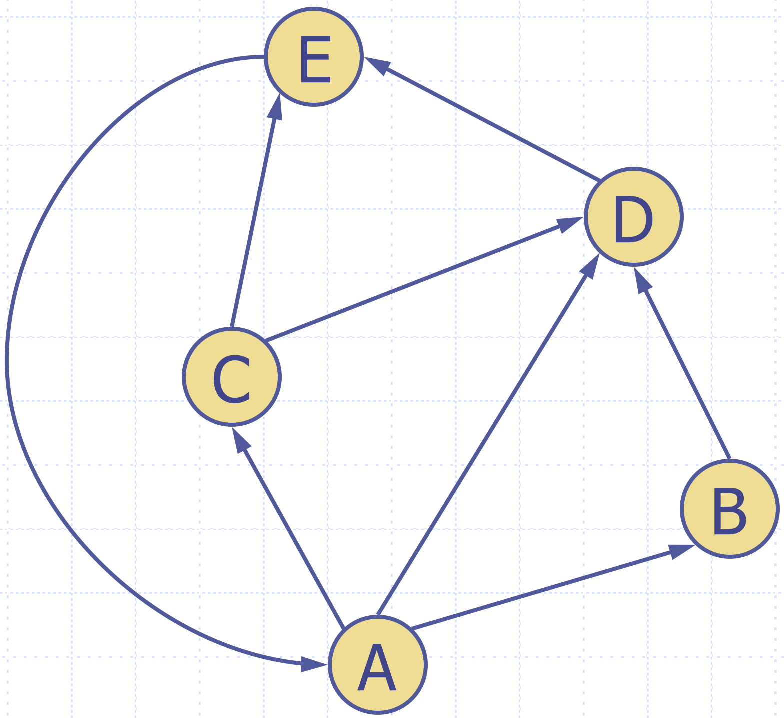
Terminology (cont.)

- Connected
  - A path from every node to every other node
  - Digraph is strongly connected if directed path
  - Digraph is weakly connected if undirected path
- Complete
  - An edge between every node
- Sparse: |E| = O(V)
- Question: What is the min and max # of edges in a fully connected simple graph?

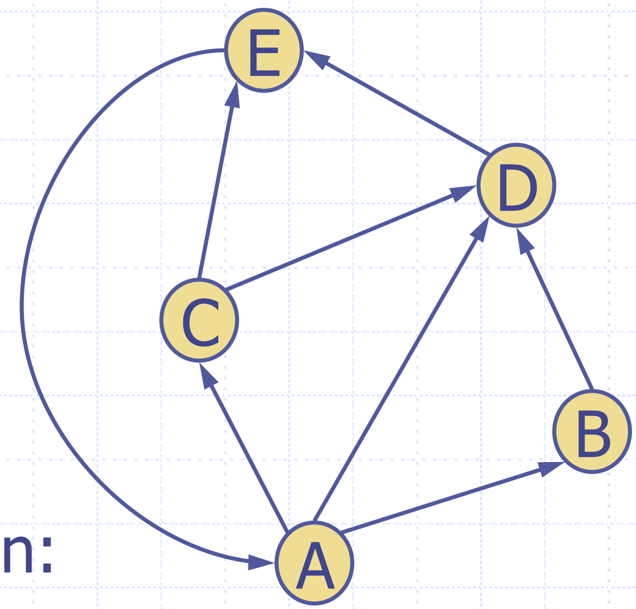


Digraphs

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



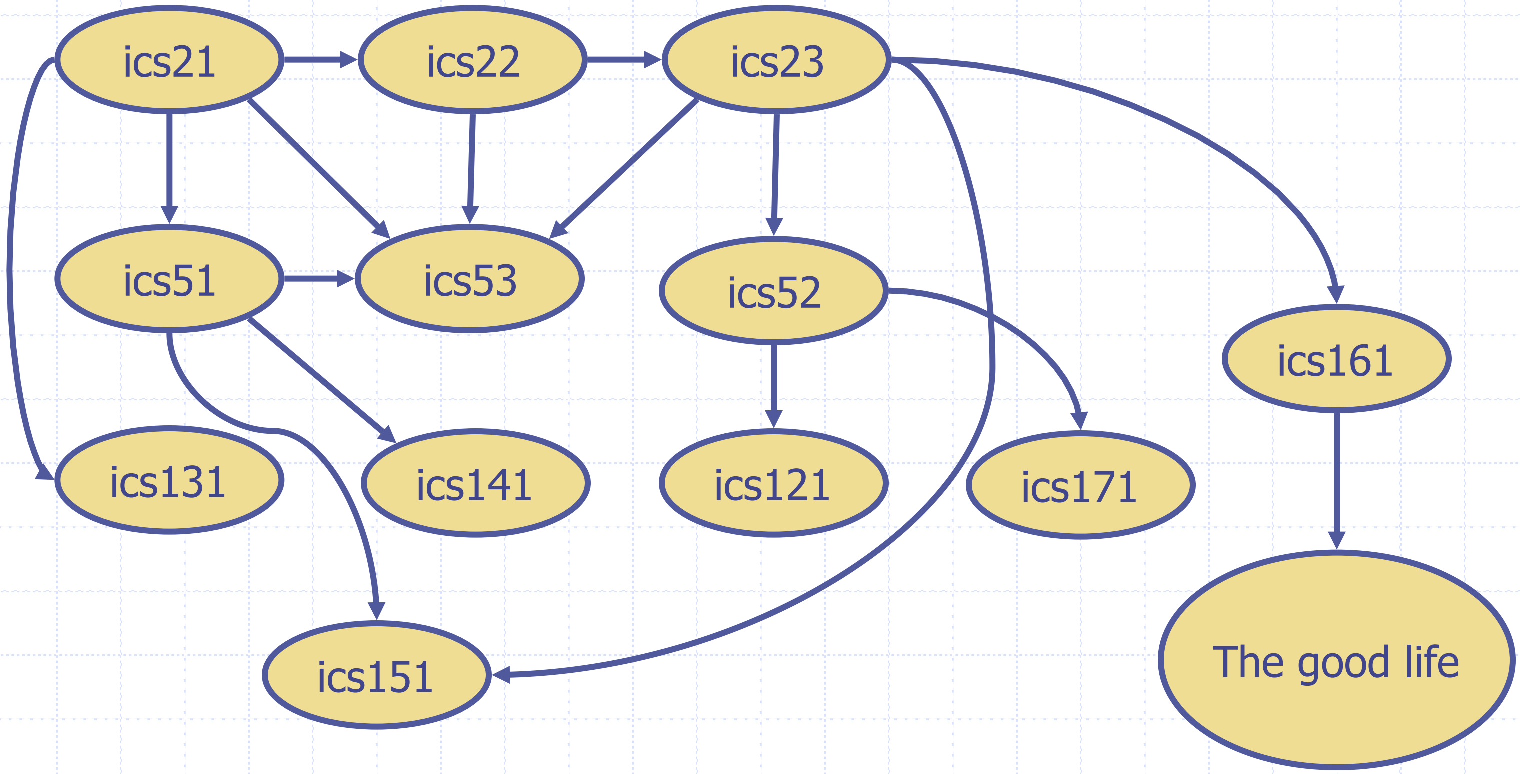
Digraph Properties



- 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, $m \leq n*(n-1)$.
- 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.

Digraph Application

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



Properties

Property 1

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

Proof: each edge is counted twice

Property 2

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

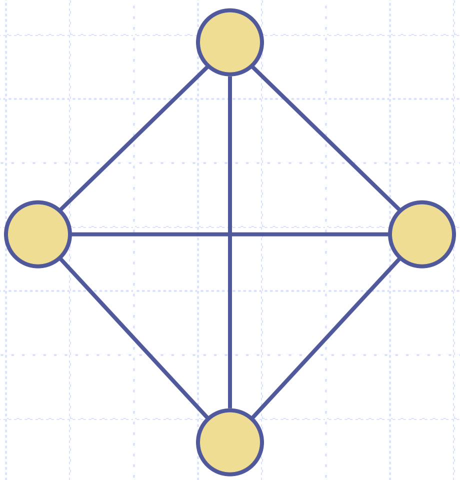
$m \le n\,(n - 1)/2$

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

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

Concrete graph representations

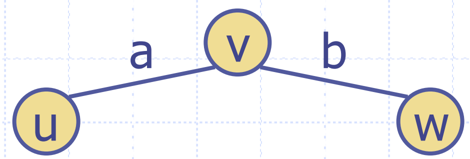
- Edge List: simple but inefficient in time
- Adjacency List: moderately simple and efficient
- Adjacency Matrix: simple but inefficient in space

Adjacency List

- Similar to Edge List
- Each vertex also has container of references to incident edges

Adjacency List Structure

- Incidence sequence for each vertex
  - sequence of references to vertex objects of incident edges
- Augmented edge objects
  - references to edges which in turn provide references to adjacent nodes



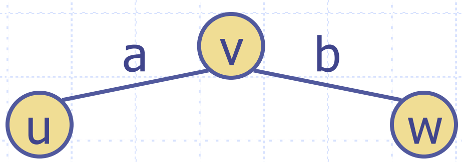
v	u, w
u	v
w	v

Adjacency list (linked list) efficiency

- vertices( ) : $O(n)$
- edges( ): $O(m)$
- endVertices(e): $O(1)$
- incidentEdges(v): $O(\deg(v))$
- areAdjacent(v, w): $O(\min(\deg(v), \deg(w)))$
- removeEdge(e): $O(\deg(u)+\deg(v))$ (can be O(1) with back links
- $e = (u,v)$
- removeVertex(v): $O(\deg(v) + \Sigma \deg(u))$ (can be O(deg(v)) with back links)
- $u \in \text{adj}(v)$

Adjacency Matrix

- Extend edge list with *v* x *v* array
  - each entry holds null reference or reference to edge connected vertex *i* to vertex *j*



	v	u	w
v	∅	**a**	∅
u	**a**	∅	**b**
w	∅	**b**	∅

Adjacency Matrix efficiency

- vertices( ) : $O(n)$
- edges( ): $O(m)$
- endVertices(e): $O(1)$
- incidentEdges(v): $O(n)$
- areAdjacent(v, w): $O(1)$
- removeEdge(e): $O(1)$
- removeVertex(v): $O(n^2)$
  - perhaps $O(n)$ with amortization

Asymptotic Performance

◆ n vertices, m edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	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

DAGs and Topological Ordering

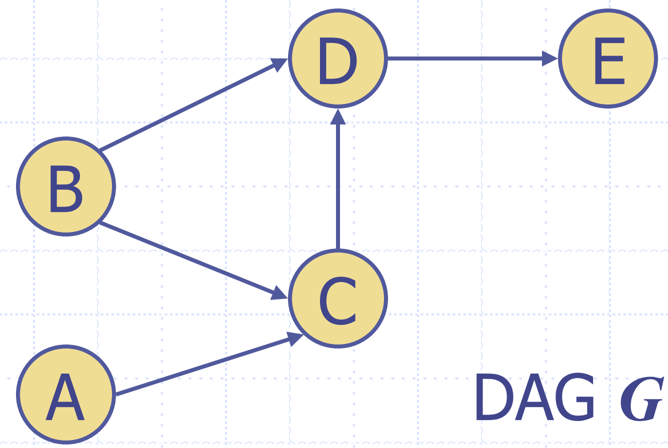
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

$$v_1, \ldots, v_n$$

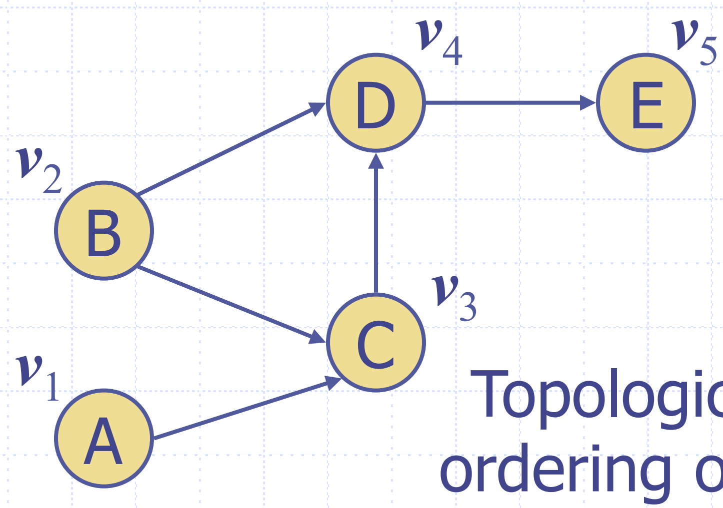
of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

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

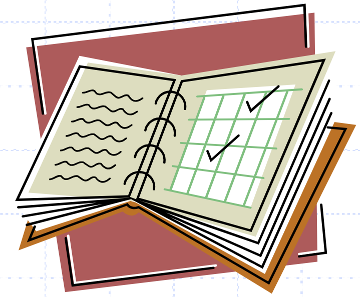


DAG G

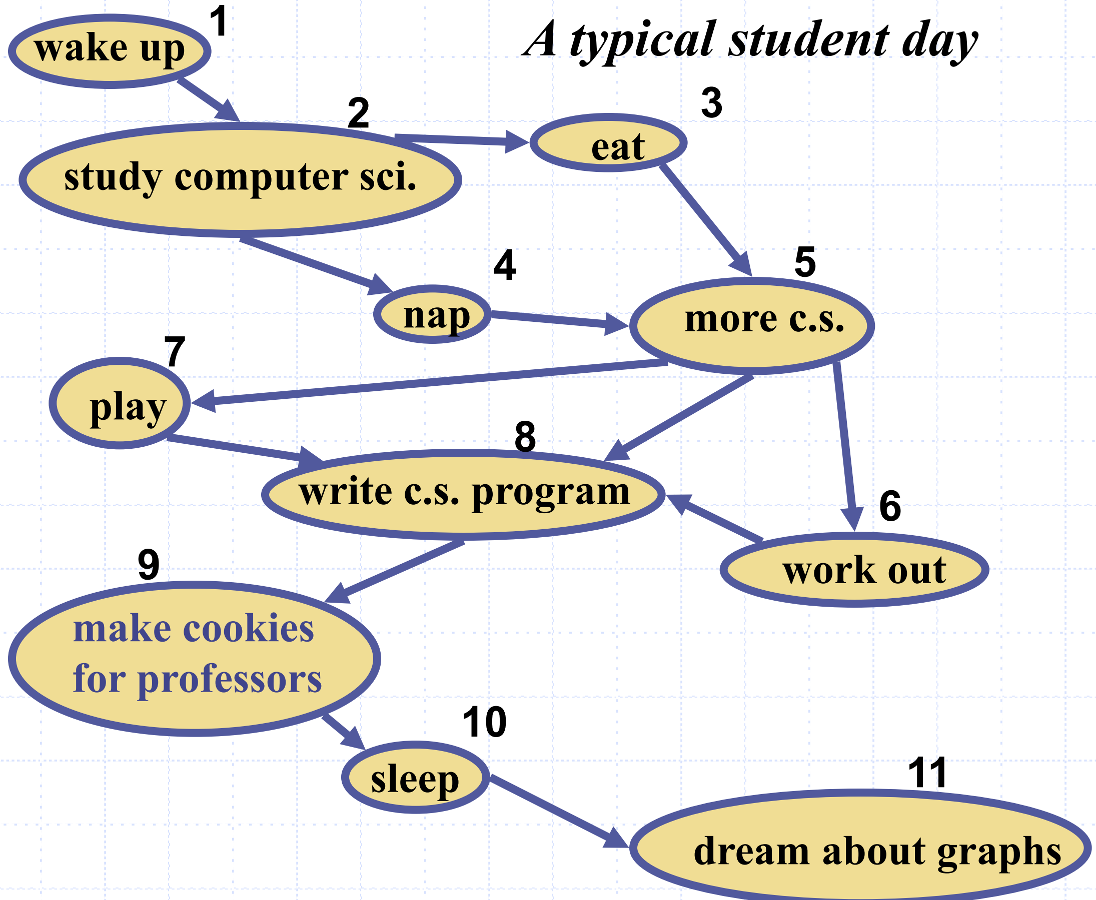


Topological ordering of G

Topological Sorting



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

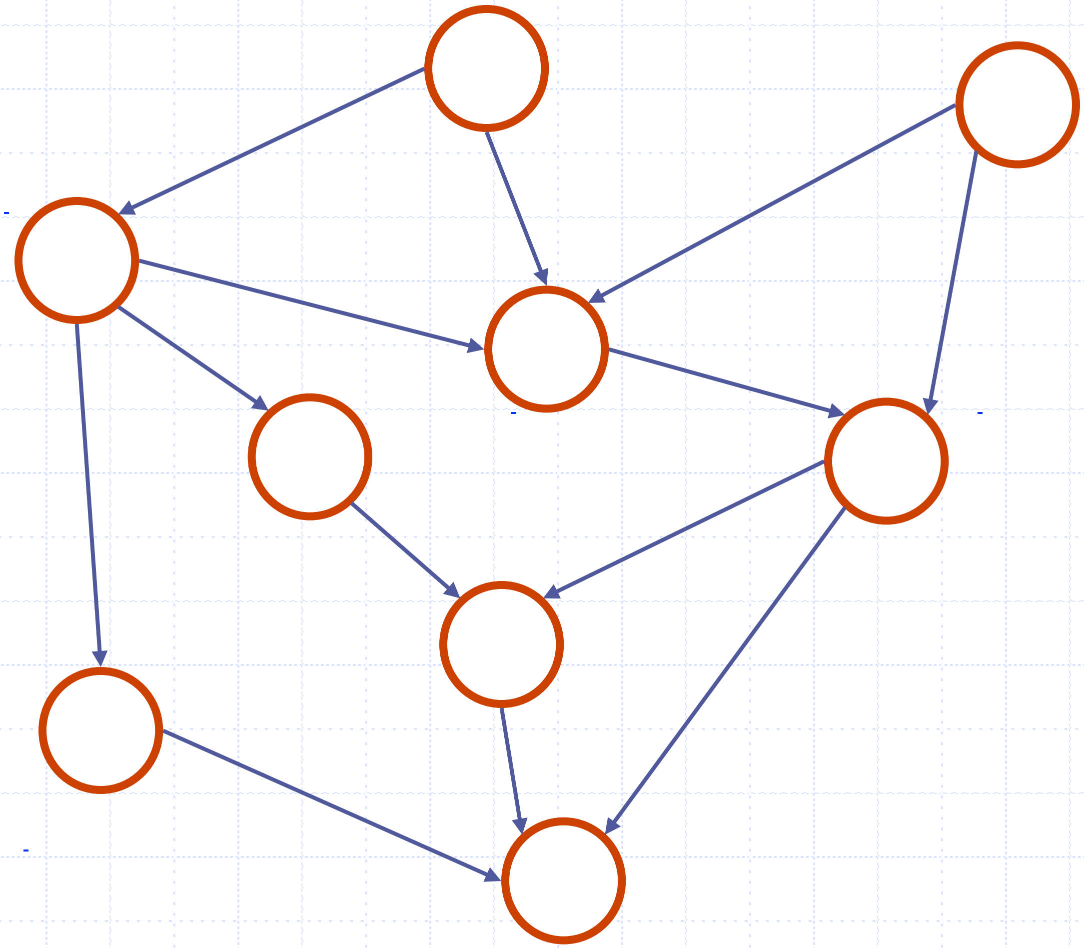


Algorithm for Topological Sorting

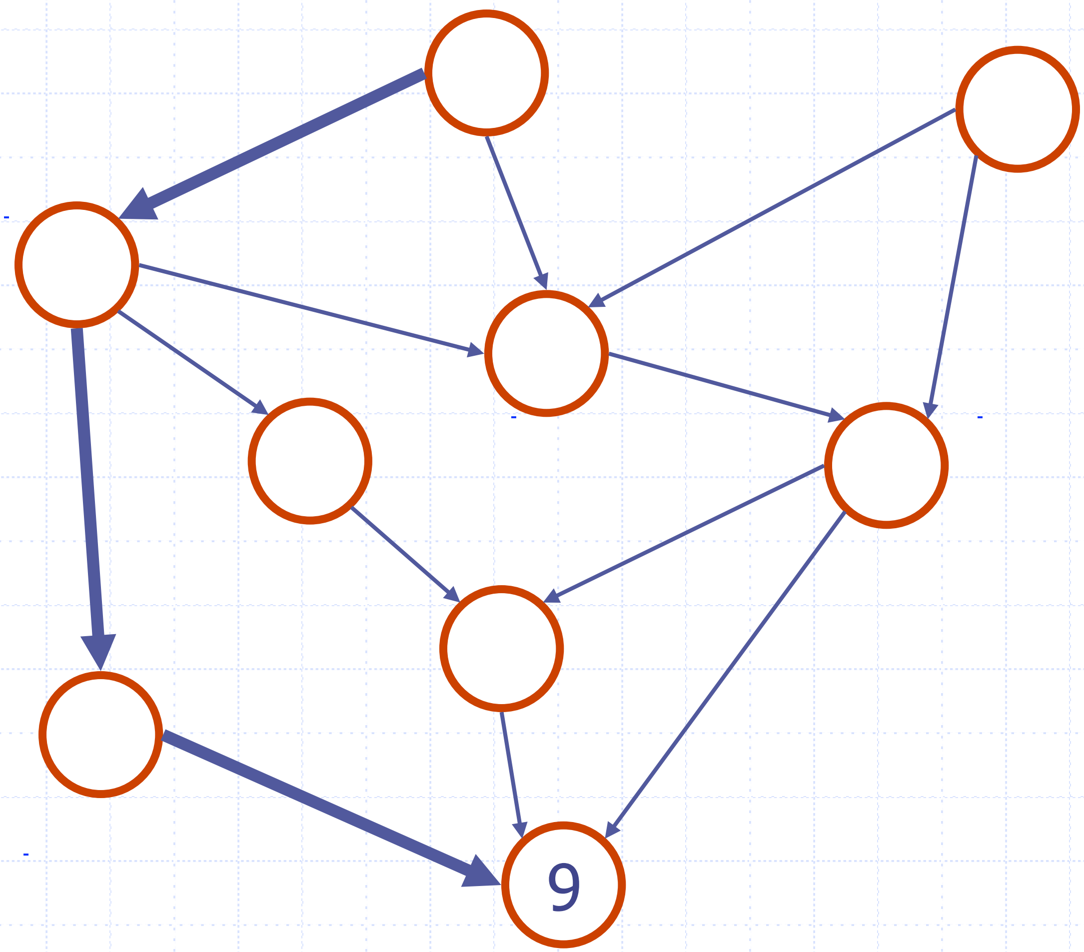
```
TopologicalSort(G)
    counter = 0; q is empty queue
    for all v in G
        if (indegree(v) == 0)
                q.enqueue(v)
    while q is not empty do
        v = q.dequeue
        v.index = ++counter;
        for each w adjacent to v
                w.indegree—
                if (w.indegree == 0)
                        q.enqueue(w)

    if (counter != G.size())
        throw cycleFoundException
```

- Running time: ???

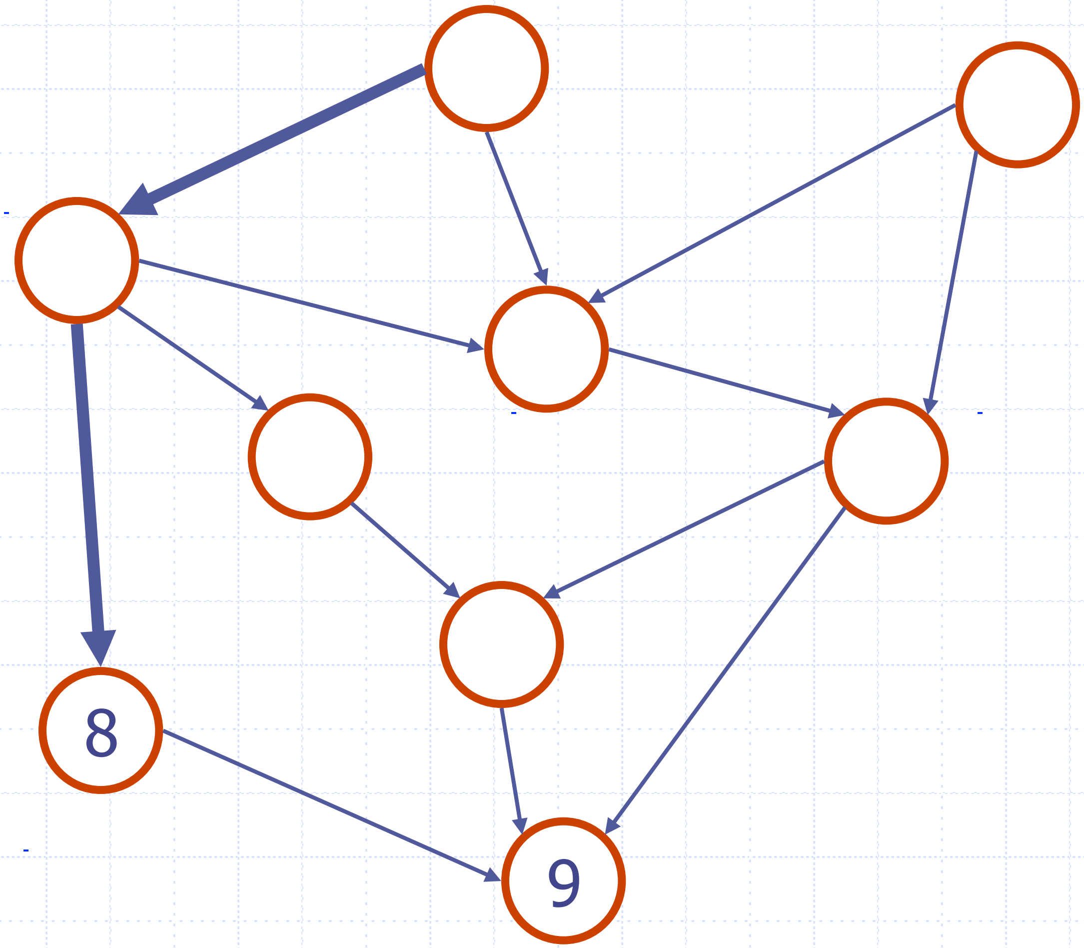
Topological Sorting Example



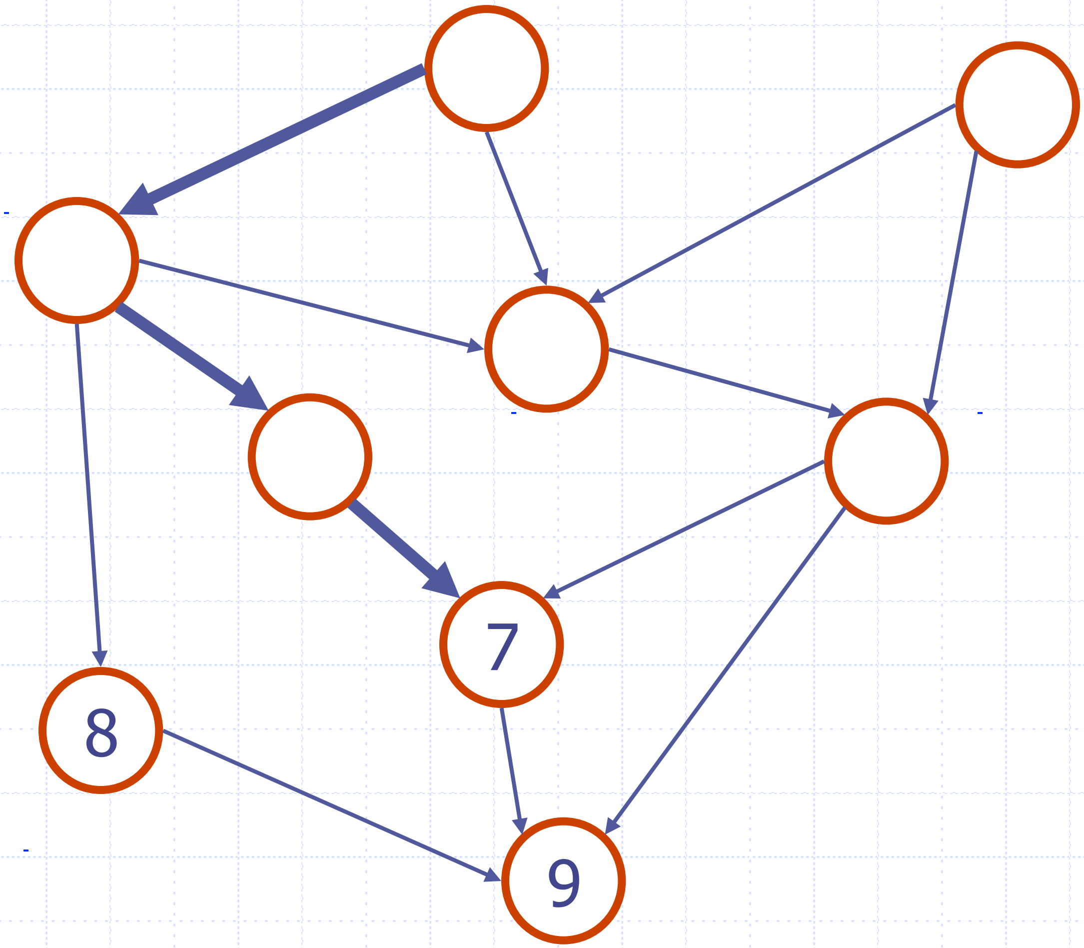
Topological Sorting Example



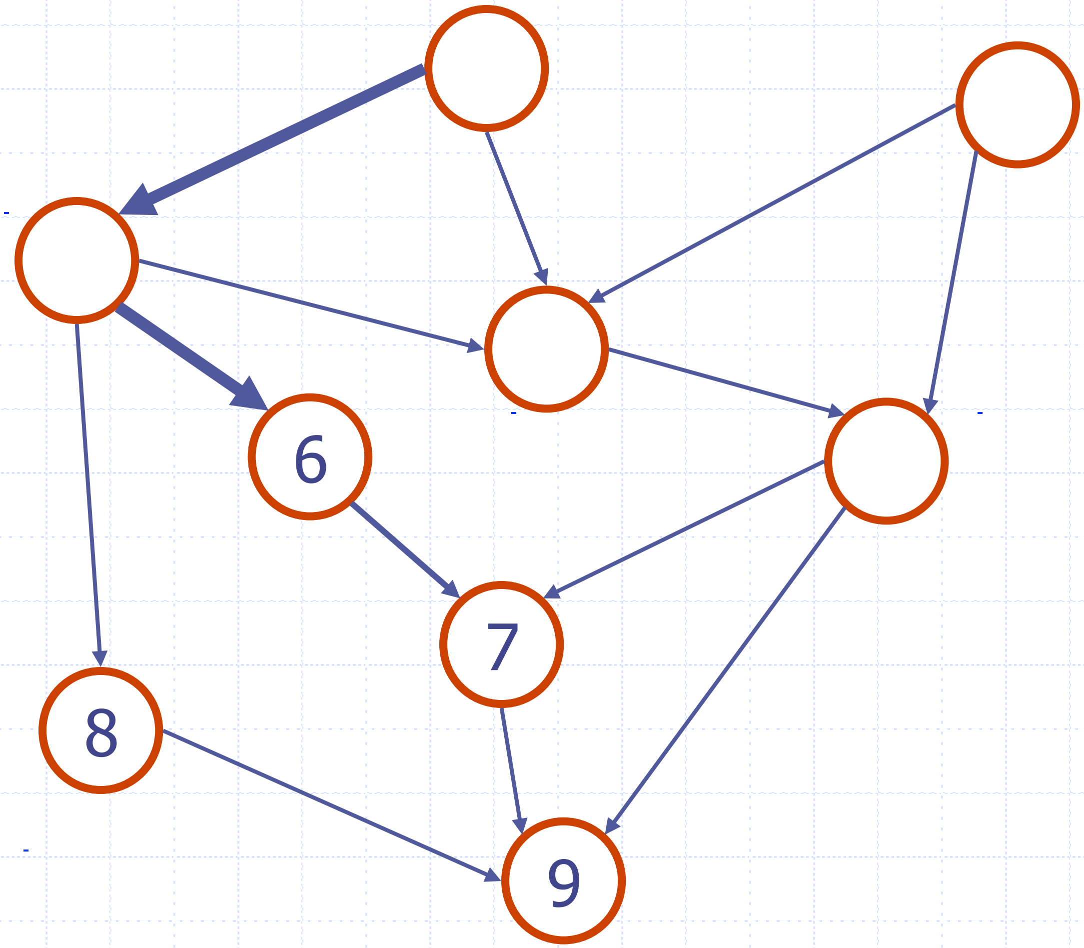
Topological Sorting Example



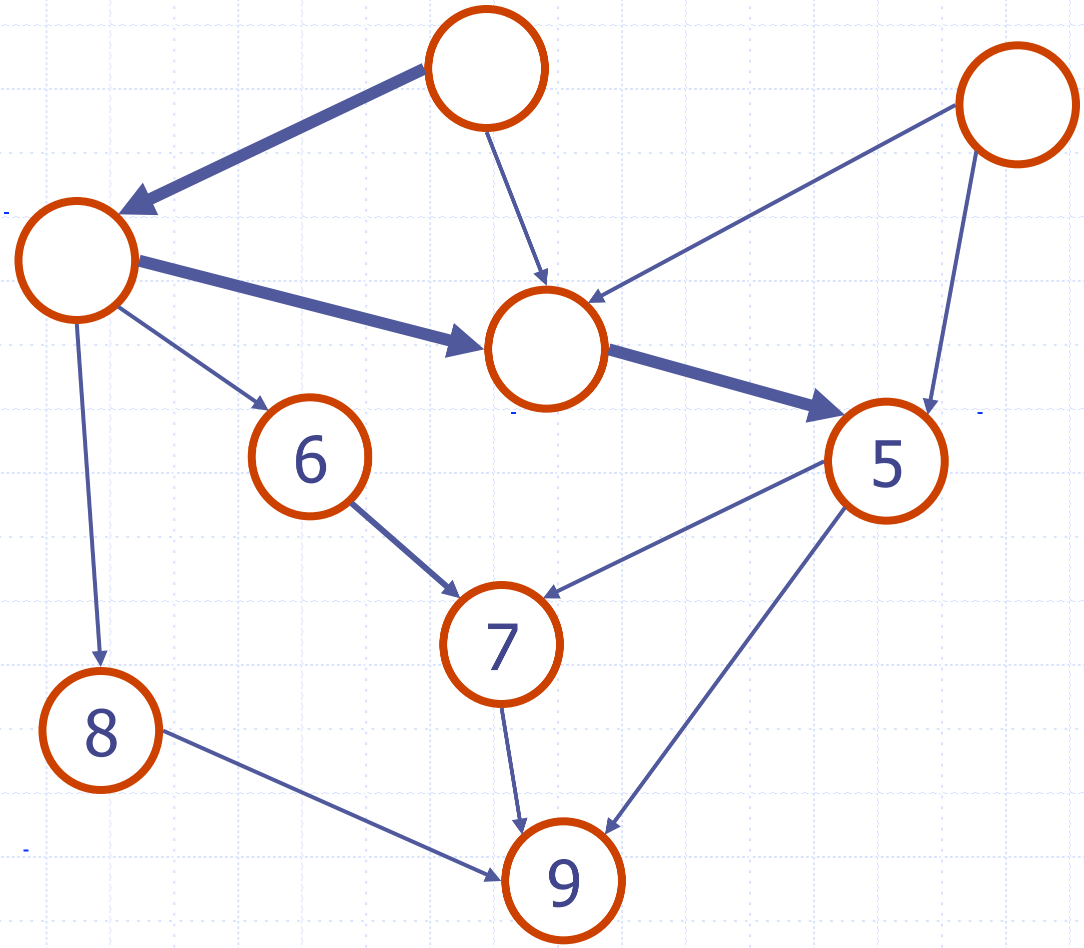
Topological Sorting Example



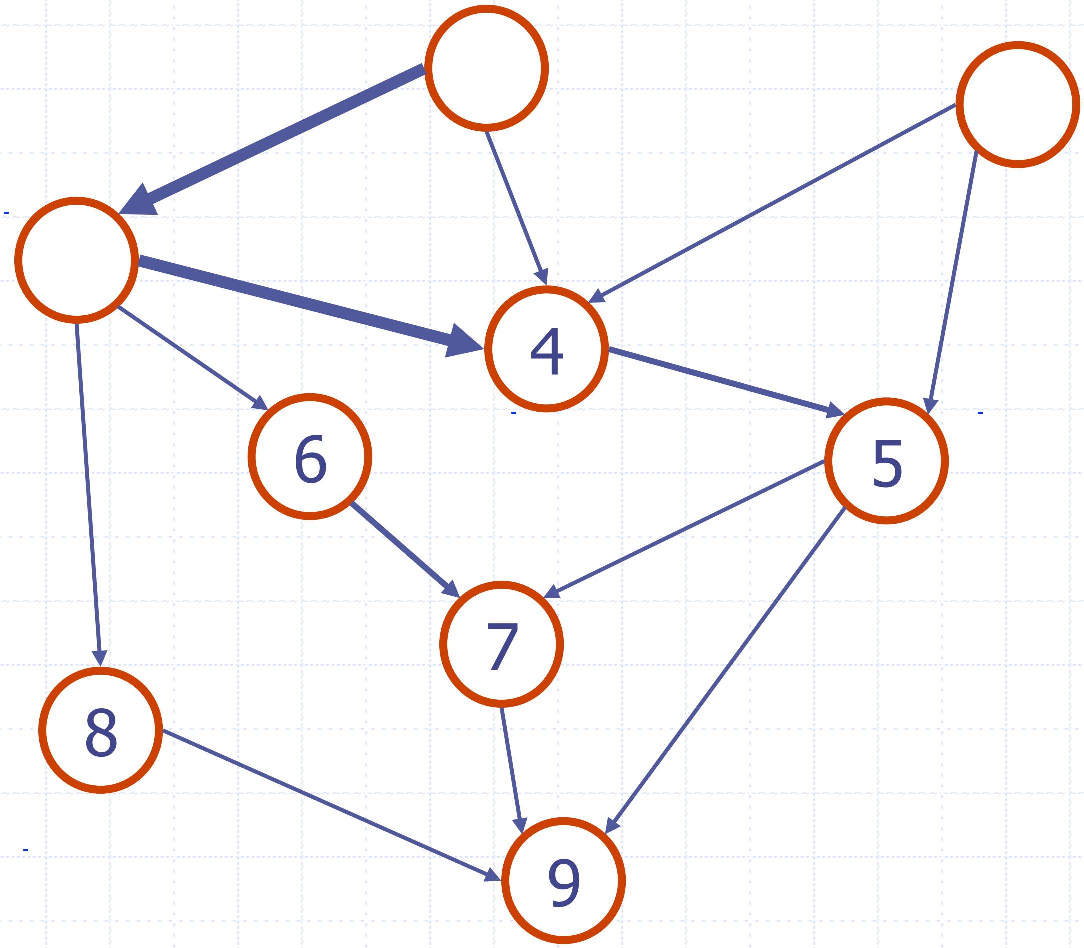
Topological Sorting Example



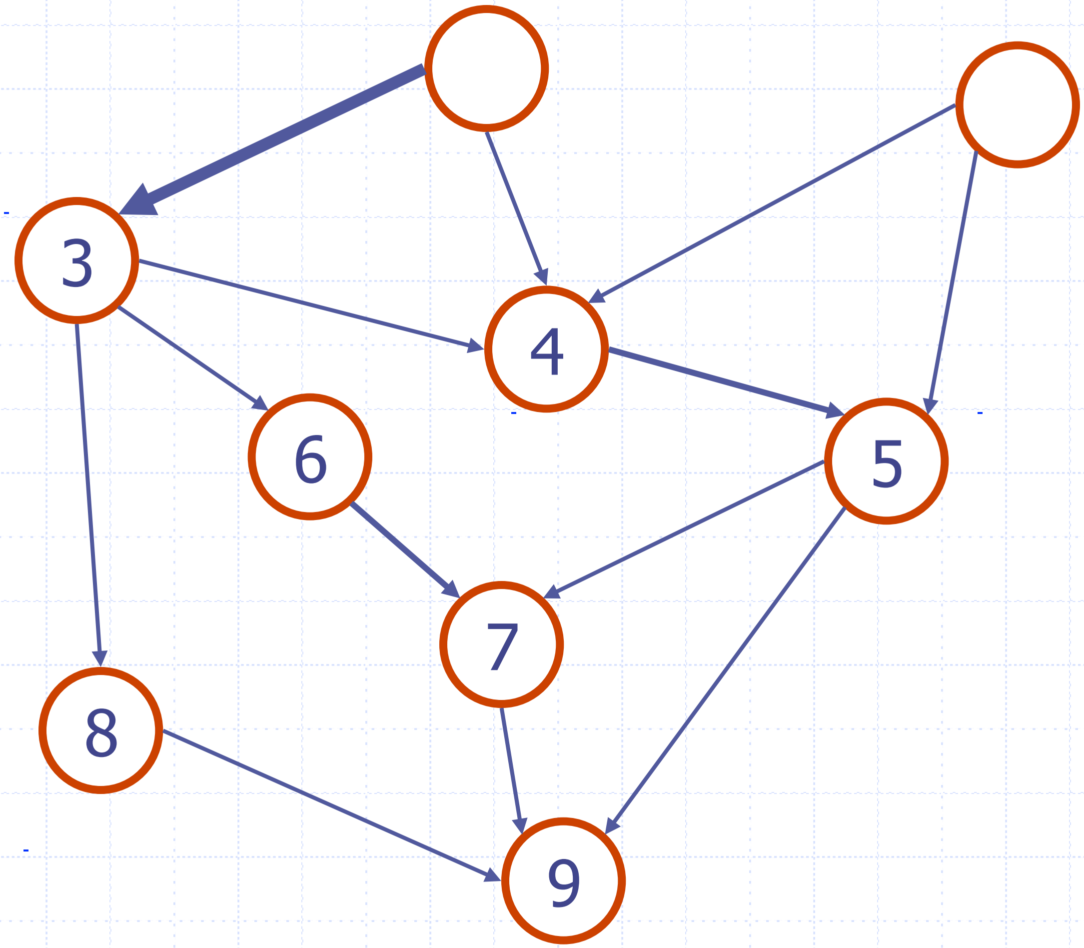
Topological Sorting Example



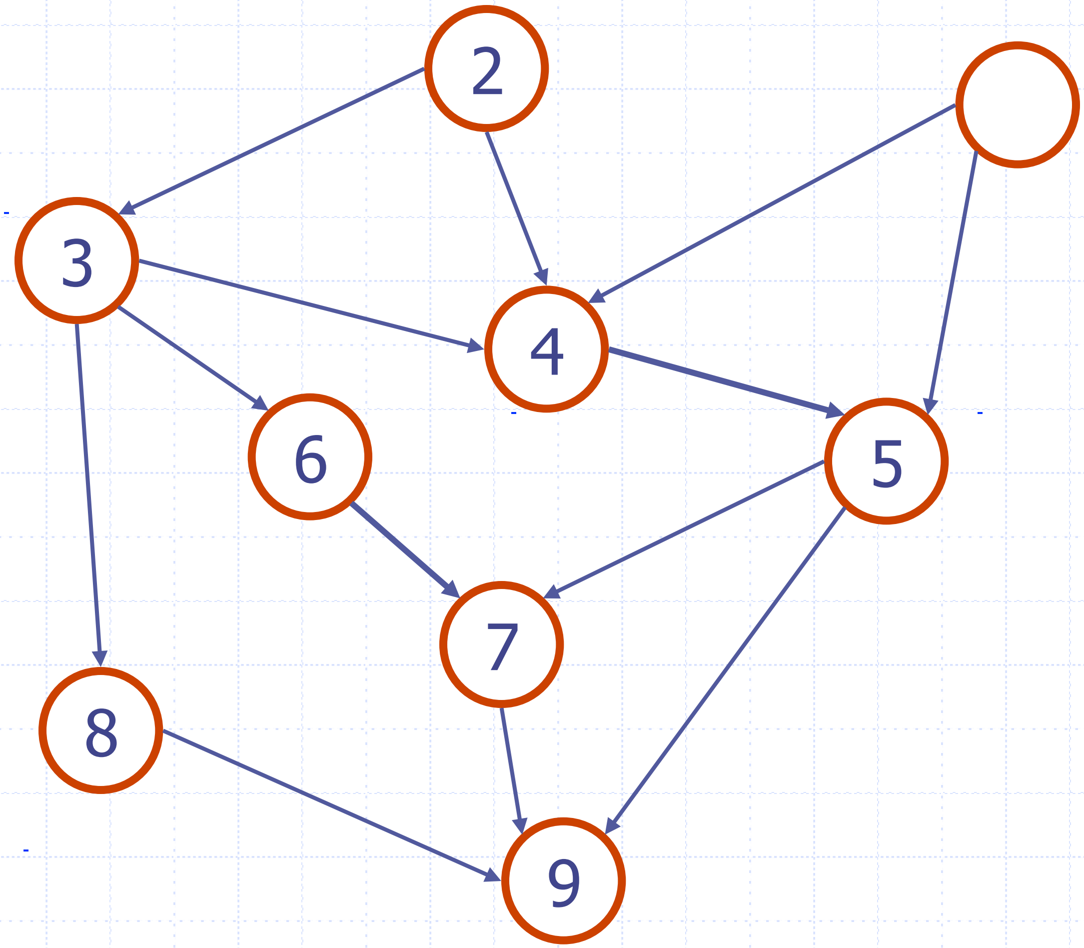
Topological Sorting Example



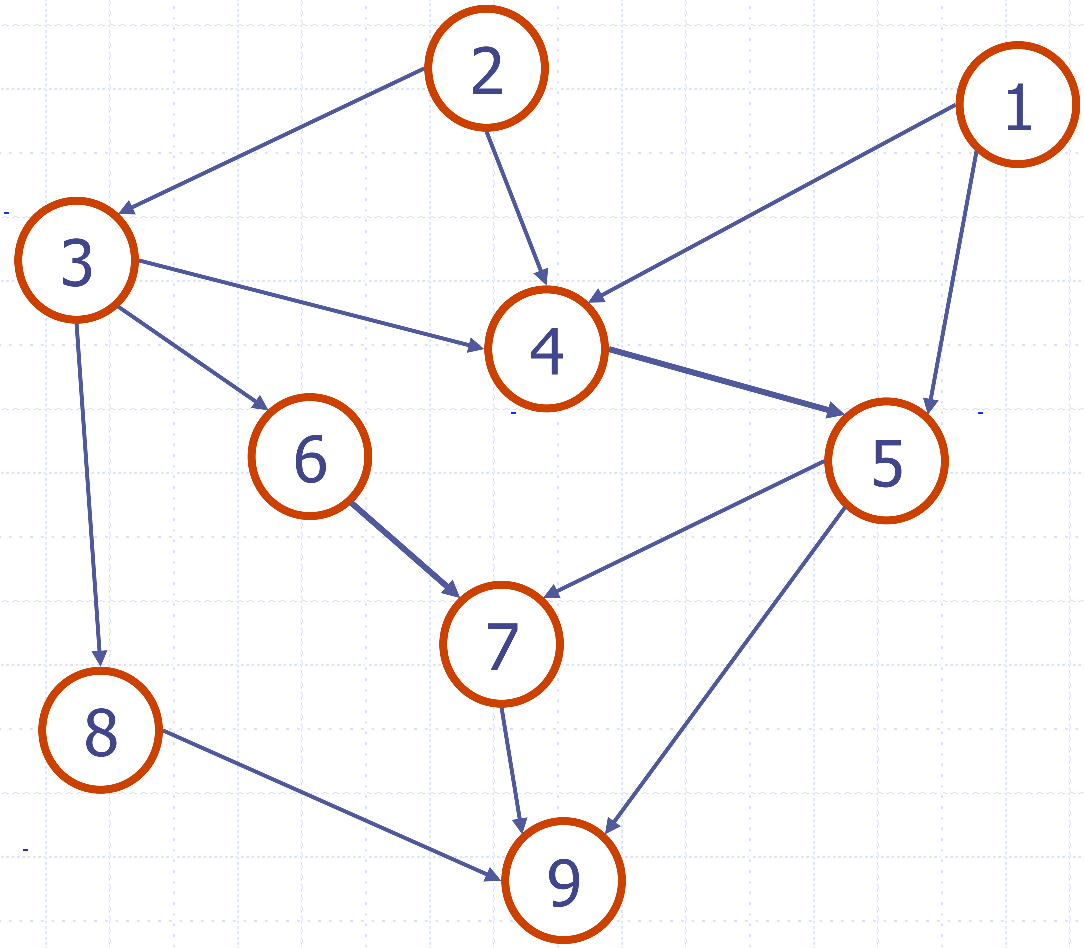
Topological Sorting Example



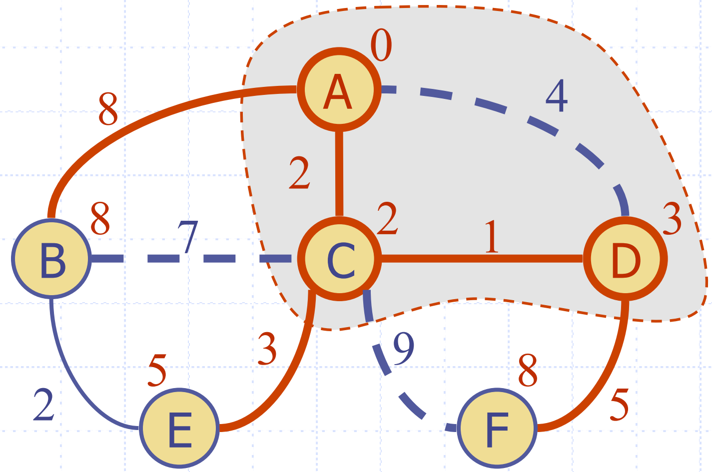
Topological Sorting Example



Topological Sorting Example

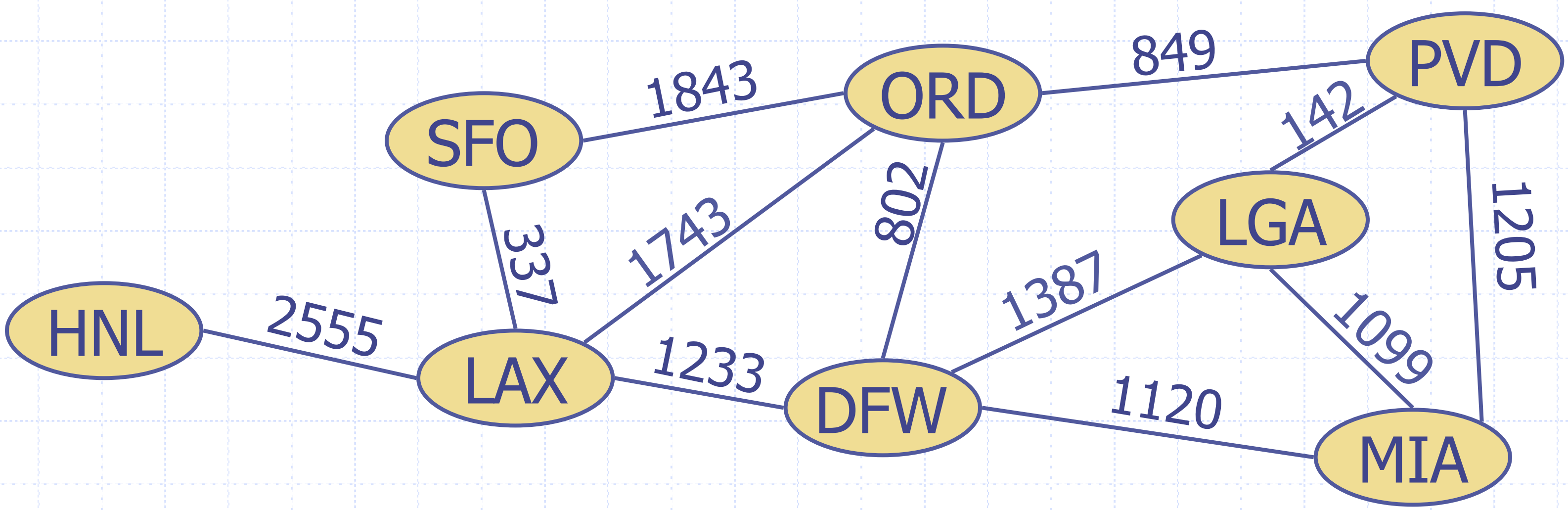


Shortest Paths



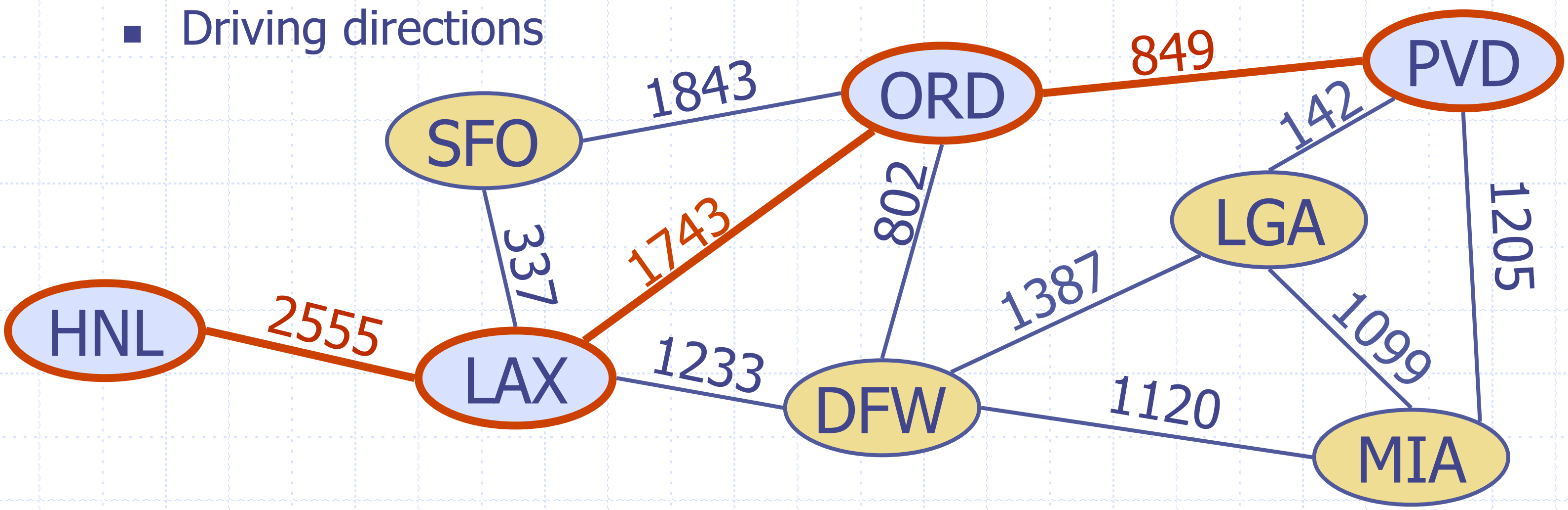
Weighted Graphs

- 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



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

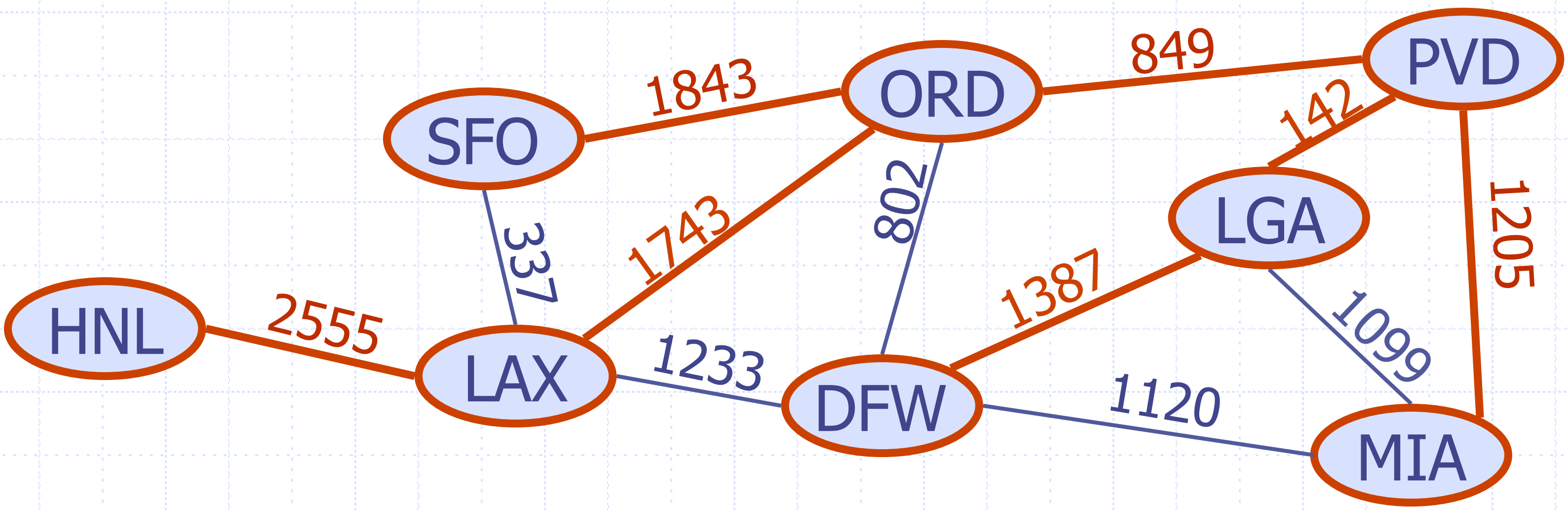


Shortest Path Properties

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



Unweighted SP: BFS Algorithm

```
Algorithm BFS(G, s)
  for all u ∈ G.vertices()  setLabel(u, UNEXPLORED)
  L ← new empty queue
  L.insertLast(s)
  setLabel(s, VISITED)
  setDist(s,0)
  i ← 1
  while ¬L.isEmpty()
     v = L.dequeue()
      for all w ∈ G.Adjacent(v)
          if getLabel(w) = UNEXPLORED
              setLabel(w, VISITED)
              setDist(w, i)
              setPath(w, v)
              L.insertLast(w)
          end
      end
      i ← i +1
  end
```

Dijkstra's Algorithm

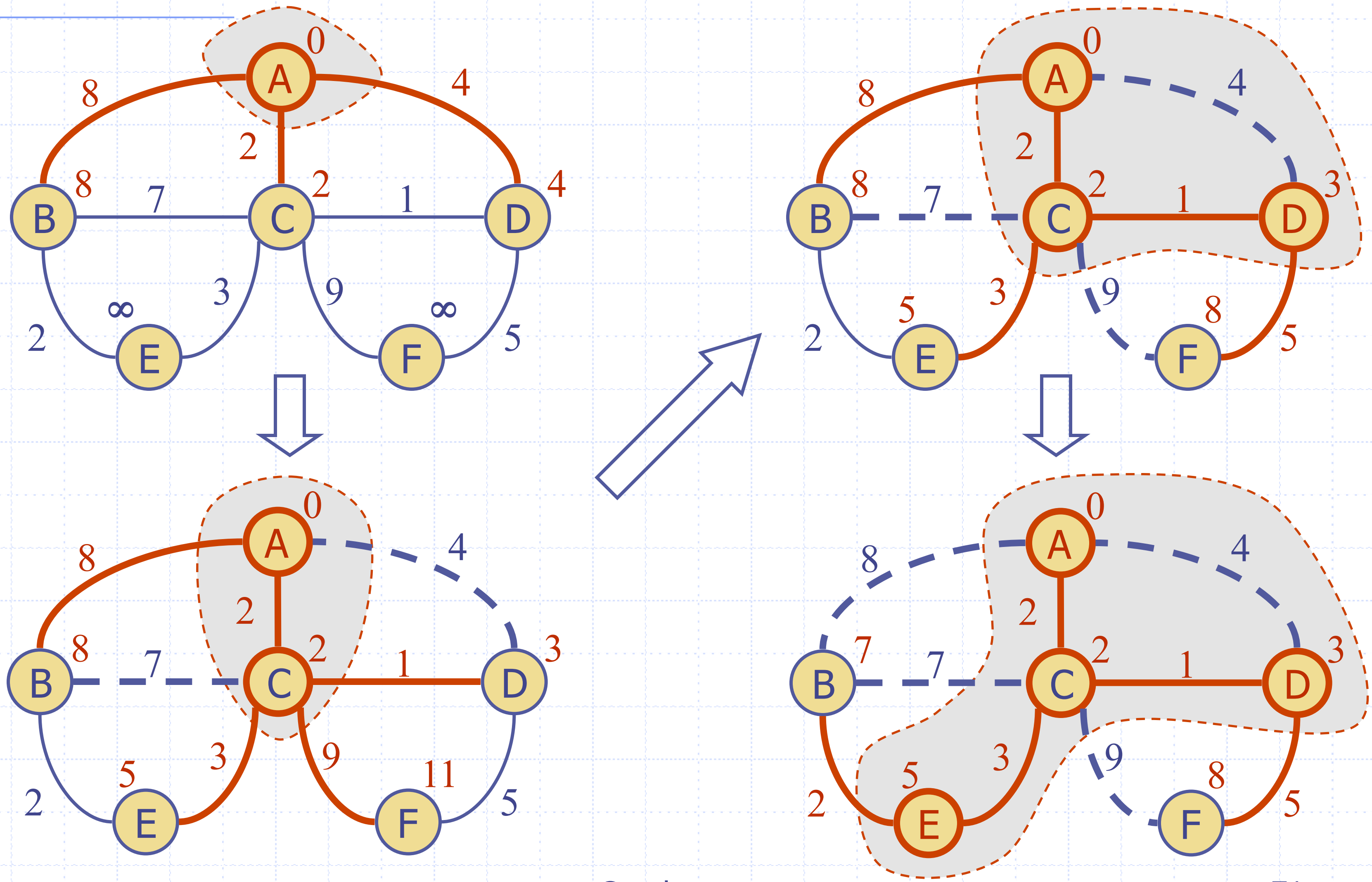
- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We 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

Dijkstra's Algorithm

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(*k*,*e*) returns a locator
  - *replaceKey*(*l*,*k*) changes the key of an item
- We store two labels with each vertex:
  - Distance (d(v) label)
  - locator in priority queue

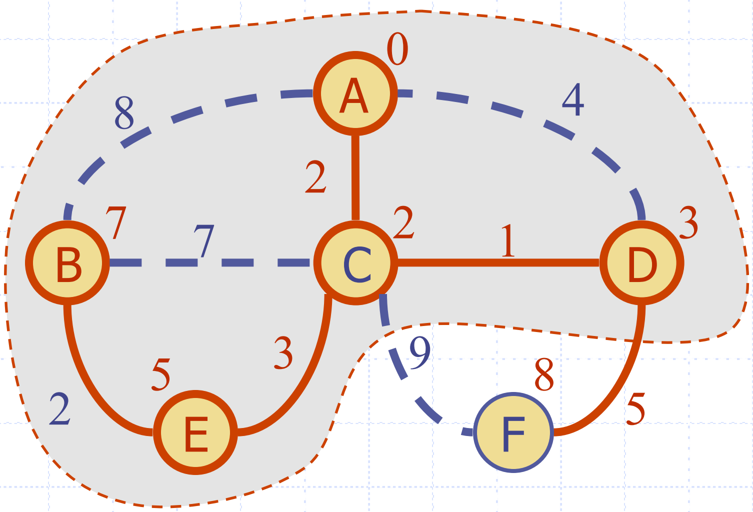
Algorithm *DijkstraDistances*(*G, s*)
  $Q \leftarrow$ new heap-based priority queue
  for all $v \in G.vertices()$
    if $v = s$
      $setDistance(v, 0)$
    else
      $setDistance(v, \infty)$
    $l \leftarrow Q.insert(getDistance(v), v)$
    $setLocator(v,l)$
  while $\neg Q.isEmpty()$
    $u \leftarrow Q.removeMin()$
    for all $e \in G.incidentEdges(u)$
      { relax edge *e* }
      $z \leftarrow G.opposite(u,e)$
      $r \leftarrow getDistance(u) + weight(e)$
      if $r < getDistance(z)$
        $setDistance(z,r)$
        $Q.replaceKey(getLocator(z),r)$

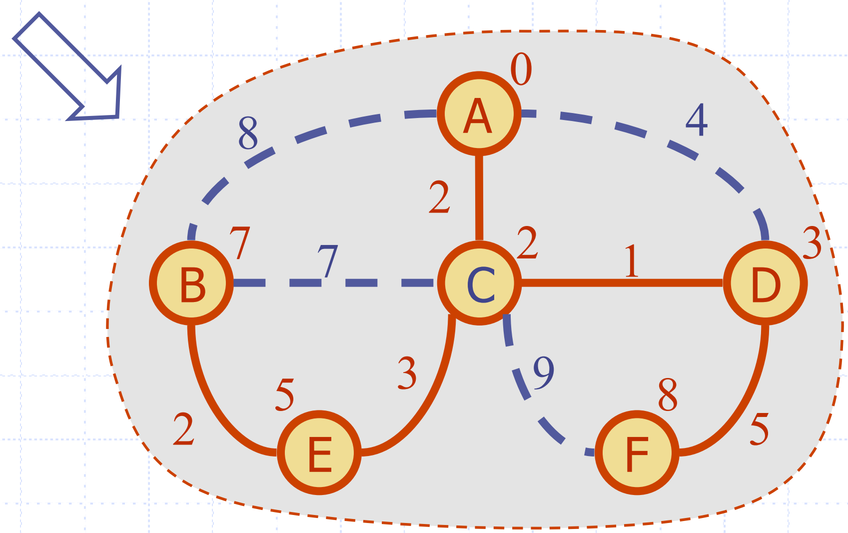
Example



A B C D E F
0 8 4 2 8 7 2 1 4 2 ∞ 3 9 ∞ 5
A B C D E F
0 8 4 2 8 7 2 1 3 2 5 3 9 11 5
A B C D E F
0 8 4 2 8 7 2 1 3 2 5 3 9 8 5
A B C D E F
0 8 4 2 7 7 2 1 3 2 5 3 9 8 5

Example (cont.)





Analysis of Dijkstra's Algorithm

- Graph operations
  - Method incidentEdges is called 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 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

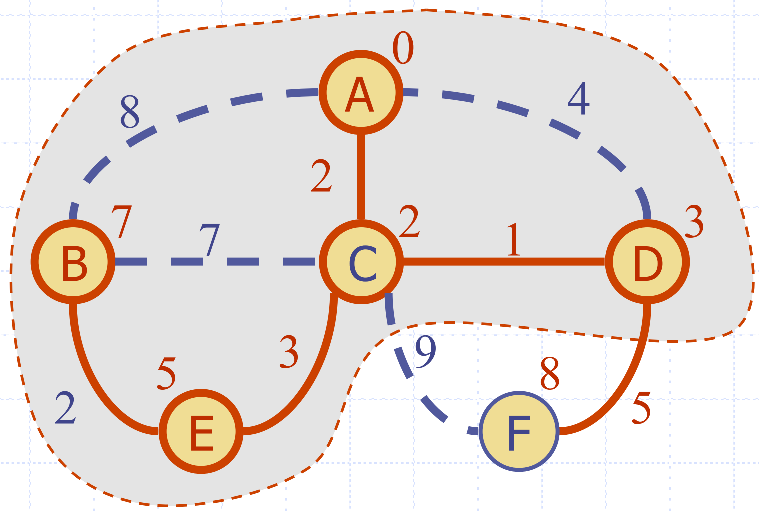
Shortest Paths Tree

- Using the template method pattern, we can extend Dijkstra's algorithm to return a tree of shortest paths from the start vertex to all other vertices
- We store with each vertex a third label:
  - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label

```
Algorithm DijkstraShortestPathsTree(G, s)

    …

  for all v ∈ G.vertices()
    …
    setParent(v, ∅)
    …

    for all e ∈ G.incidentEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(getLocator(z),r)
```

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)$\geq$d(D), F's distance cannot be wrong. That is, there is no wrong vertex.



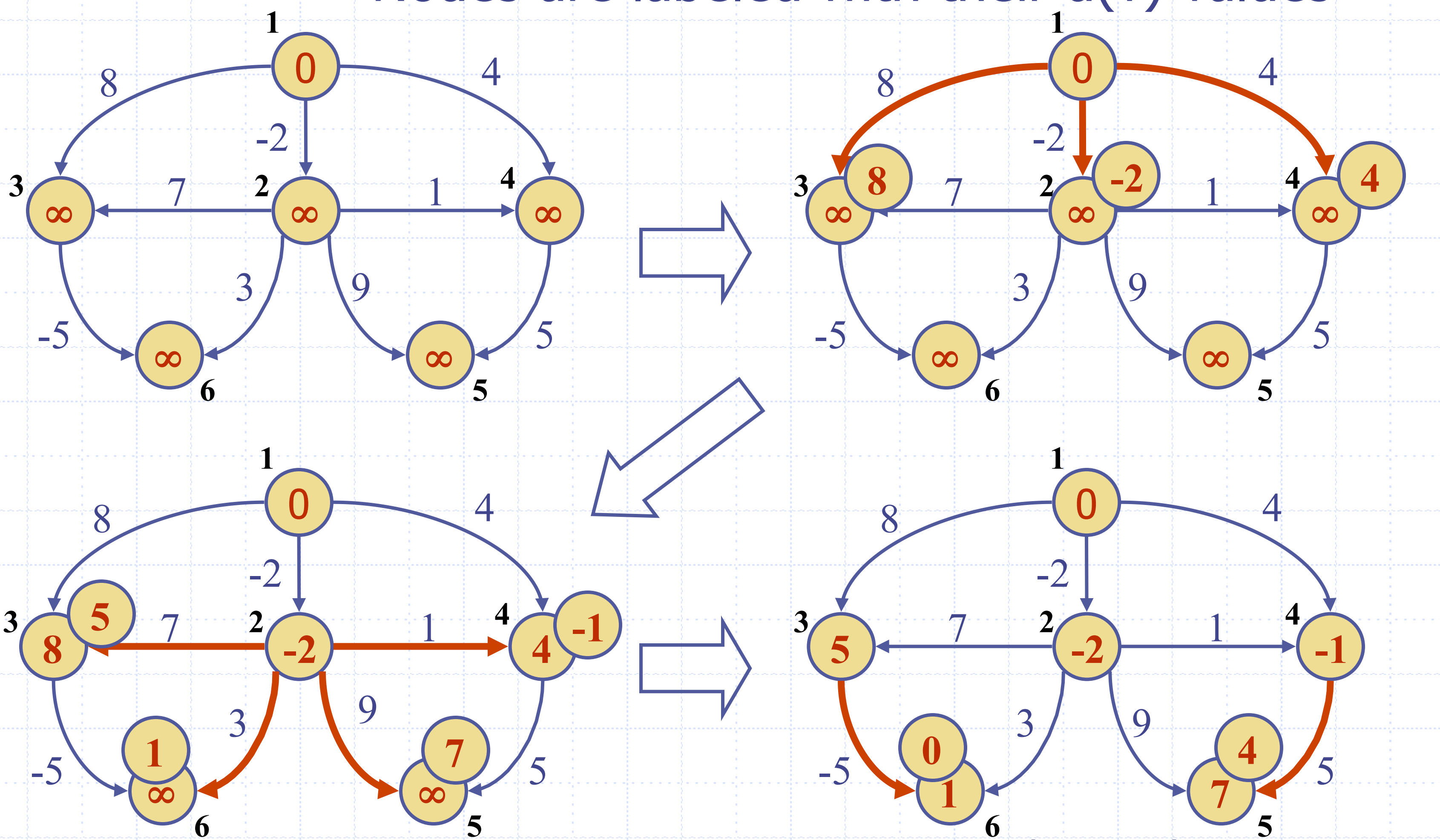
DAG-based Algorithm

- 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 $v \in G.vertices()$
        if $v = s$
            setDistance$(v, 0)$
        else
            setDistance(v, ∞)
    Perform a topological sort of the vertices
    for $u \leftarrow 1$ **to** n **do**   {in topological order}
        for each $e \in G.outEdges(u)$
            { relax edge e }
            $z \leftarrow G.opposite(u,e)$
            $r \leftarrow getDistance(u) + weight(e)$
            if $r < getDistance(z)$
                setDistance(z,r)

DAG Example

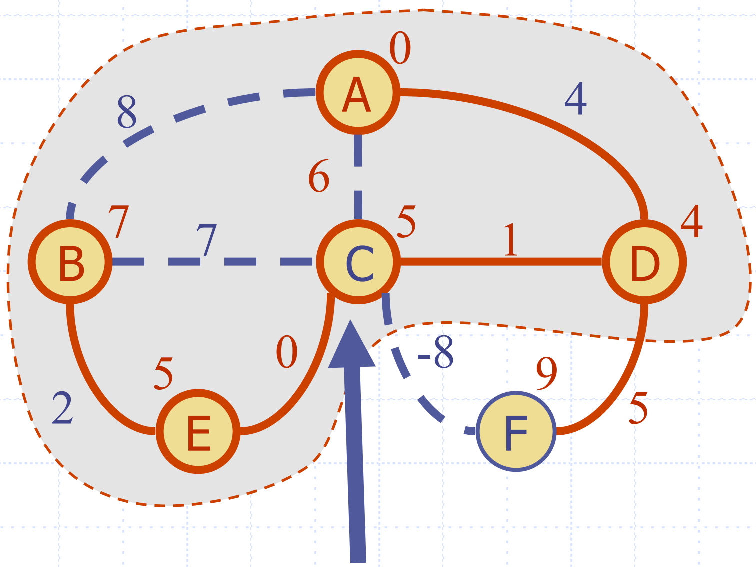
Nodes are labeled with their d(v) values



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!

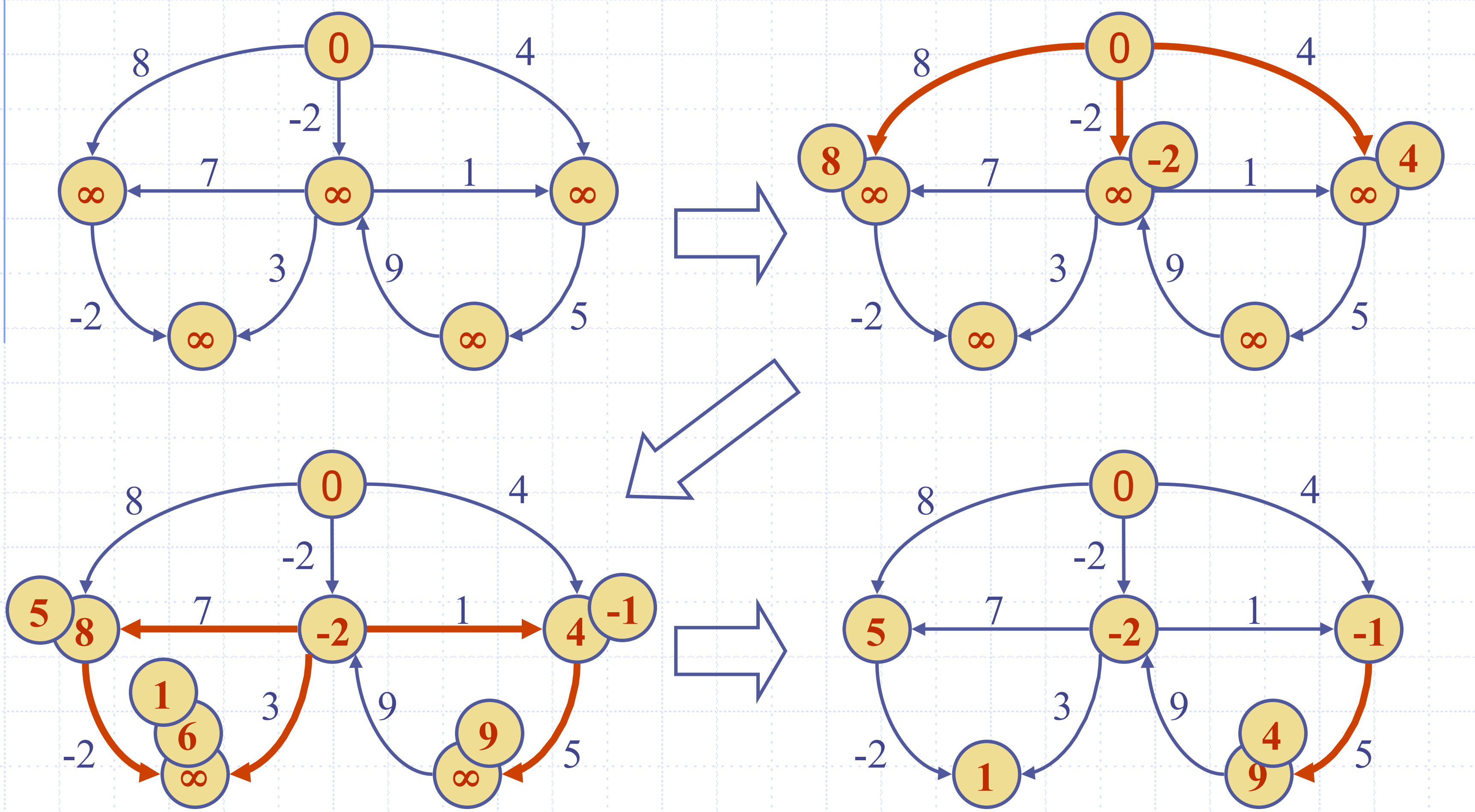
Bellman-Ford Algorithm

- 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).

```
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)
```

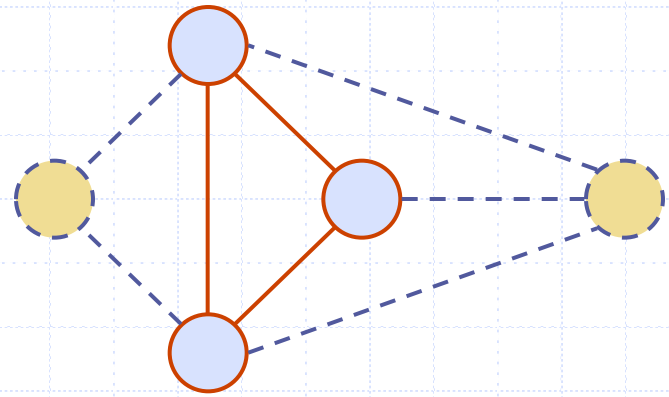
Bellman-Ford Example

Nodes are labeled with their d(v) values

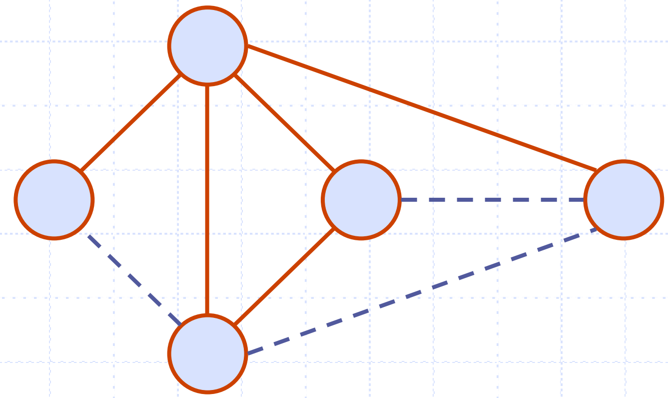


Subgraphs

- 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



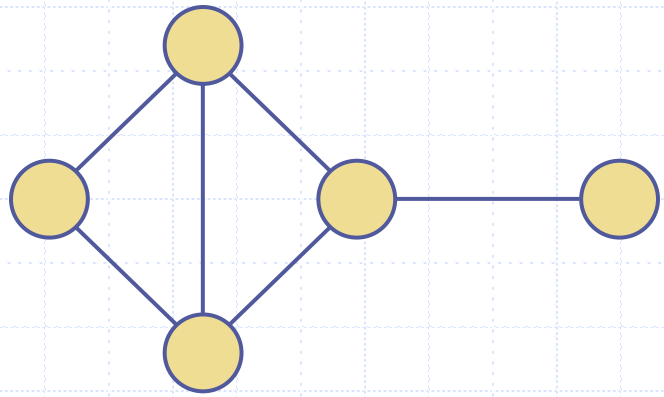
Subgraph



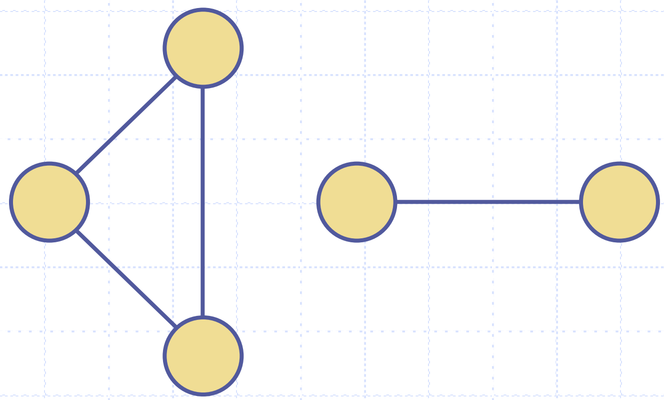
Spanning subgraph

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

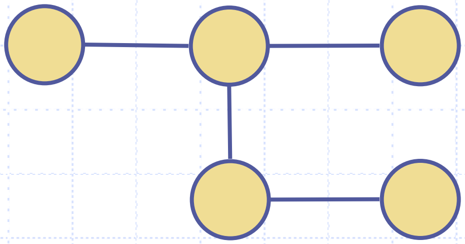


Non connected graph with two connected components

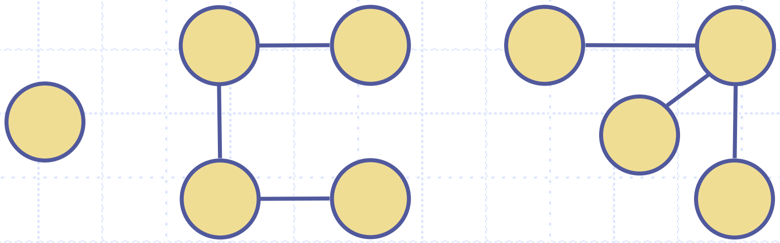
Trees and Forests

- 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



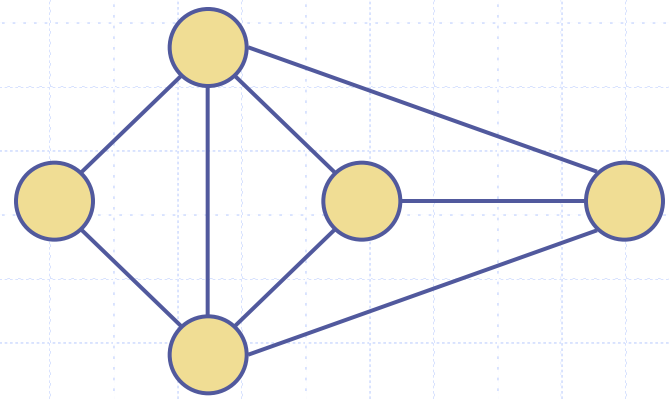
Tree



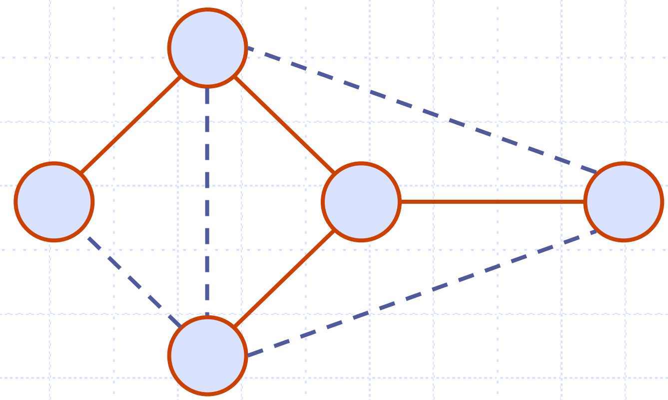
Forest

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

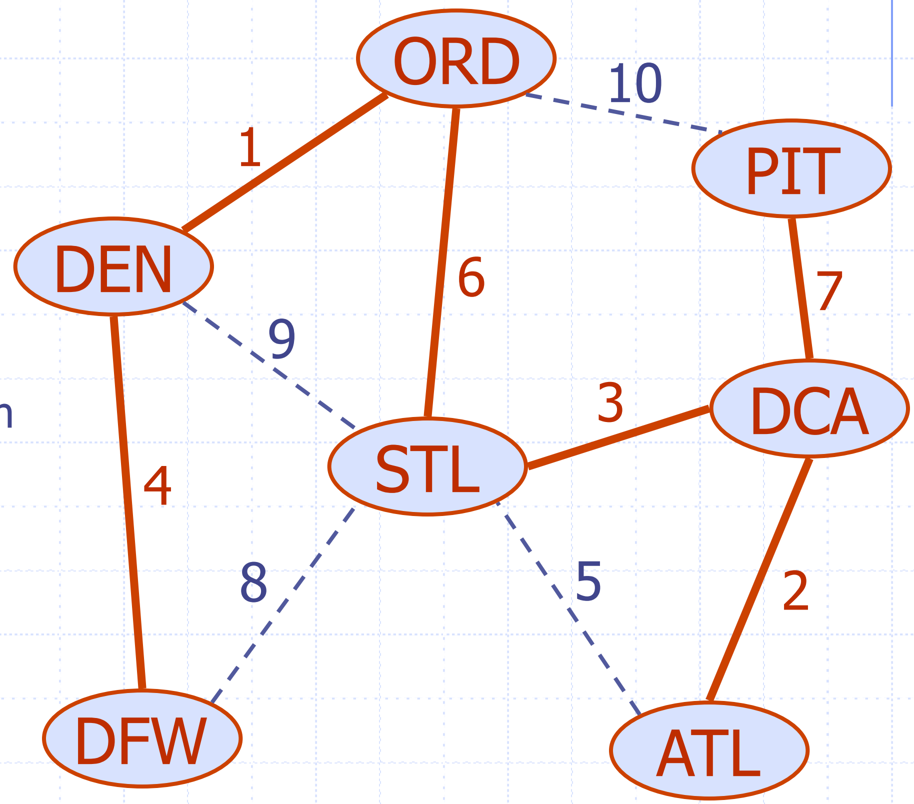
Minimum Spanning Trees

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

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

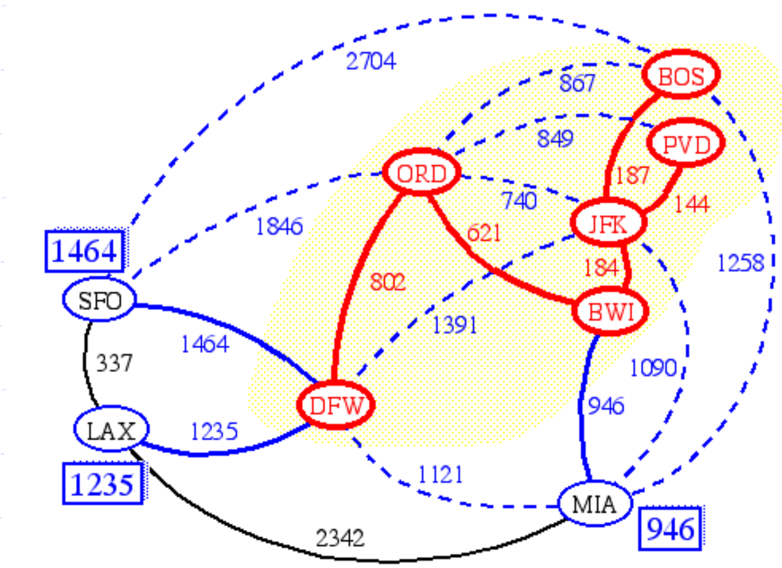
Minimum spanning tree (MST)
- Spanning tree of a weighted graph with minimum total edge weight

- Applications
  - Communications networks
  - Transportation networks



Prim-Jarnik's Algorithm

- Similar to Dijkstra's algorithm (for a connected graph)
- 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 a label $d(v)$ = 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

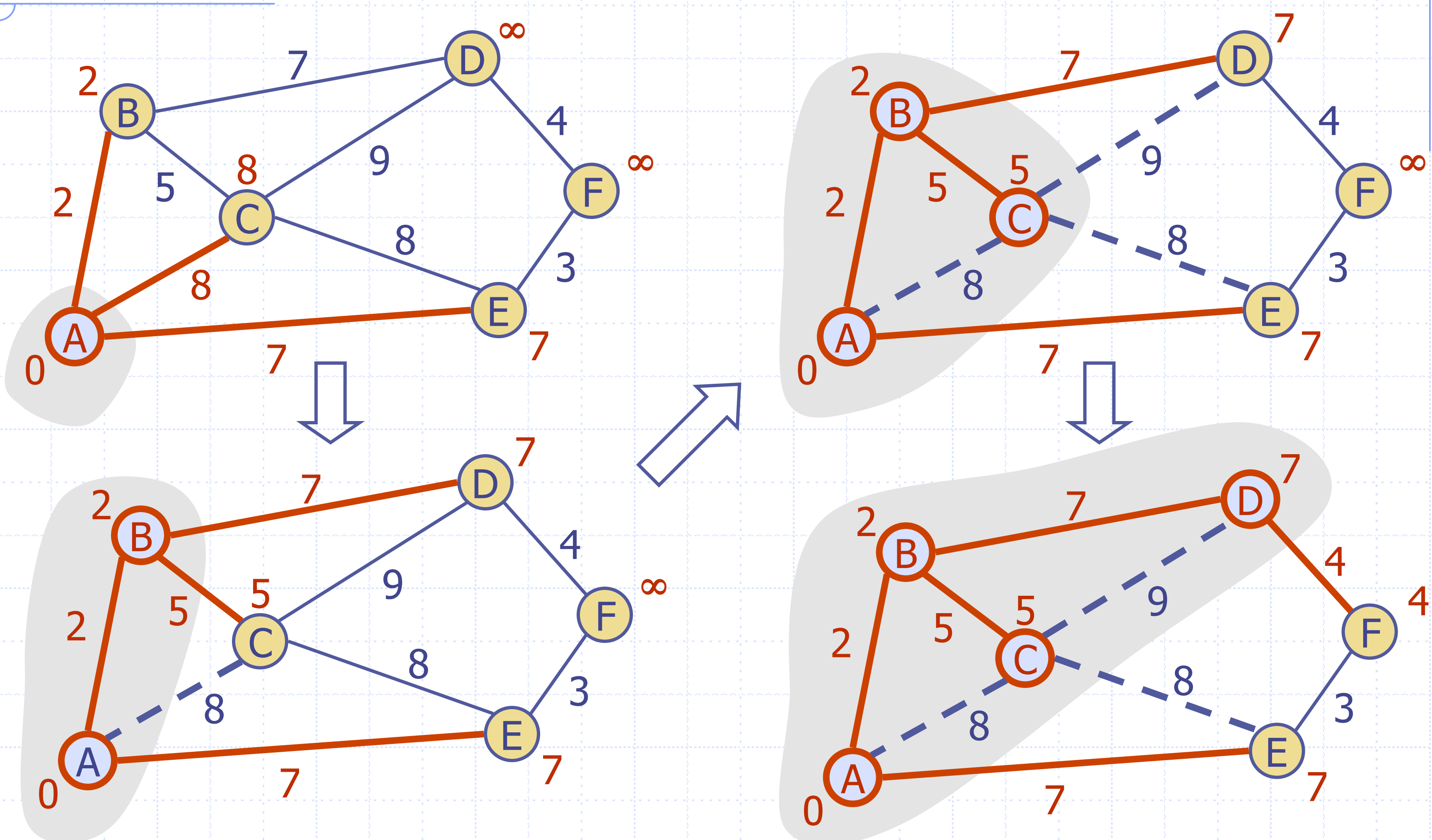


Prim-Jarnik's Algorithm (cont.)

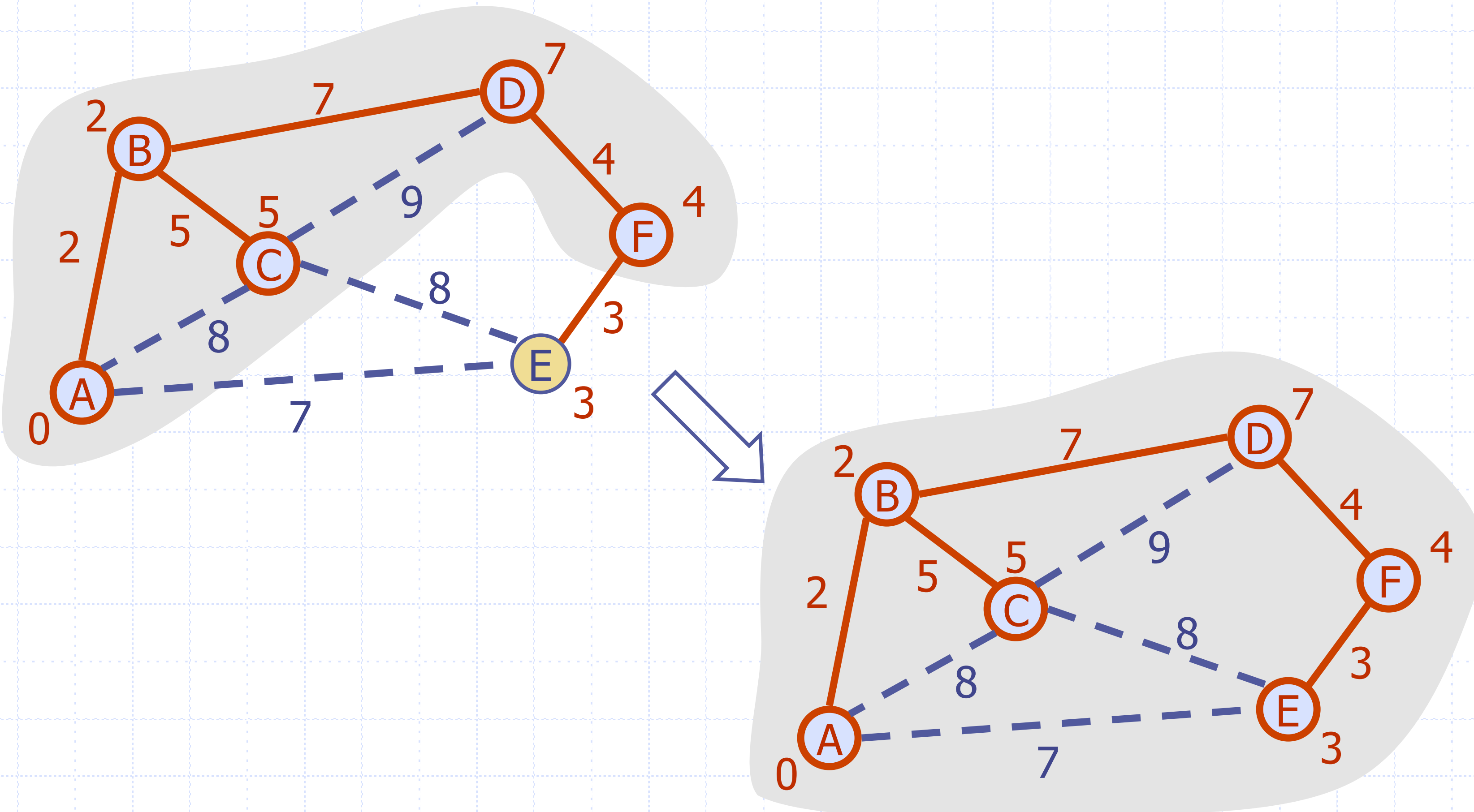
- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(*k*,*e*) returns a locator
  - *replaceKey*(*l*,*k*) changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator in priority queue

```
Algorithm PrimJarnikMST(G)
  Q ← new heap-based priority queue
  s ← a vertex of G
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    setParent(v, ∅)
    l ← Q.insert(getDistance(v), v)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
      z ← G.opposite(u,e)
      r ← weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(z,r)
```

Example



Example (contd.)



Analysis

- Graph operations
  - Method incidentEdges is called 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

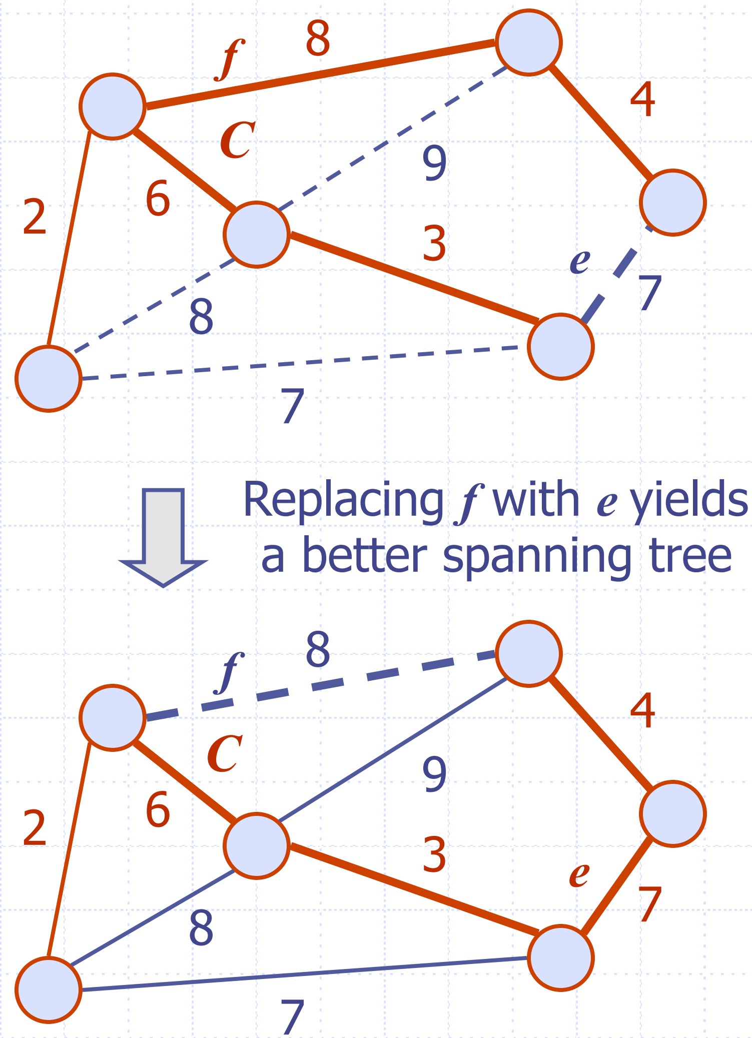
A 2nd Idea: Cycle Property

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



Partition Property

Partition Property:

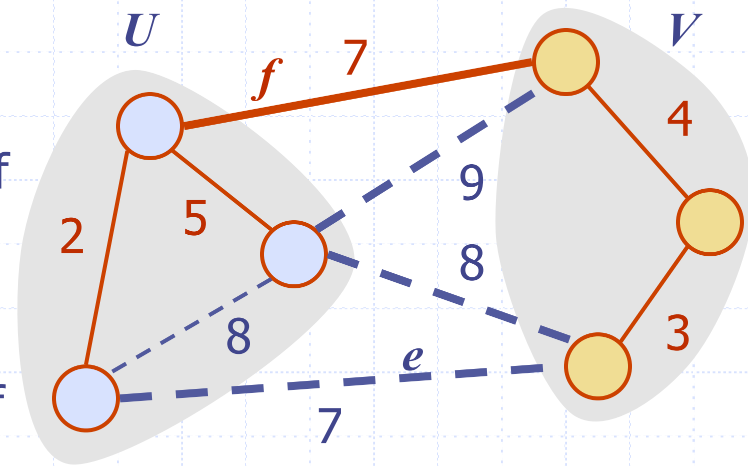
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

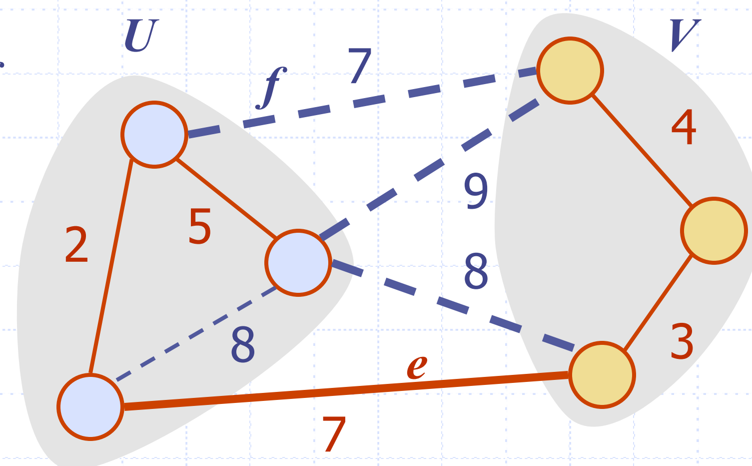
- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property,

$$weight(f) \leq weight(e)$$

- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e



Replacing f with e yields another MST



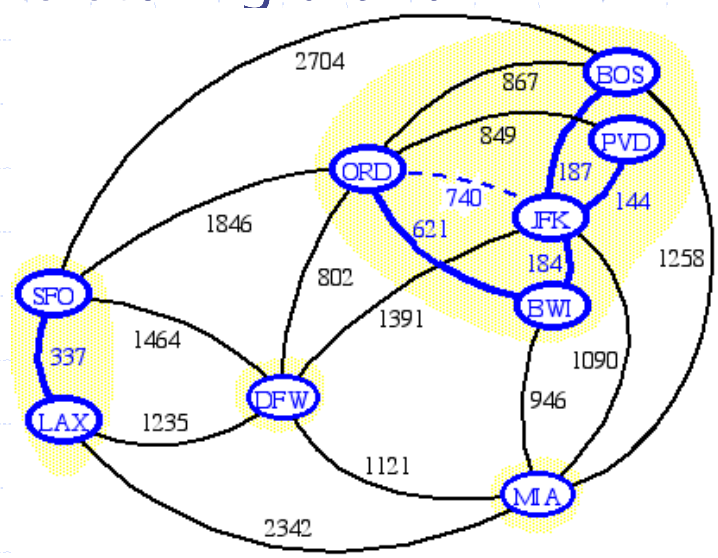
Kruskal's Algorithm

- A priority queue stores the edges outside the cloud
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - We are left with one cloud that encompasses the MST
  - A tree *T* which is our MST

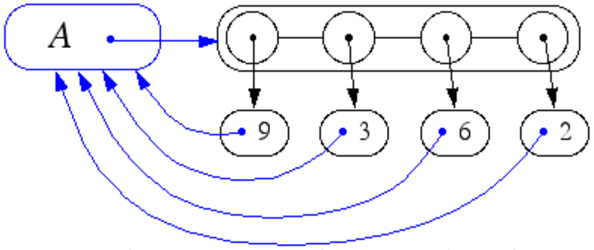
Algorithm *KruskalMST*(*G*)
  for each vertex *V* in *G* **do**
    define a ***Cloud(v)*** of $\leftarrow$ {*v*}
  let ***Q*** be a priority queue.
  Insert all edges into ***Q*** using their weights as the key
  T $\leftarrow \varnothing$
  while ***T*** has fewer than ***n***-1 edges **do**
      edge ***e*** = ***T.removeMin()***
    Let ***u***, ***v*** be the endpoints of *e*
    if ***Cloud(v)*** $\neq$ ***Cloud(u)*** **then**
      Add edge ***e*** to ***T***
      Merge ***Cloud(v)*** and ***Cloud(u)***
  return ***T***

Data Structure for Kruskal Algorithm

- The algorithm maintains a forest of trees
- 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 the operations:
  - -**find**(u): return the set storing u
  - -**union**(u,v): replace the sets storing u and v with their union



Representation of a 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(u,v), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation union(u,v) is $\min(n_u, n_v)$, where n_u and n_v are the sizes of the sets storing u and v
- 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

Partition-Based Implementation

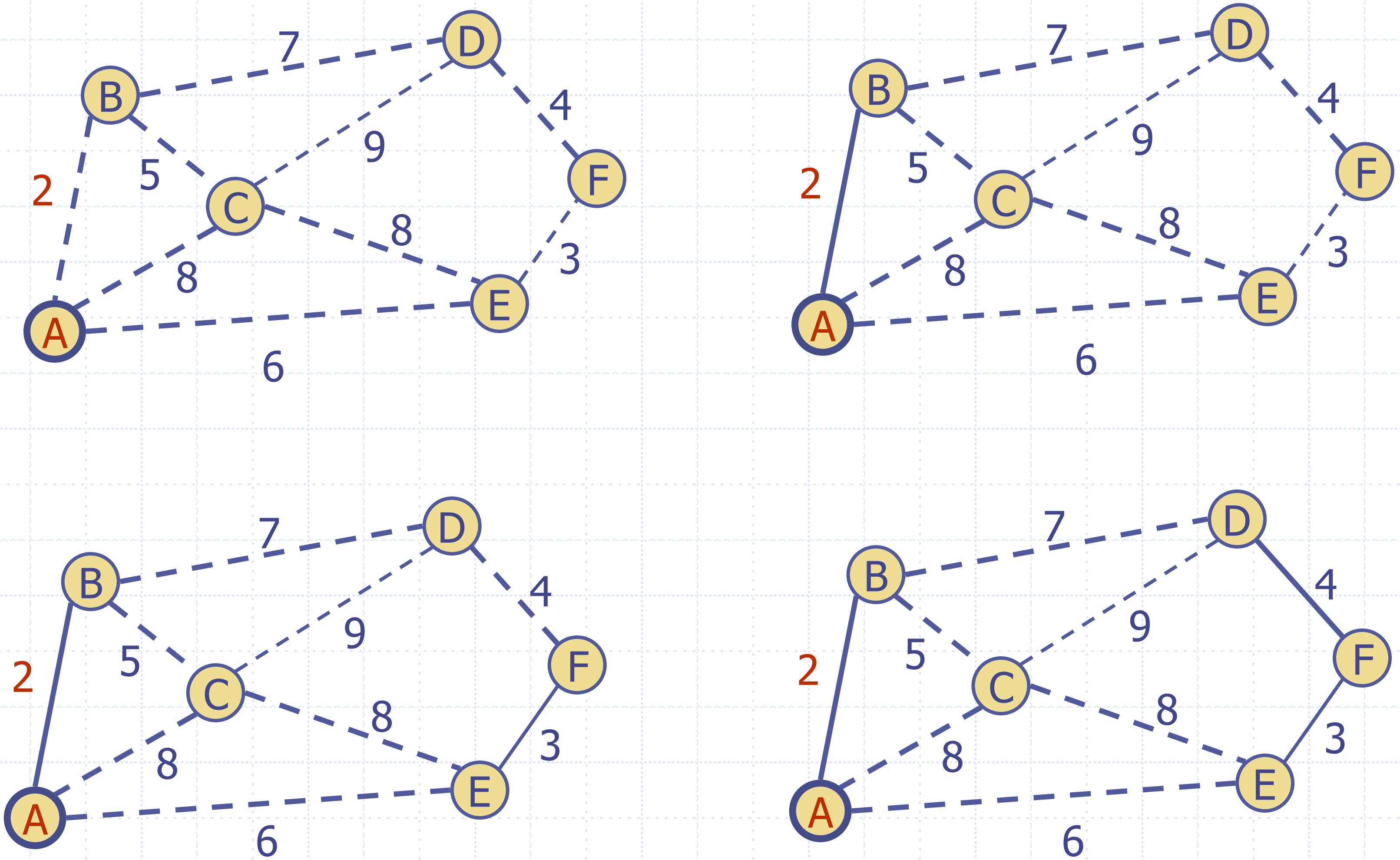
- A partition-based version of Kruskal's Algorithm performs cloud merges as unions and tests as finds.

```
Algorithm Kruskal(G):
  Input: A weighted graph G.
  Output: An MST T for G.
Let P be a partition of the vertices of G, where each vertex forms a separate set.
Let Q be a priority queue storing the edges of G, sorted by their weights
Let T be an initially-empty tree
while Q is not empty do
   (u,v) ← Q.removeMinElement()
   if P.find(u) != P.find(v) then
            Add (u,v) to T
            P.union(u,v)
return T
```

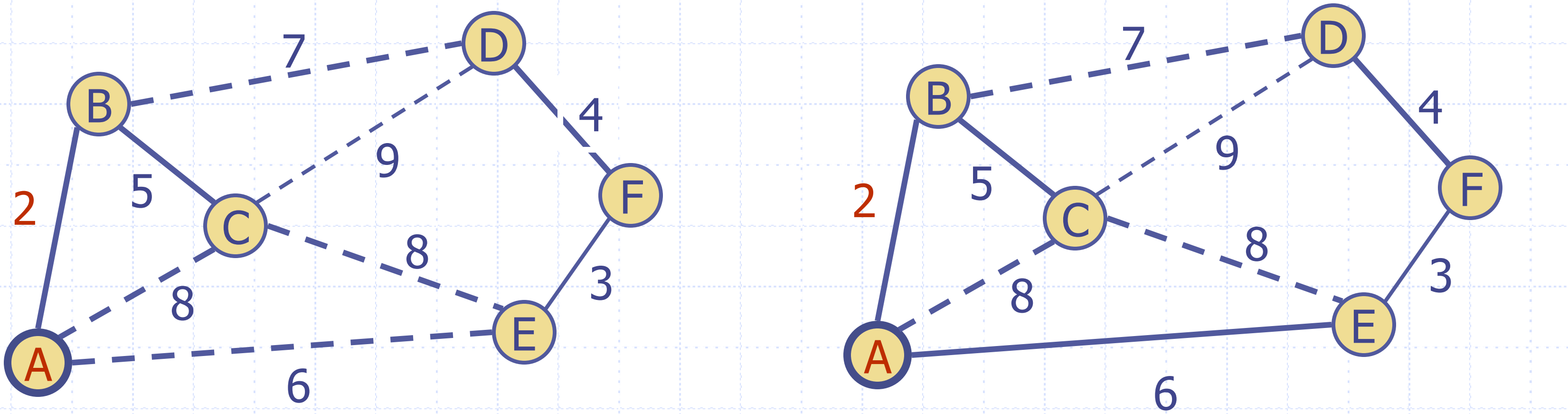
Running time: $O(m \log n)$

or $O(m \log^* n)$ with path compression

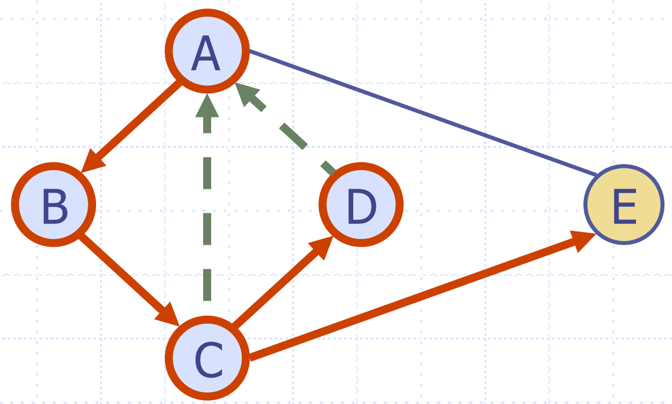
Example


A
B
C
D
E
F
2
5
7
9
4
8
8
3
6
A
B
C
D
E
F
2
5
7
9
4
8
8
3
6
A
B
C
D
E
F
2
5
7
9
4
8
8
3
6
A
B
C
D
E
F
2
5
7
9
4
8
8
3
6

Example

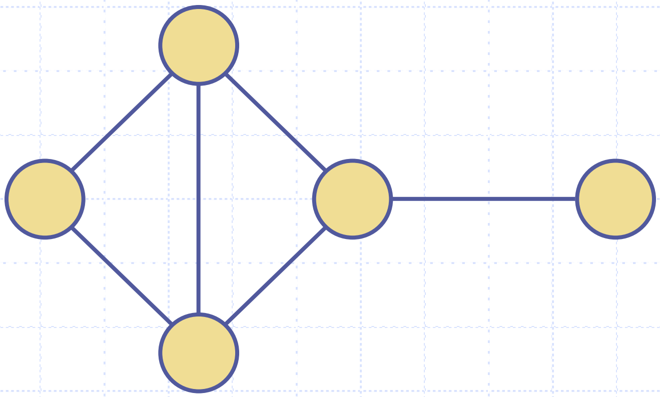


Depth-First Search

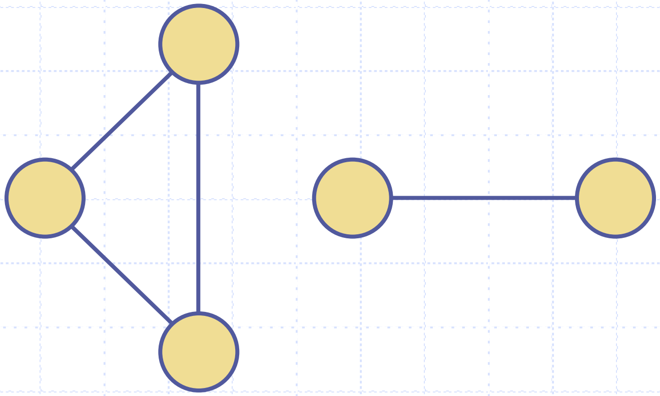


Connectivity

- 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

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

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)
```

Example

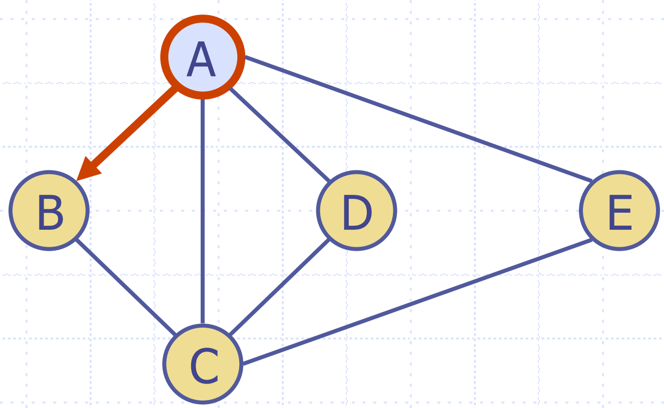
unexplored vertex

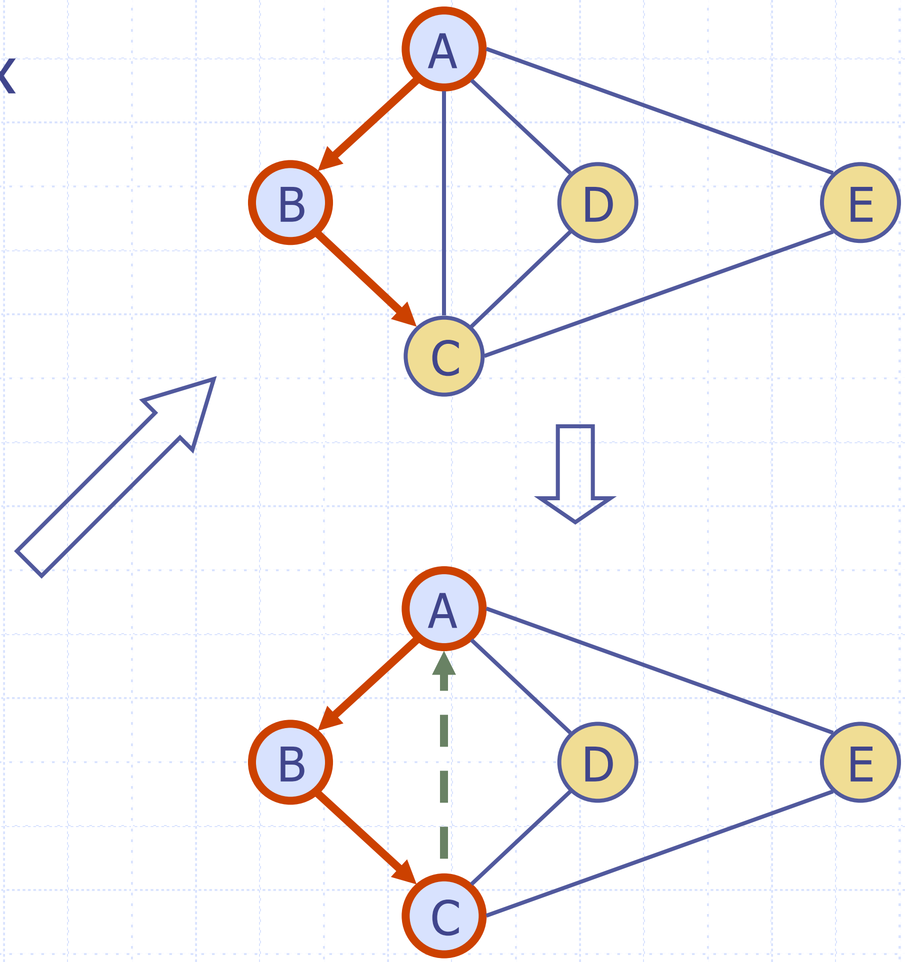
visited vertex

unexplored edge

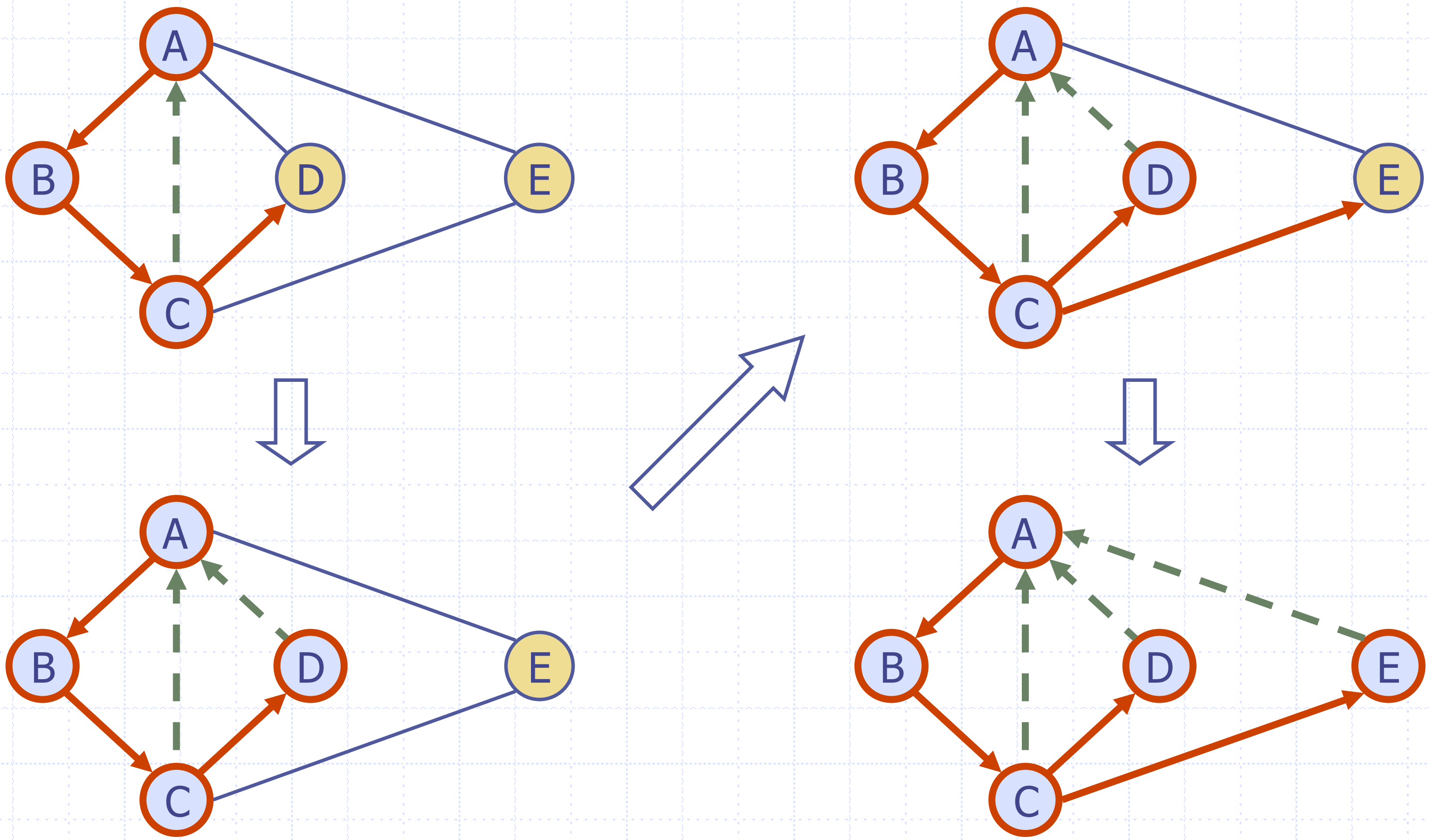
discovery edge

back edge





Example (cont.)



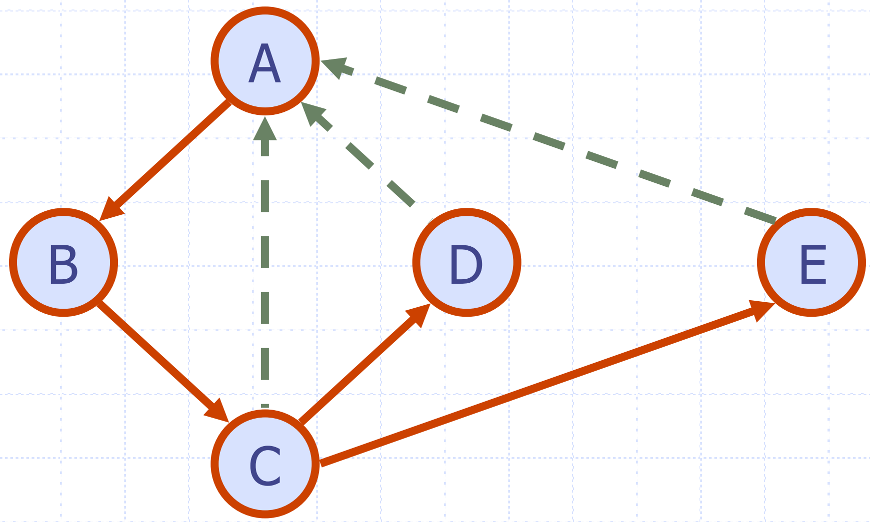
Properties of DFS

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*



DFS Analysis

- Each edge or vertex initialized: O(n+m)
- Each edge or vertex marked once O(n+m)
- Each edge visited twice (once for each vertex): O(m)
- Each vertex v visited ind(v) times: O(m)
- Assumes opposite is constant time
- 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)
        x = pathDFS(G, w, z)
        if (not x=null)
          return x
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
  return null
```

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

Finding Articulation Points

- An articulation point is a vertex such that removing the vertex would disconnect the graph

- How can we find such points?

DFS for articulation pts

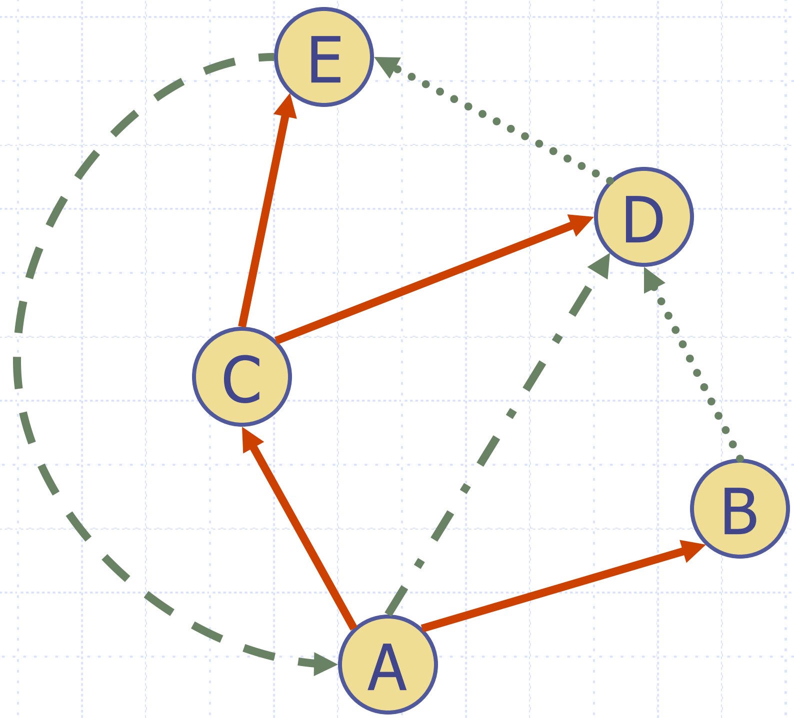
- Key idea—if I do a DFS, v cannot be an articulation point if it has a child that has a back edge to an ancestor (i.e. there is a cycle)
- Do a DFS to keep track of:
  - Order of visitation
  - lowest # back edge in descendents
- Finally, check if some child's "low" is at least as large as v's "num"
- Special case for root; if it has 2 (or more) children, it is automatically an articulation pt

Algorithm

- findArt(v)
  - v.visited = true
  - v.low=v.num = counter++ // low=num at start
  - foreach w adjacent to v, (v,w) not visited
    - if (!w.visited)
      - mark e= (v,w) visited
      - findArt(w)
      - if (w.low >= v.num) // no cycle back to anc. in decendants
        - output v as articulation pt
      - v.low=min(v.low,w.low); // record if cycle dec. to anc.
    - else
      - v.low = min(v.low, w.num) // back edge

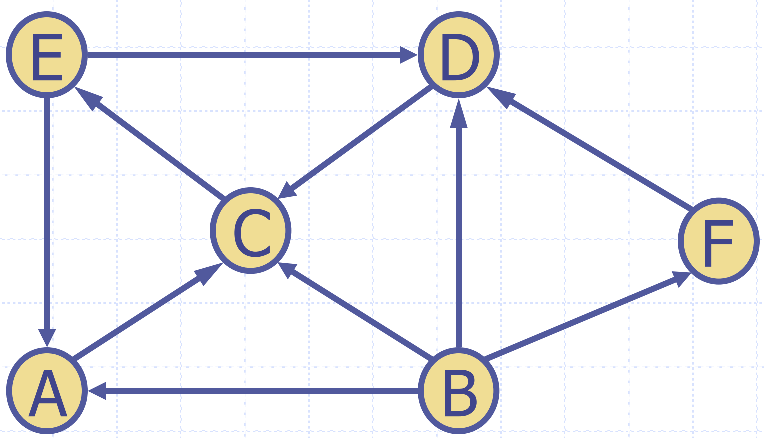
Directed DFS

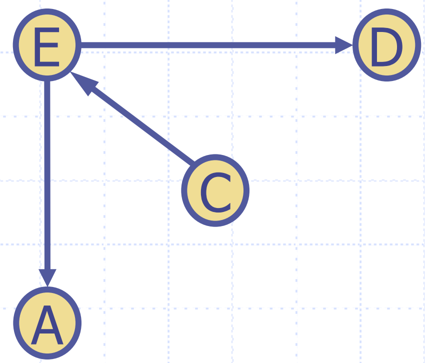
- We can specialize DFS and to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex s determines the vertices reachable from s

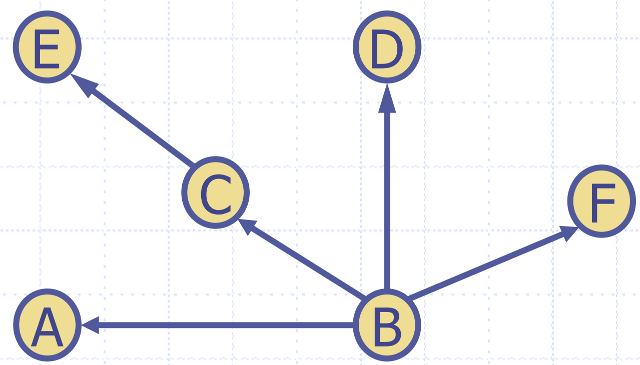


Reachability

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

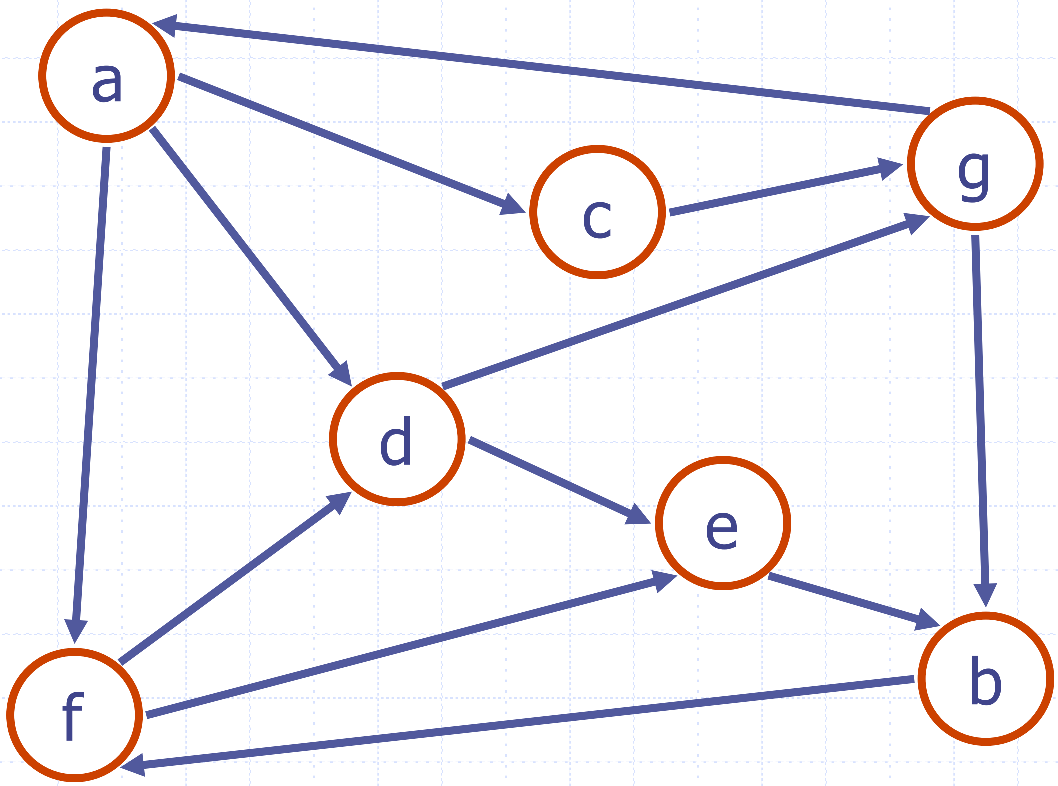




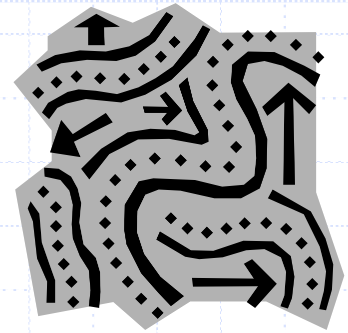


Strong Connectivity

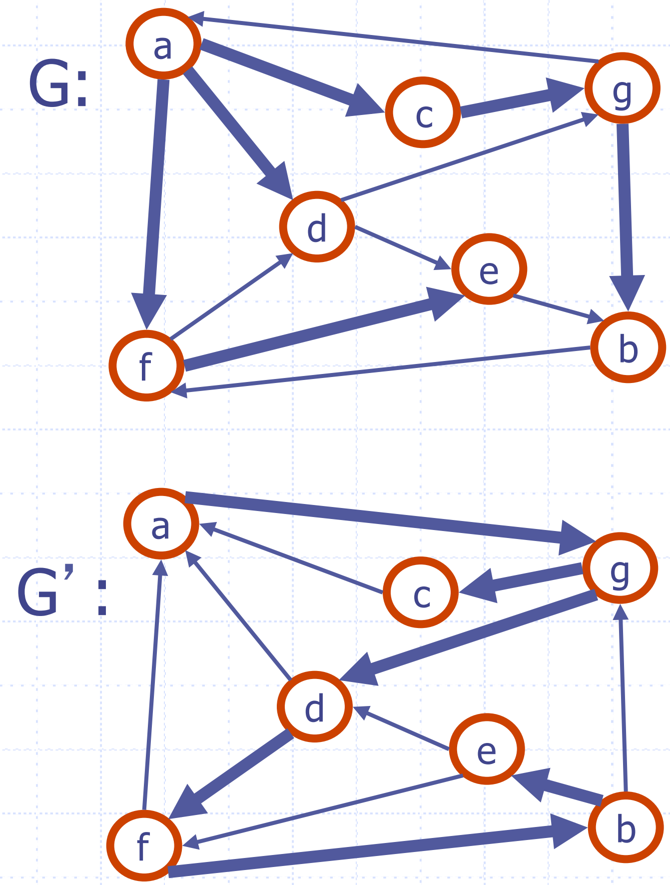
- Each vertex can reach all other vertices



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).



Topological Sorting Algorithm using DFS

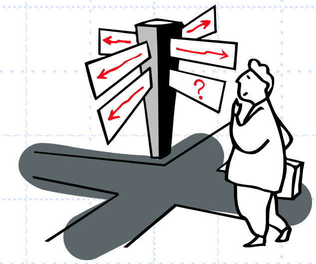
- Simulate the algorithm by using depth-first search

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

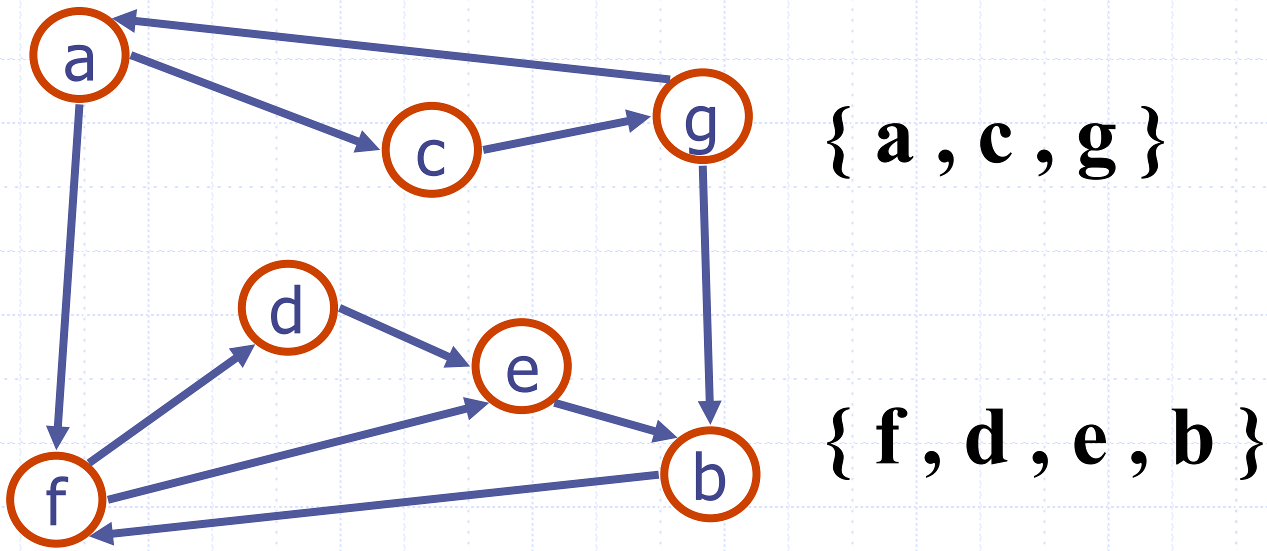
- O(n+m) time.

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

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).



Network Flow Problems

- What is the max flow from a source to a sink
- Dual problem is min cut (lowest cost to disconnect source from sink graph)
- Basic idea is to find paths from source to sink, compute flow, and keep track of **residual graph**

A possible algorithm sketch:

```
FG = RG = G
Set weights in FG to zero
while P = NonZeroPath(RG, s, t)
    FG = Addpath(FG, P, flow(P))
    RG = G – FG
end
```

Flow Path Finding

- We can specialize the DFS algorithm to find a nonzero flow path between two given vertices u and z using the template method pattern

```
Algorithm nonZeroPath(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    if flow(S) > 0
      return S.elements()
    else
      return null;
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        x = pathDFS(G, w, z)
        if (not x=null)
          return x
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
  return null
```

Network Flow Problems

- What is the max flow from a source to a sink
- Dual problem is min cut (lowest cost to disconnect source from sink)
- Basic idea is to find paths from source to sink, compute flow, and keep track of **residual graph**

A possible algorithm sketch:

```
FG = RG = G
Set weights in FG to zero
while P = NonZeroPath(RG, s, t)
    FG = Addpath(FG, P, flow(P))
    RG = G – FG
end
```

Where is the problem here?

Network Flow Problems

- What is the max flow from a source to a sink
- Dual problem is min cut (lowest cost to disconnect source and sink)
- Basic idea is to find paths from source to sink, compute flow, and keep track of **residual graph**

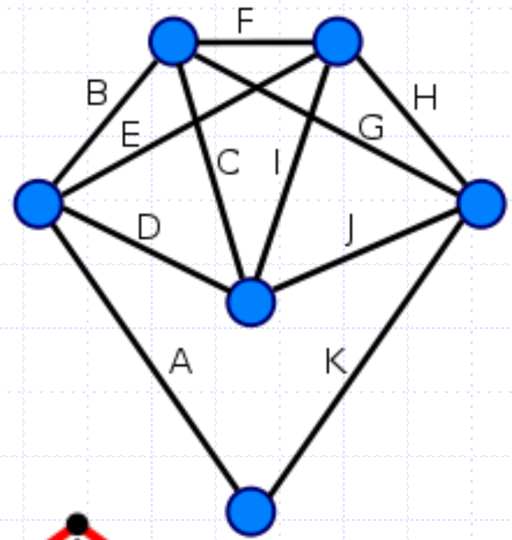
A possible algorithm sketch:

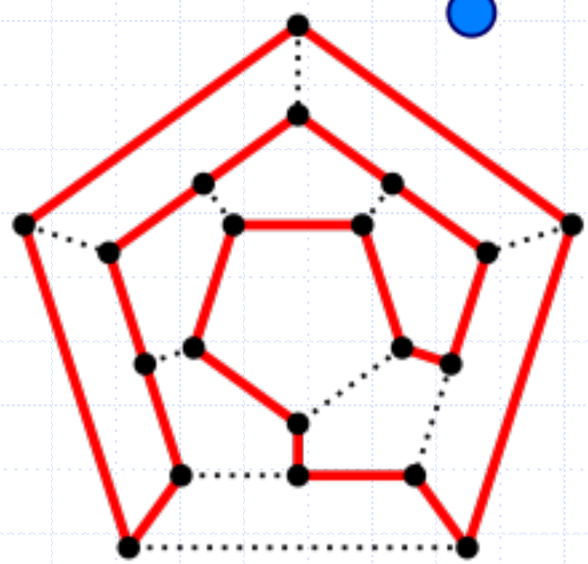
FG = RG = G
Set weights in FG to zero
while P = NonZeroPath(RG, s, t)
    FG = Addpath(FG, P, flow(P))
    RG = G – FG
    Augment(RG, P, G)
end

Good algorithms are
    $O(|E||V| + |V|^{2+e})$

A Few Words on Complexity

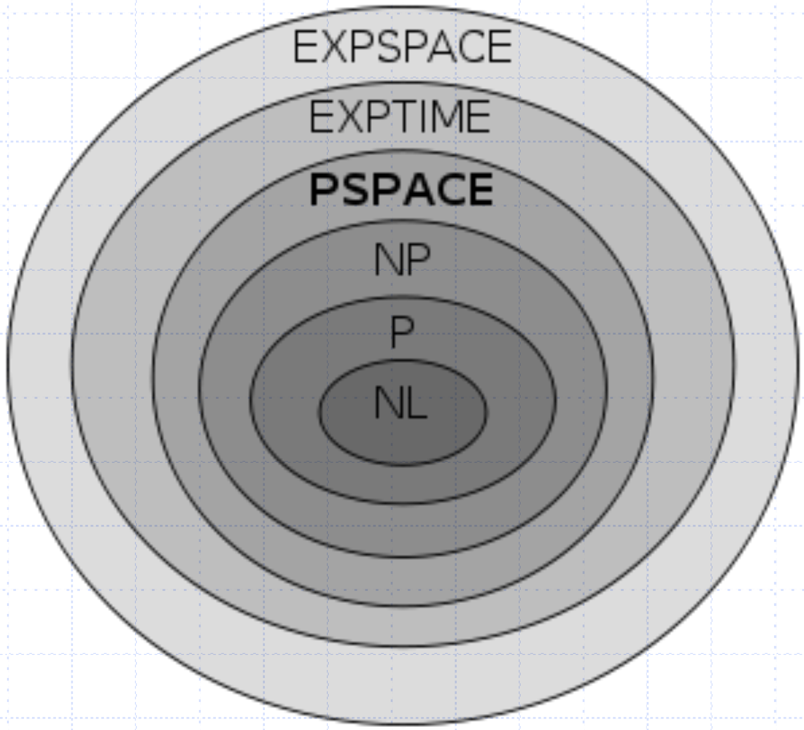
- Computational Problems are curiously brittle
  - Euler Tour – visit all edges once = polynomial time
  - Hamiltonian Cycle – visit all vertices once = very hard (exponential)





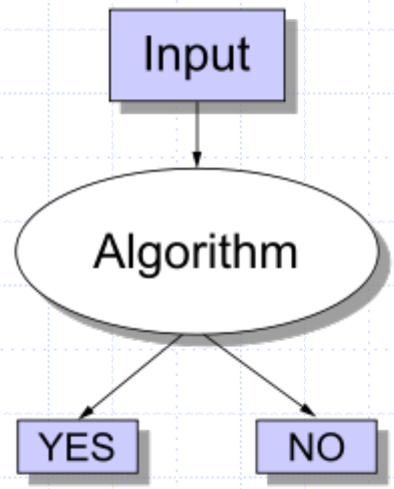
Complexity Theory

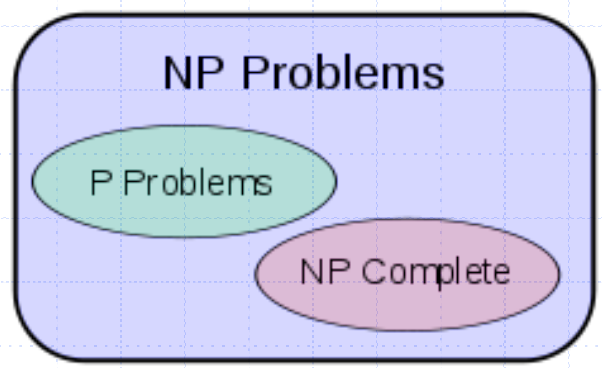
- Complexity theory studies the difficulty of computation problems
- The key is a complexity heirarchy of problems



P=NP is THE open question

- Consider only decision problems
- P – polynomial time
- NP – nondeterministic polynomial time
- NP complete – hardest problems in NP

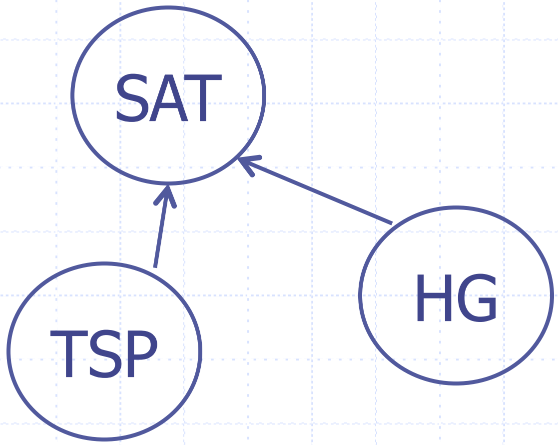




The recipe

- Establishing NP: Cook 1971 – any NP problem can be reduced to SAT
- Proving NP-complete
  - Show is in NP by exhibiting an algorithm
  - Show complete by reducing some known problem to it

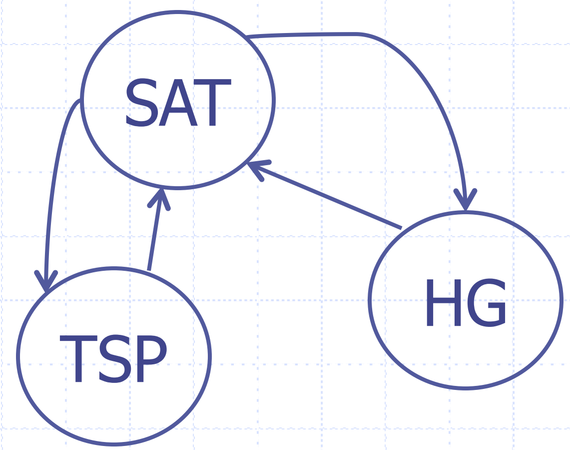
Cook's theorem



The recipe

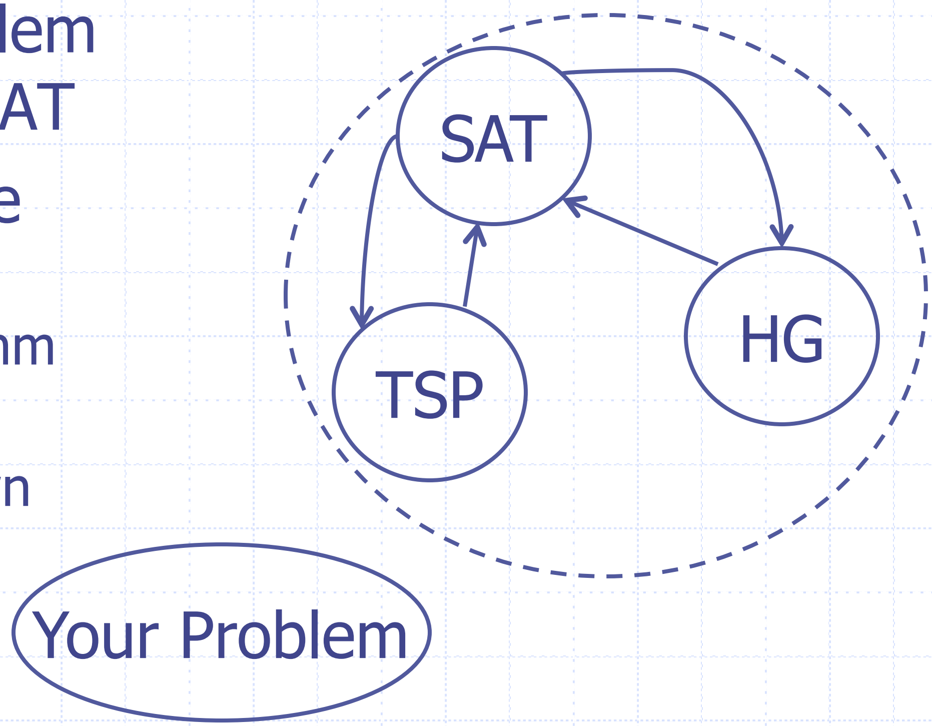
- Establishing NP: Cook 1971 – any NP problem can be reduced to SAT
- Proving NP-complete
  - Show is in NP by exhibiting an algorithm
  - Show complete by polynomial reduction of some known problem to it

Reduction



The recipe

- Establishing NP: Cook 1971 – any NP problem can be reduced to SAT
- Proving NP-complete
  - Show is in NP by exhibiting an algorithm
  - Show complete by reducing some known problem to it



http://en.wikipedia.org/wiki/List_of_NP-complete_problems

Even Worse

- The Halting Problem
  - Will a given program halt on a given input?
  - halt(prog)=> yes/no
  - Loop(P)
    - If (halt(P(P))) inf loop
      - Else halt
  - What is Loop(Loop)?
- If loop(loop) halts, then loop(loop)=inf loop
- If loop(loop) is inf loop, then loop(loop) halts

Summary

- Graphs - directed/undirected weighted
- Data structures
- Traversals (BFS, DFS)
  - what you can compute with them
- Shortest path
- Minimum Spanning Trees