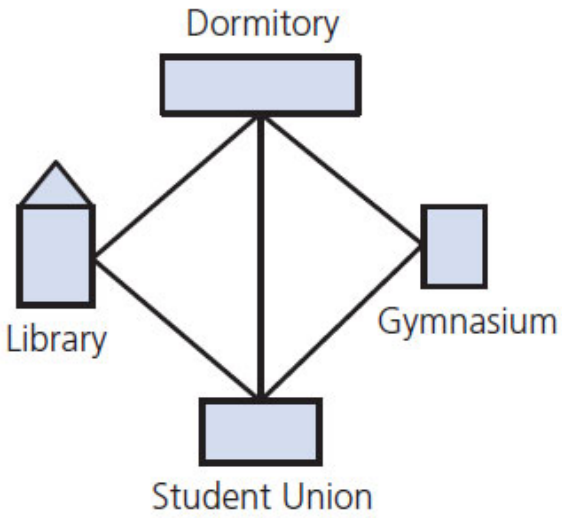
EECE 2560:
Fundamentals of Engineering Algorithms

Graphs

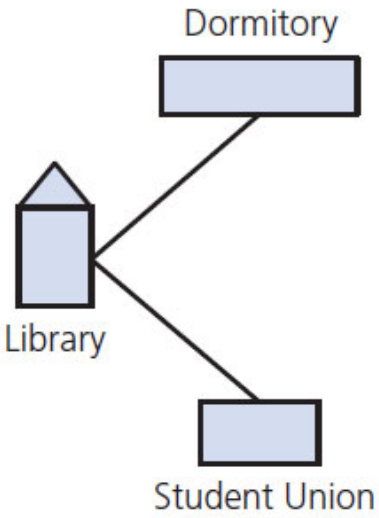
What is a Graph?

- A **graph** represents relations among data items
- $G = \{V, E\}$
  - A graph is a set of vertices (nodes) V and
  - A set of edges E that connect the vertices.
- A **subgraph** consists of a subset of a graph's vertices and a subset of its edges.

(a) A campus map as a graph



(b) A subgraph of the graph in part a

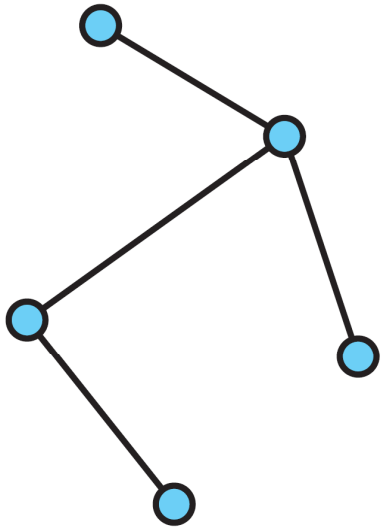


Graphs Terminologies

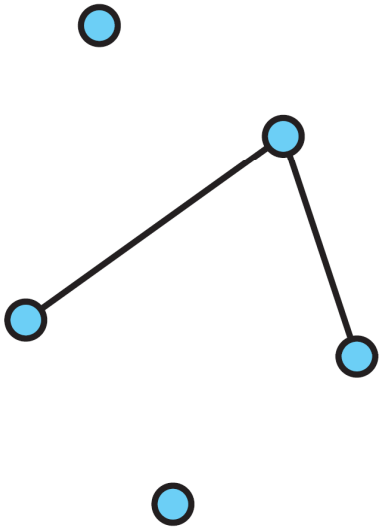
- Two vertices of a graph are **adjacent** if they are joined by an edge.
- A **path** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex.
- Although a path may pass through the same vertex more than once a **simple path** may not.
- A **cycle** is a path that begins and ends at the same vertex;
- A **simple cycle** is a cycle that does not pass through other vertices more than once.
- A graph with no cycles is ***acyclic***.
- A **simple** graph is a graph with no duplicate edges and no **self-loop** (i.e., an edge that connects a vertex to itself).

Types of Graphs (1 of 3)

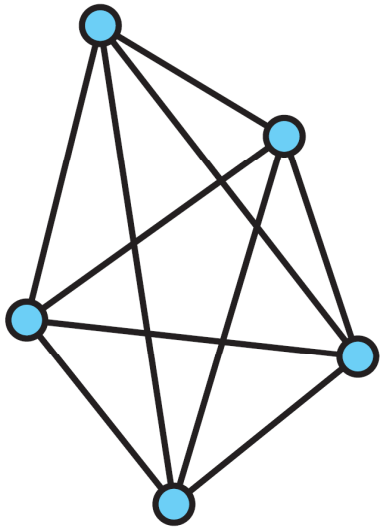
- A graph is **connected** if each pair of distinct vertices has a path between them.
- In a **complete graph** , each pair of distinct vertices has an edge between them.



Connected



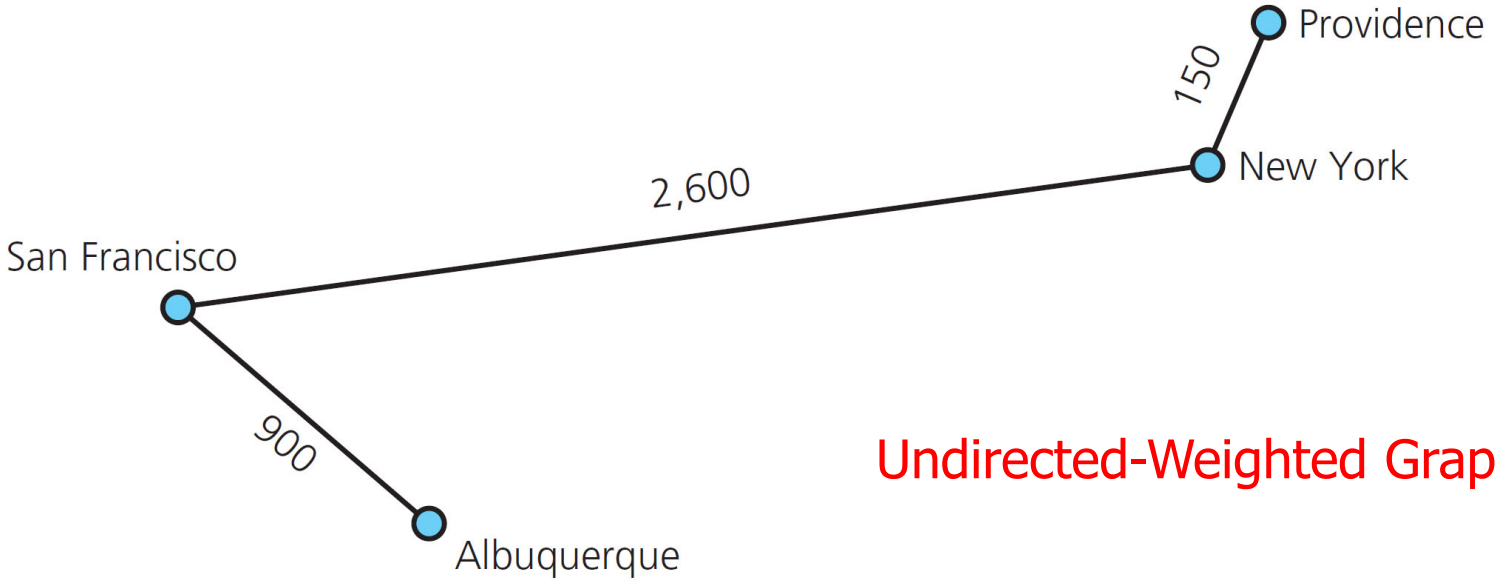
Disconnected



Complete

Types of Graphs (2 of 3)

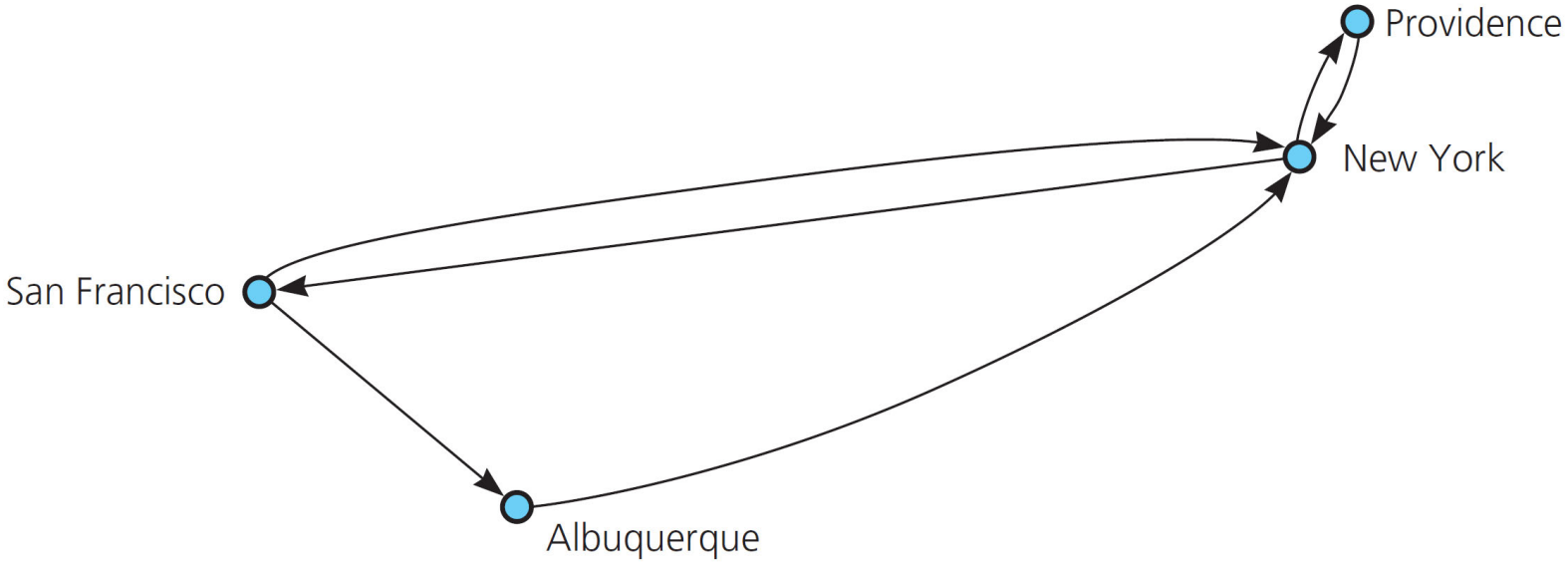
- A **weighted graph** is a graph with edges labeled with numeric values.
- An **undirected graph** has edges with no direction. That is, you can travel in either direction along the edges between the vertices of an undirected graph.
- A connected, acyclic, undirected graph is a **tree**.



Undirected-Weighted Graph

Types of Graphs (3 of 3)

- Each edge in a **directed graph**, or **digraph**, has a direction and is called a **directed edge**.



Directed-Unweighted Graph

Graphs Theorems

- A simple undirected graph is a **tree** if it is connected and contains no cycles.
  - Any vertex in this graph can be selected as the root node of the tree.

- For an *n*-node graph *G, a*ny two of the following implies the third:
  - *G* is connected
  - *G* contains no cycles
  - *G* has $n - 1$ edges.

Graphs as ADTs

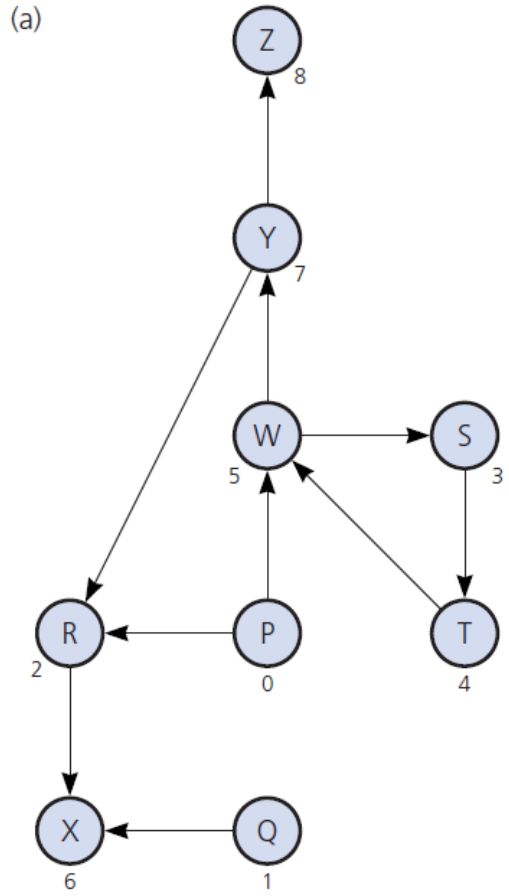
- ADT graph operations
  - Test if empty
  - Get number of vertices, edges in a graph
  - See if edge exists between two given vertices
  - Add vertex to graph whose vertices have distinct, different values from new vertex
  - Add/remove edge between two given vertices
  - Remove vertex, edges to other vertices
  - Retrieve vertex that contains given value

- Insertion and removal operations are somewhat different for graphs than for other ADTs in that they apply to both vertices or edges.

Graphs Implementation (1 of 5)

- Using adjacency matrix to implement a directed graph:

(a)

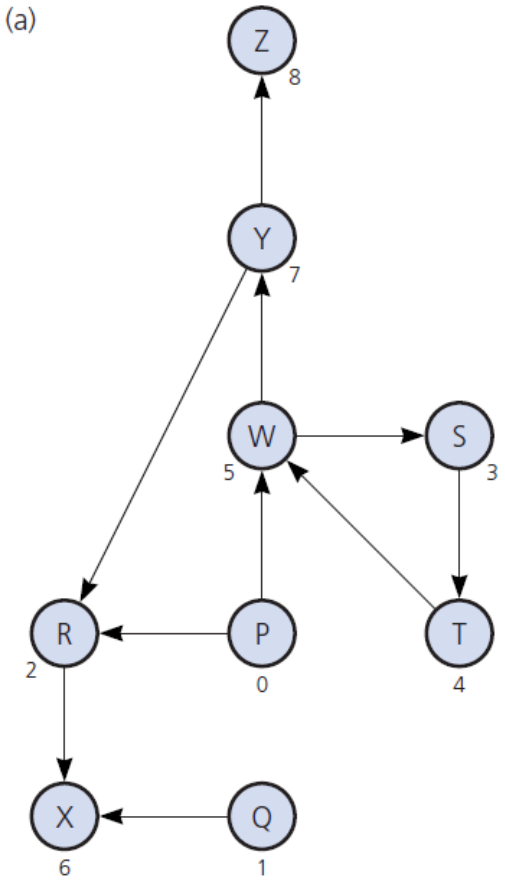


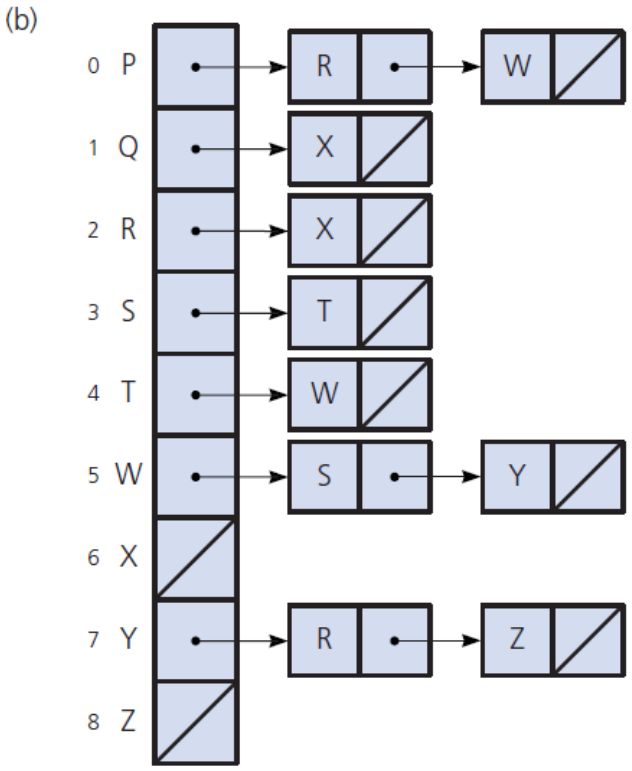
(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Graphs Implementation (2 of 5)

- Using adjacency list to implement a directed graph:



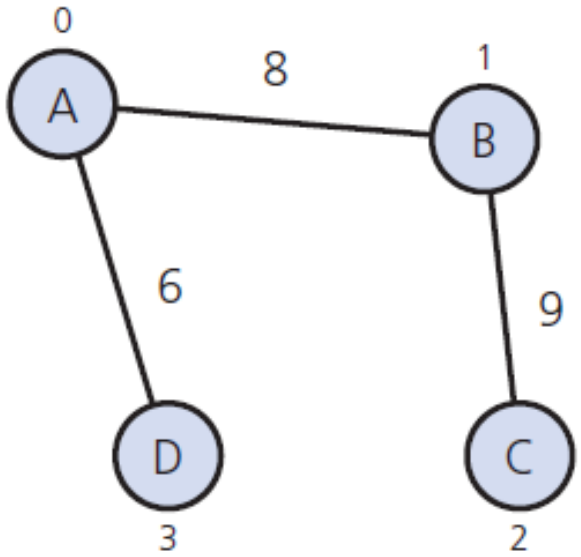


The sum of the lengths of all the adjacency lists is |E|

Graphs Implementation (3 of 5)

- Using adjacency matrix to implement a weighted undirected graph:

(a)

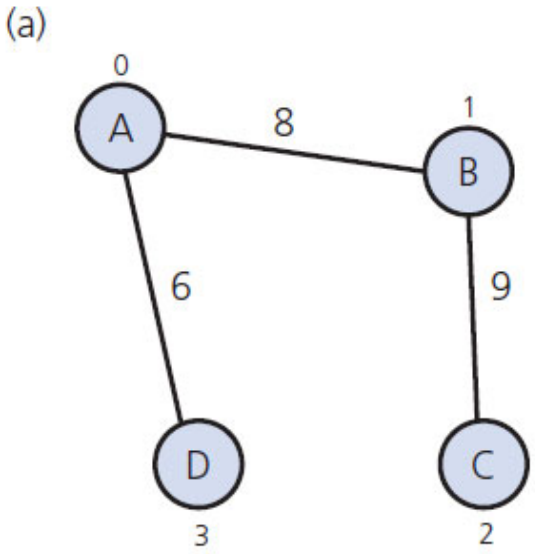


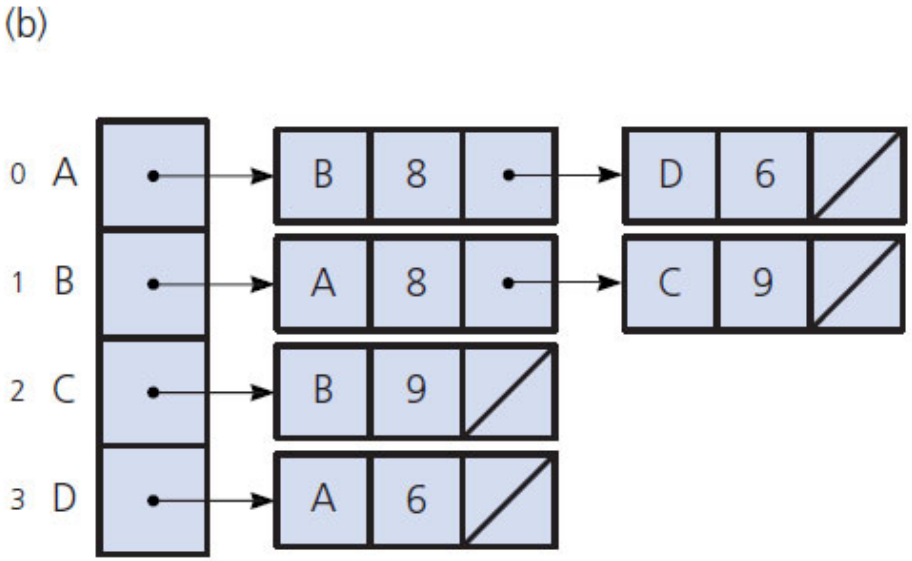
(b)

		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Graphs Implementation (4 of 5)

- Using adjacency list to implement a weighted undirected graph:





The sum of the lengths of all the adjacency lists is 2|*E*|

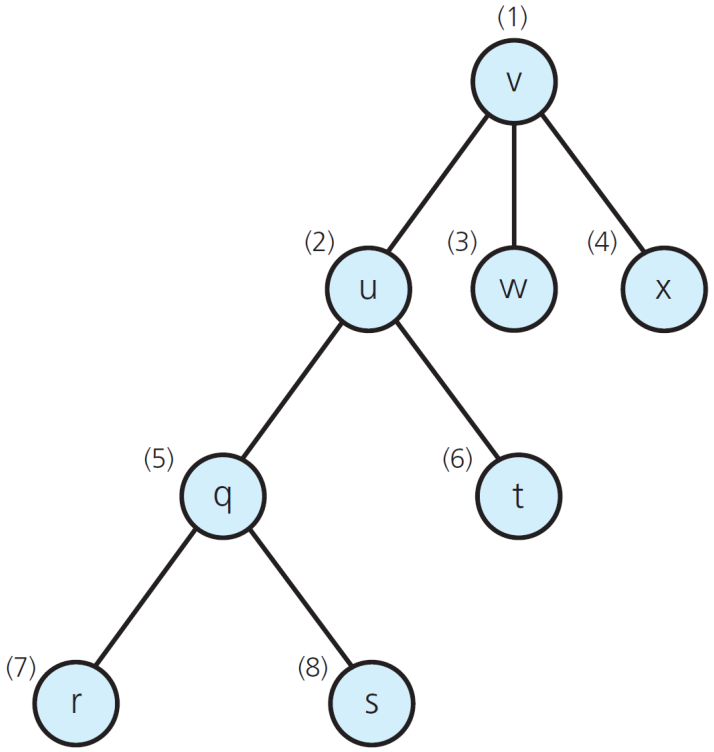
Graphs Implementation (5 of 5)

- A ***Sparse*** graph is a graph with $|E|$ is much less than $|V|^2$.
- A ***Dense*** graph is a graph with $|E|$ is close to $|V|^2$.
- Adjacency matrix representation:
  - Is usually the method of choice for ***dense*** graphs
  - Supports process of seeing if there exists an edge from vertex *i* to vertex *j*
- Adjacency list representation:
  - Is usually the method of choice for ***sparse*** graphs.
  - Often requires less space than adjacency matrix
  - Supports finding vertices adjacent to given vertex

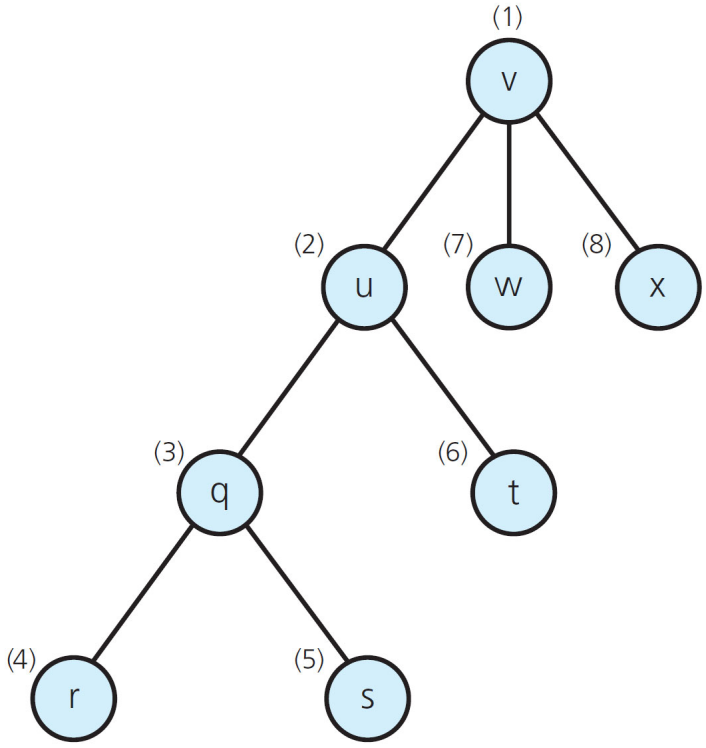
Graph Traversals

- Given a graph $G = (V, E)$ and a distinguished ***source*** vertex s, graph traversal means systematically explores the edges of G to "discover" every vertex that is reachable from s.
  - Graph traversal also answers the question: is there a path from s to t?
- There are two approaches of graph traversal:
  - **Breadth-First Search** (BFS) discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.
  - **Depth-First Search** (DFS) searches "deeper" in the graph whenever possible before backing up.
- Unlike a tree traversal, which always visits all of the nodes in a tree, a graph traversal does not necessarily visit all of the vertices in the graph unless the graph is connected.
  - If a graph traversal does not visit all vertices in the graph, then the graph is **not connected**.

BFS vs DFS



BFS Example



DFS Example

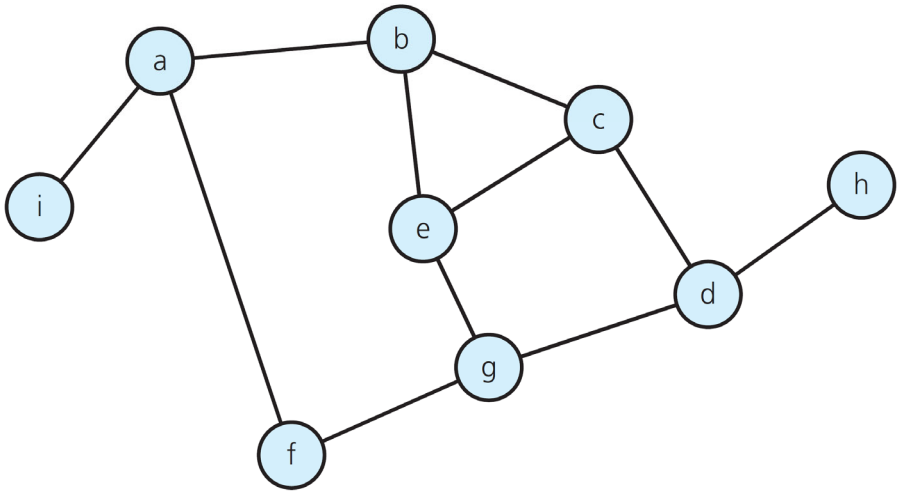
Breadth-First Search (1 of 2)

- BFS strategy: "**first visited**" is "**first explored**"
- This strategy is reflected in using a **queue** as in the following algorithm:

```
bfs(v: Vertex)
 q = a new empty queue
 q.enqueue(v) // Add v to queue
 Mark v as visited
 while (!q.isEmpty()) {
   q.dequeue(w)
   for (each unvisited vertex u adjacent to w) {
     Mark u as visited
     q.enqueue(u)
   }
}
```

Breadth-First Search (2 of 2)

- The results of a breadth-first traversal, beginning at vertex *a*, of the following graph, is: *a, b, f, i, c, e, g, d, h*




```
bfs(v: Vertex)
 q = a new empty queue
 q.enqueue(v) // Add v to queue
 Mark v as visited
 while (!q.isEmpty()) {
   q.dequeue(w)
   for (each unvisited vertex u adjacent to w) {
     Mark u as visited
     q.enqueue(u) }
}
```

Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

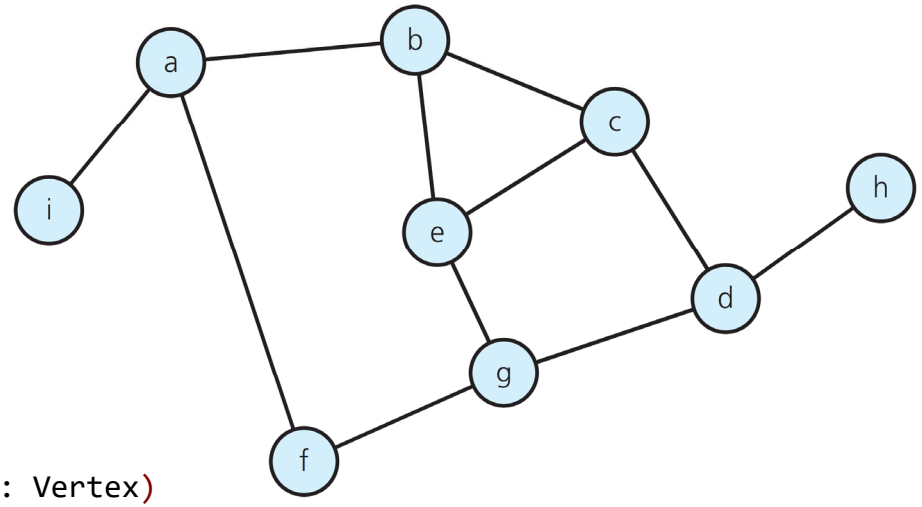
Depth-First Search (1 of 2)

- DFS strategy: "**last visited**" is "**first explored**"
- This strategy is reflected in using a **stack** as in the following algorithm:

```
dfs(v: Vertex)
 s= a new empty stack
 s.push(v) // Push v onto the stack
 Mark v as visited
 while (!s.isEmpty()) {
   if (no unvisited vertices are adjacent to
           the vertex on the top of the stack)
     s.pop() // Backtrack
   else {
    Select an unvisited vertex u adjacent to the
      vertex on the top of the stack
    s.push(u)
    Mark u as visited }
  }
```

Depth-First Search (2 of 2)

- The results of a depth-first traversal, beginning at vertex *a*, of the following graph, is: *a, b, c, d, g, e, f, h, i*




```
dfs(v: Vertex)
 s= a new empty stack
 s.push(v) // Push v onto the stack
 Mark v as visited
 while (!s.isEmpty()) {
   if (no unvisited vertices adjacent to top vertex)
     s.pop() // Backtrack
   else {
    Select an unvisited vertex u adjacent to top vertex
    s.push(u)
    Mark u as visited }
 }
```

Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

Detect Cycles in a Graph (1 of 3)

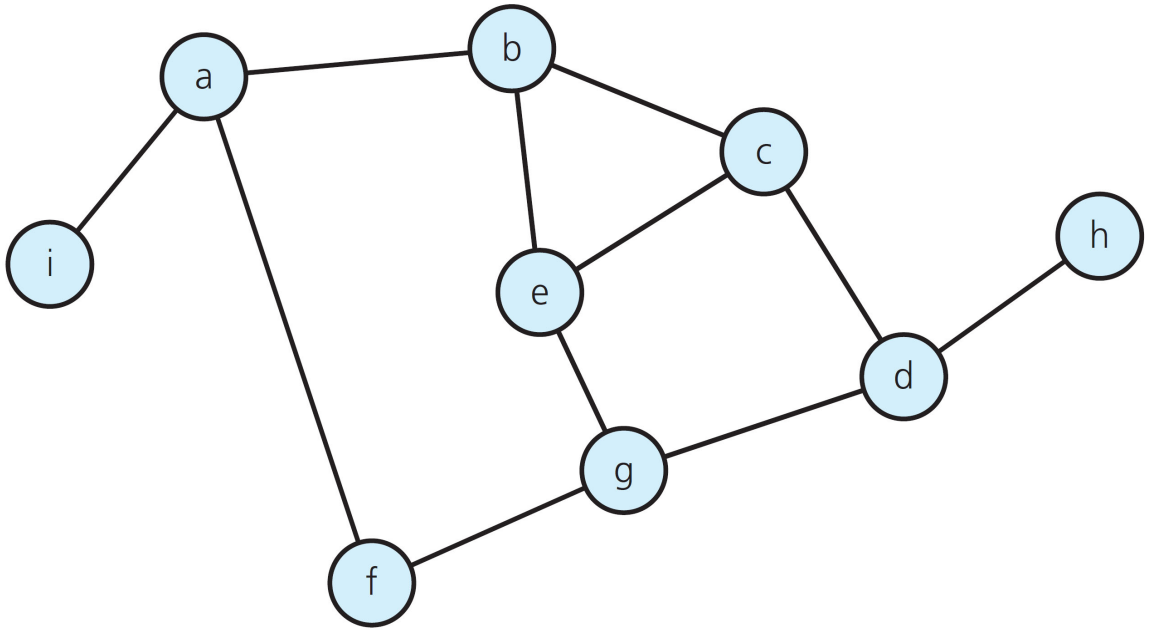
- The following is a modified DFS algorithm to determine whether a graph contains a cycle.
- In this algorithm, each vertex has the following attributes:
  - **`visited`**: to indicate whether the vertex has been visited.
  - **`predecessor`**: to mark its predecessor vertex on the DFS path.
  - **`onStack`**: to indicate whether the vertex is still in the stack.
  - **`finished`**: to indicate that the vertex is not in the stack anymore.
- If the vertex at the top of the stack has a neighbor that is still in the stack and it is not its predecessor, then a cycle exists.

Detect Cycles in a Graph (2 of 3)

```
hasCycle(v: Vertex): Boolean {
 s = a new empty stack
 Mark all vertices as unvisited and not onStack
 s.push(v)
 v.onStack = true
 while (!s.isEmpty()) {
  top = s.peek()
  if (top has a neighbor marked as onStack and
       it is not top's predecessor)
            return true // A cycle exists
  else if (top has an unvisited neighbor u)   {
             u.onStack=true
             s.push(u)                          }
        else { top.finished=true
               s.pop()}
 }
 return false
}
```

Detect Cycles in a Graph (3 of 3)

- Applying the `hasCycle` algorithm on the following graph, starting from vertex *a,* will result in a stack with the following contents (from bottom to top): *a, b, c, d, g, e*
- As the top of the stack, vertix *e*, has a neighbor marked as `onStack` (vertic *b*), then the algorithm will return `true`.

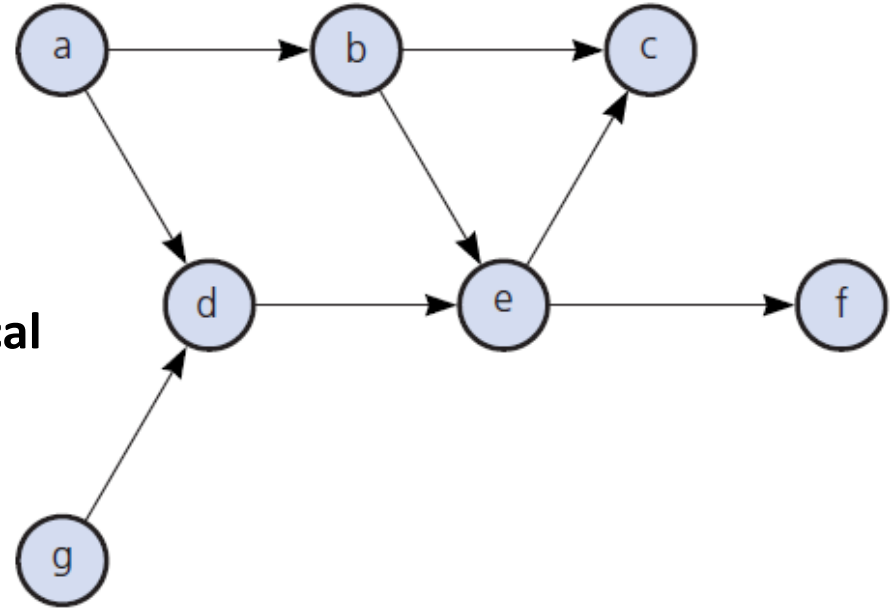


Graphs Applications

- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits
- Traveling Salesperson

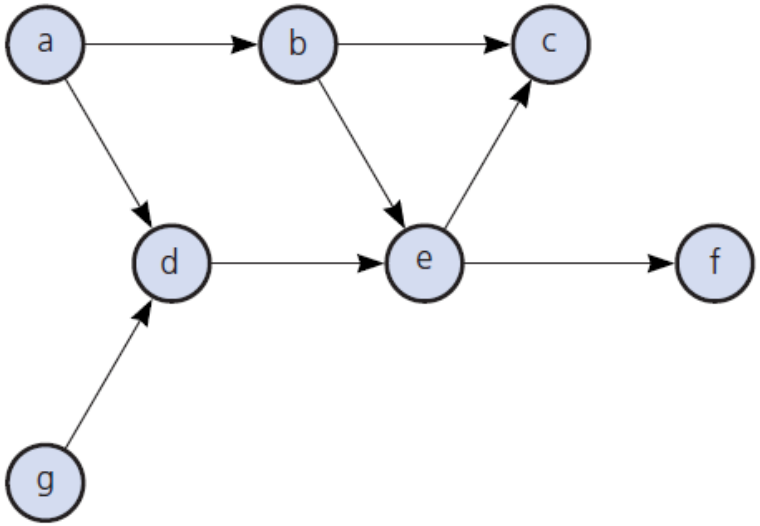
Topological Sorting

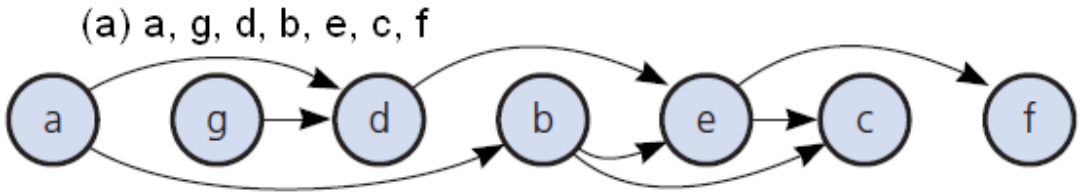
- The shown directed graph without cycles has a natural order. For example, vertex *a* precedes *b*, which precedes *c*.
- If the vertices represent academic courses, the graph represents the prerequisite structure for the courses.
- In what order should you take all seven courses so that you will satisfy all prerequisites?
- **Topological sorting** provides a linear order of the vertices in a directed graph without cycles that answers this question.
- That linear order is called **topological order**.
- In a list of vertices in topological order, vertex *x* precedes vertex *y* if there is a directed edge from *x* to *y* in the graph.

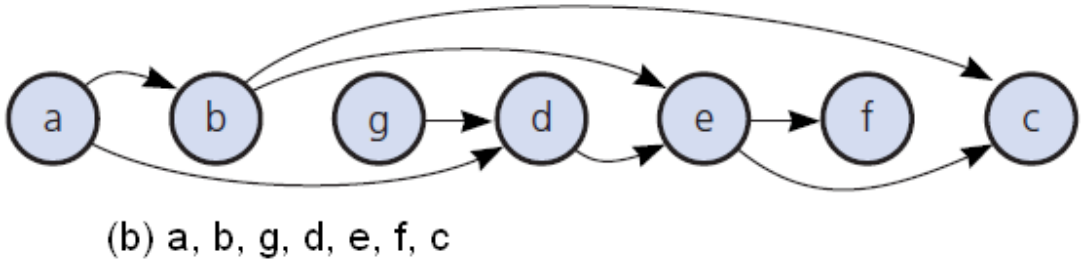


Topological Order Examples

- If you arrange the vertices of a directed graph linearly and in a topological order, the edges will all point in one direction.
- As shown, the vertices in a given graph may have several topological orders.





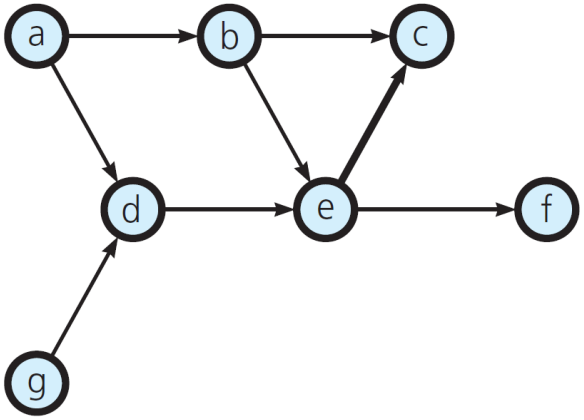
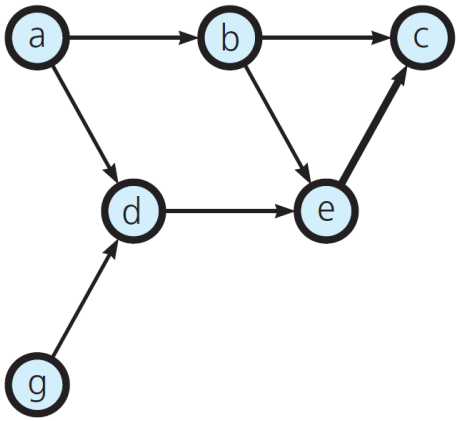
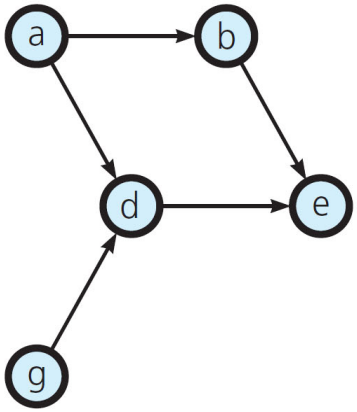
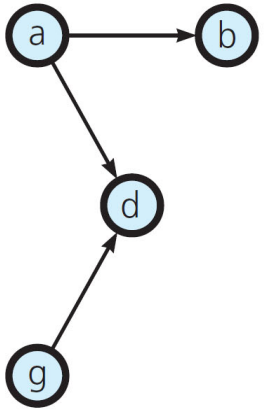


Topological Sorting Algorithm

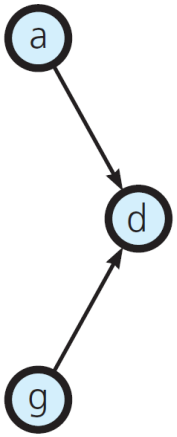
- There are several simple algorithms for finding a topological order.
- The following algorithm arranges the vertices in graph `theGraph` into a topological order by:
  1. Finding a vertex that has no successor.
  2. Removing from the graph this vertex and all edges that lead to it, and adding it to the beginning of a list.
  3. Adding each subsequent vertex that has no successor to the beginning of the list.
  4. When the graph is empty, the list of vertices will be in topological order.

```
topSort1(theGraph: Graph, aList: List)
        n = number of vertices in theGraph
        for (step = 1 through n)
        {
          Select a vertex v that has no successors
          aList.insert(1, v)
          Remove from theGraph vertex v and its edges
        }
```

Topological Sorting Example (1 of 2)

theGraph	aList
	f
	cf
	ecf
	becf

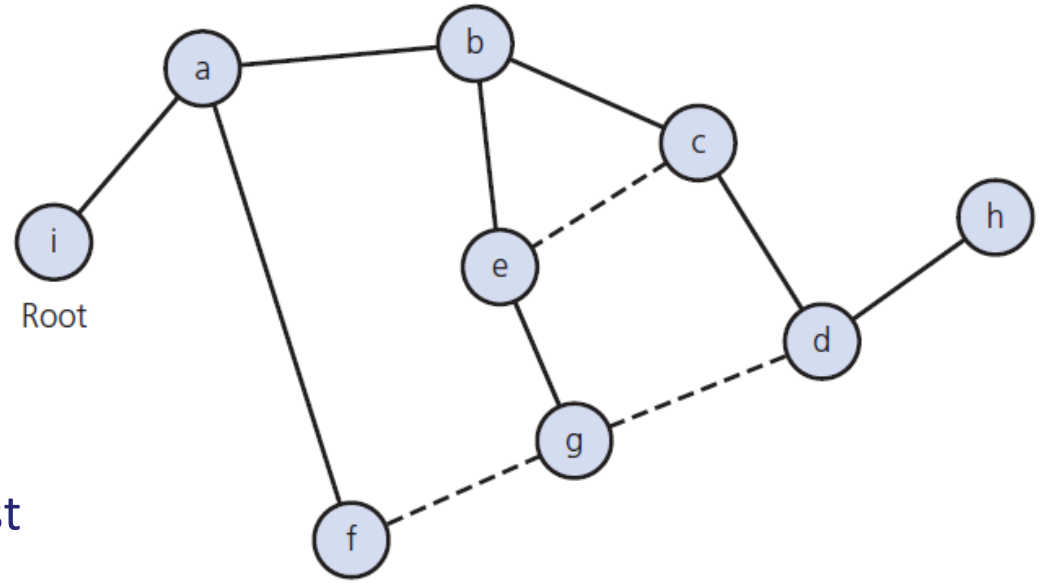
Topological Sorting Example (2 of 2)

theGraph	aList
	dbecff
	gdbecf

theGraph	aList
	agdbecf

Spanning Trees

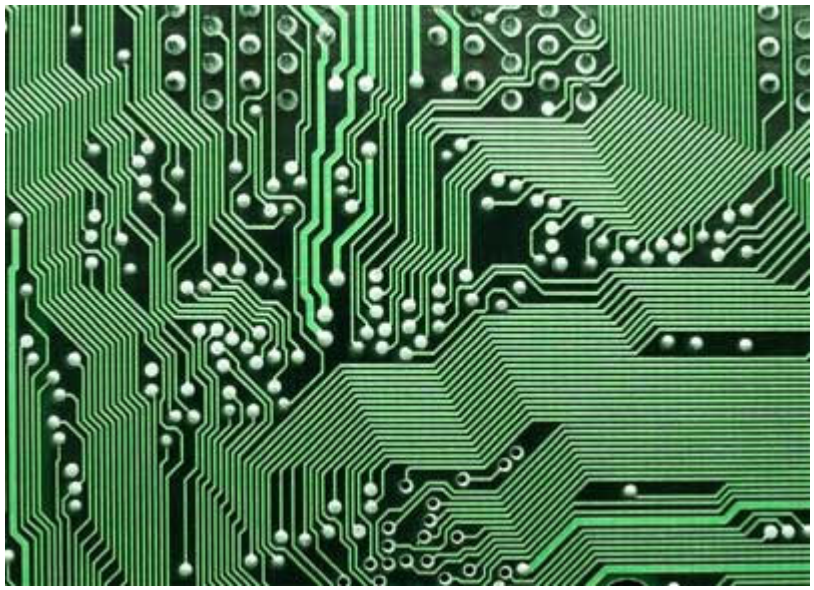
- A tree is an undirected connected graph without cycles.
- Detecting a cycle in an undirected graph
  - Connected undirected graph with *n* vertices must have at least *n* – 1 edges.
  - If it has exactly *n* – 1 edges, it cannot contain a cycle
  - With more than *n* – 1 edges, must contain at least one cycle



- A **spanning tree** of a connected undirected graph *G* is a subgraph of *G* that contains all of *G* 's vertices and enough of its edges to form a tree (as shown in the figure).
- There may be several spanning trees for a given graph.

Spanning Trees Application

- Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of *n* pins, we can use an arrangement of *n* - 1 wires, each connecting two pins. Which means: finding a spanning tree that connects these pins.

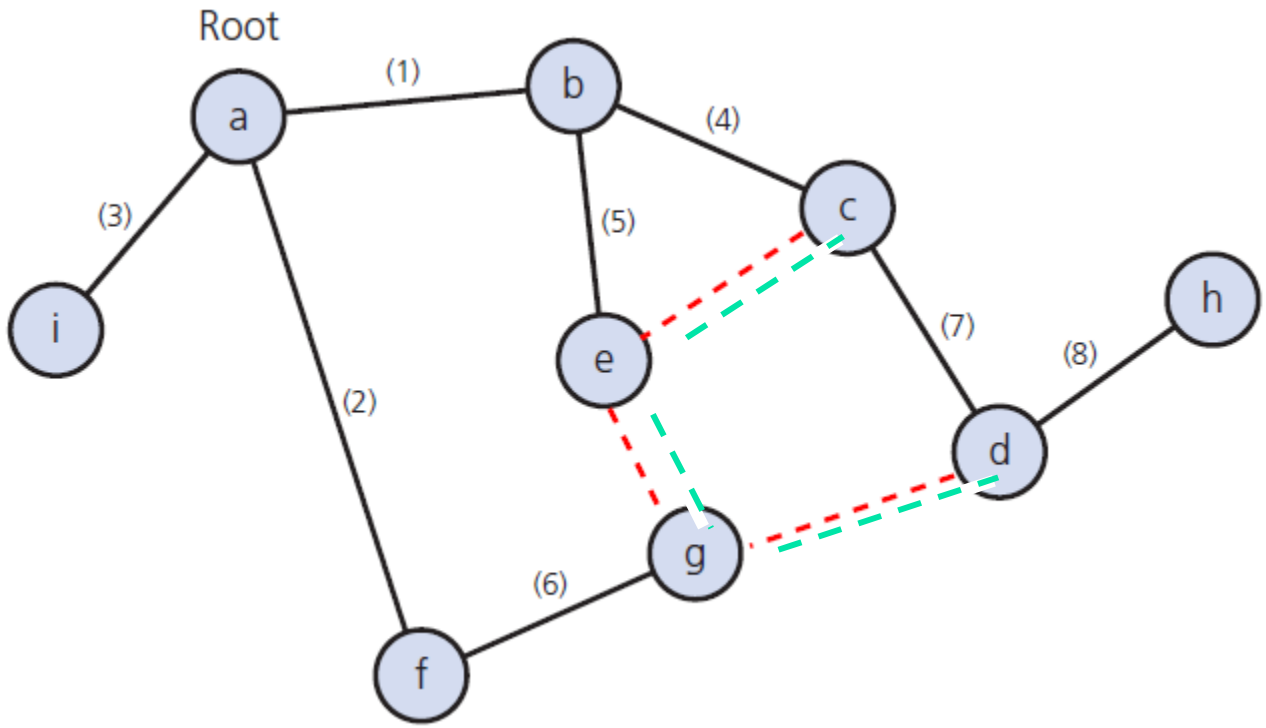


©2009 www.phys.org

Spanning Trees Algorithms

- Two algorithms for determining a spanning tree of a graph are based on the previous traversal algorithms: BFS and DFS.
- In general, these algorithms will produce different spanning trees for any particular graph.

BFS Spanning Trees



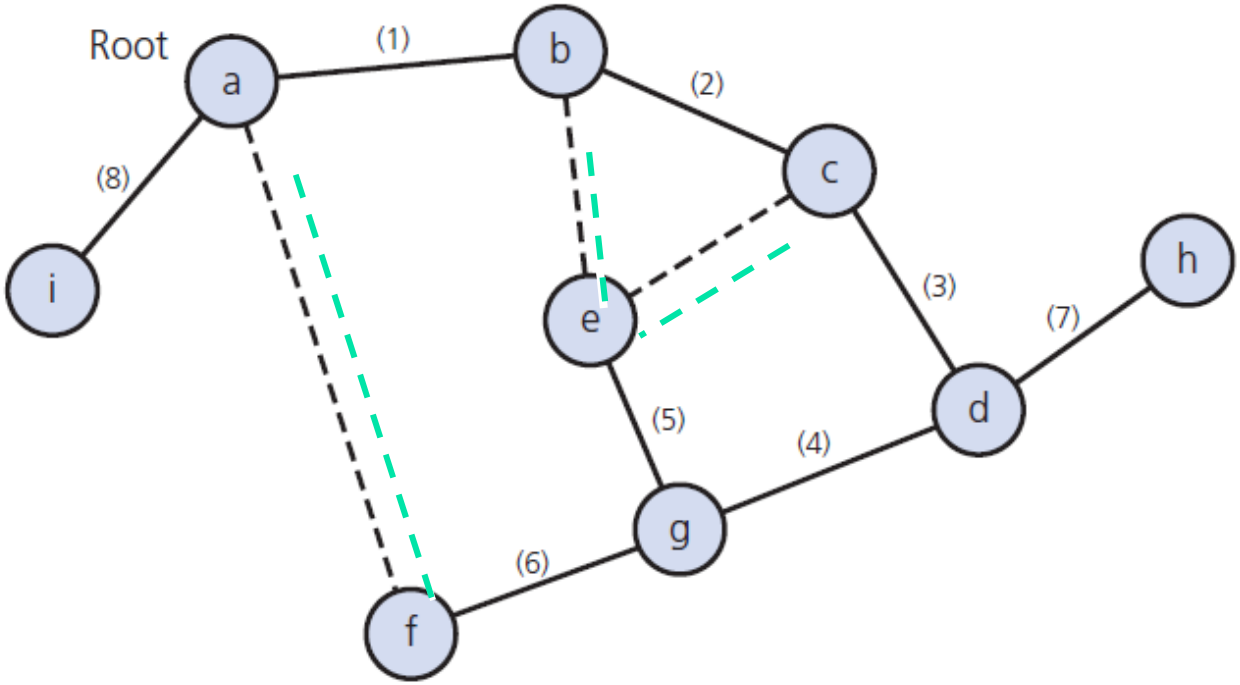
The BFS spanning tree algorithm visits vertices in this order:

a, b, f, i, c, e, g, d, h

Numbers indicate the order in which the algorithm marks edges.

The algorithm can keep track of these edges by maintaining a predecessor attribute with every node as the algorithm proceeds.

DFS Spanning Trees



The DFS spanning tree algorithm visits vertices in this order:

a, b, c, d, g, e, f, h, i

Numbers indicate the order in which the algorithm marks edges.

The algorithm can keep track of these edges by maintaining a predecessor attribute with every node as the algorithm proceeds.

Minimum Spanning Trees

- The **minimum spanning tree (MST)** of a weighted-connected undirected graph is its spanning tree that has the least cost.
  - A tree cost is the sum of its edges weights (costs).

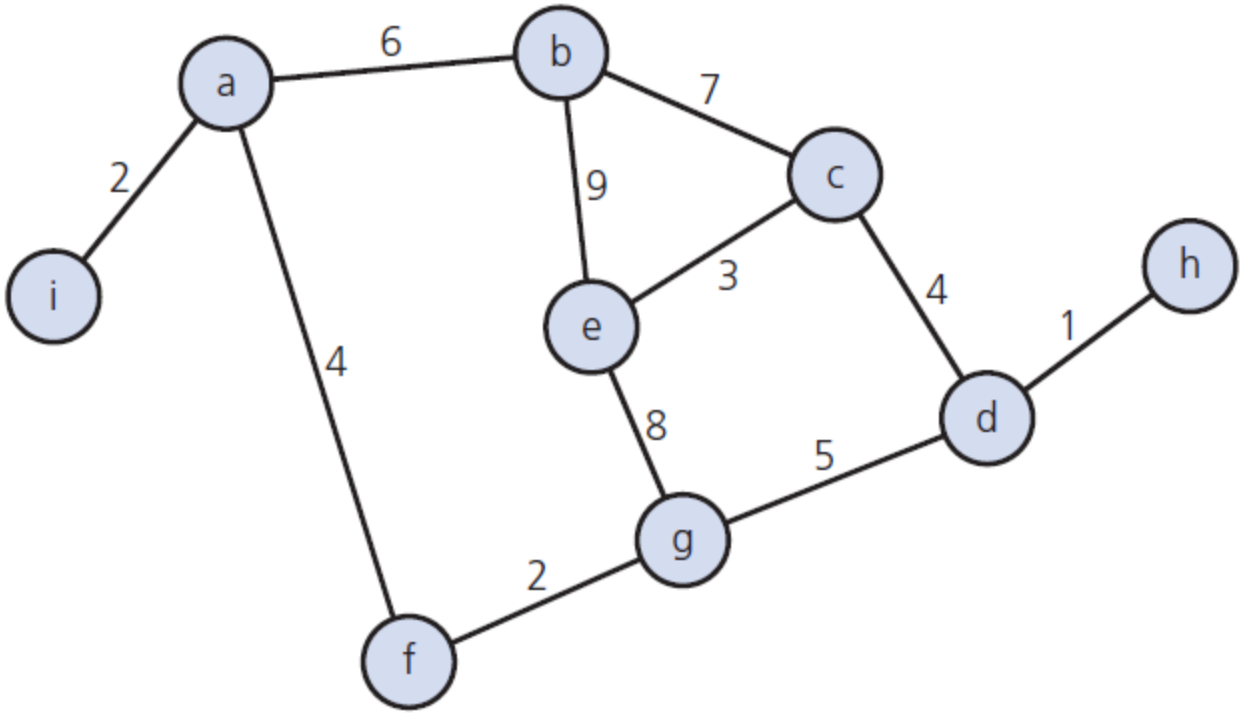
- Although there may be several minimum spanning trees for a particular graph, their costs are equal.

MST: Prim's Algorithm

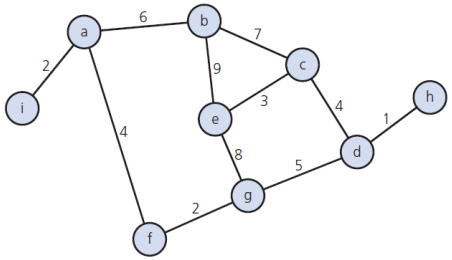
```
// Determines a minimum spanning tree for a weighted,
// connected, undirected graph whose weights are
// nonnegative, beginning with any vertex v.
primsAlgorithm(v: Vertex)
  Mark vertex v as visited and include it
       in the minimum spanning tree
  while (there are unvisited vertices)
  {
     Find the least-cost edge (w, u) from a visited
        vertex w to some unvisited vertex u
     Mark u as visited
     Add the vertex u and the edge (w, u) to the
        minimum spanning tree
  }
```

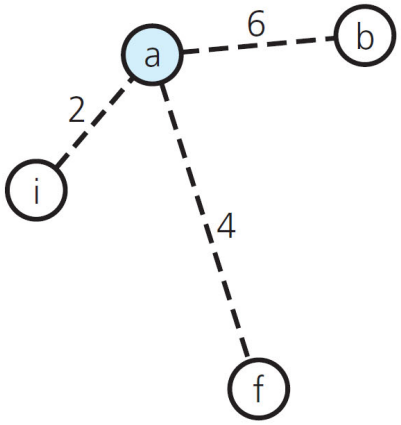
MST Prim Example (1 of 4)

Use Prim's algorithm to find the MST of the following weighted, connected, undirected graph

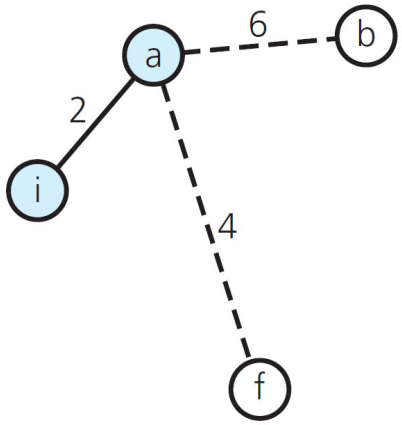


MST Prim Example (2 of 4)

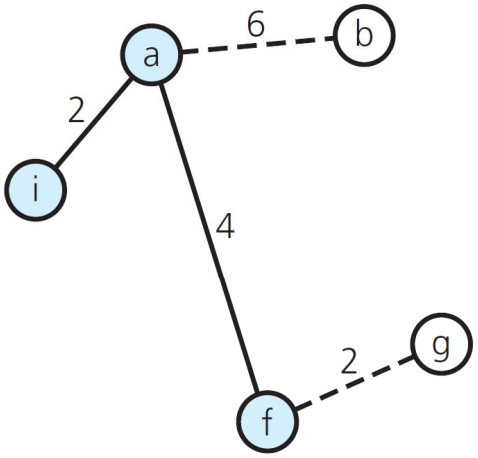




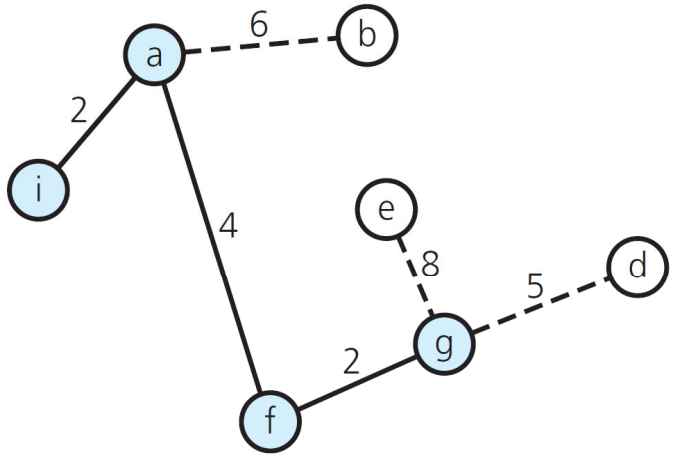
(a) Mark a, consider edges from a



(b) Mark i, include edge (a, i)

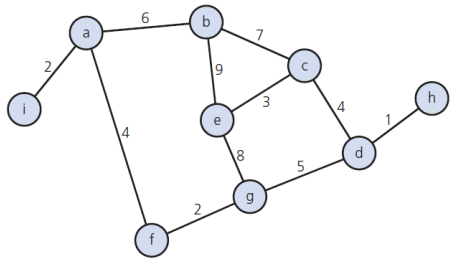


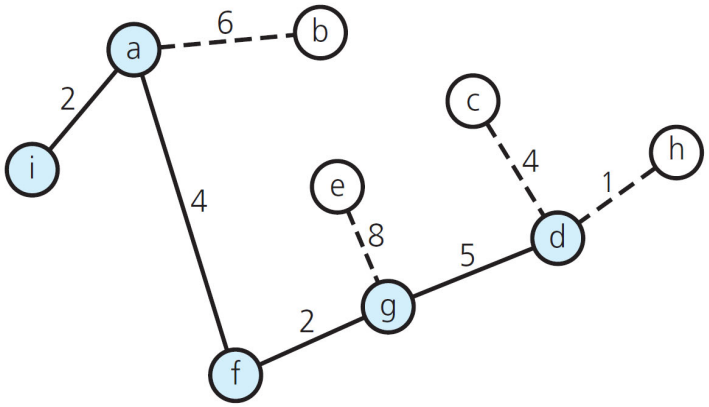
(c) Mark f, include edge (a, f)



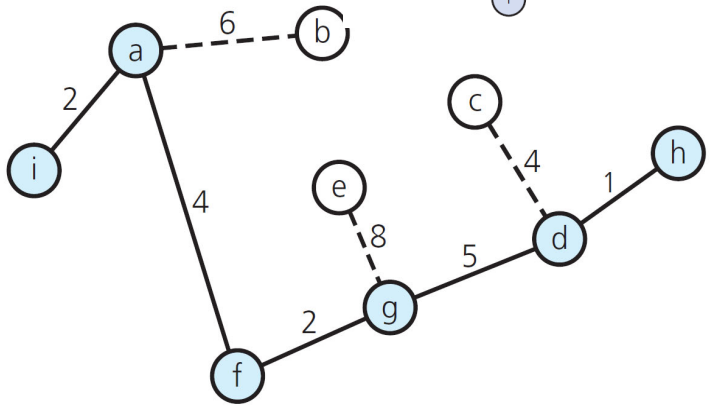
(d) Mark g, include edge (f, g)

MST Prim Example (3 of 4)

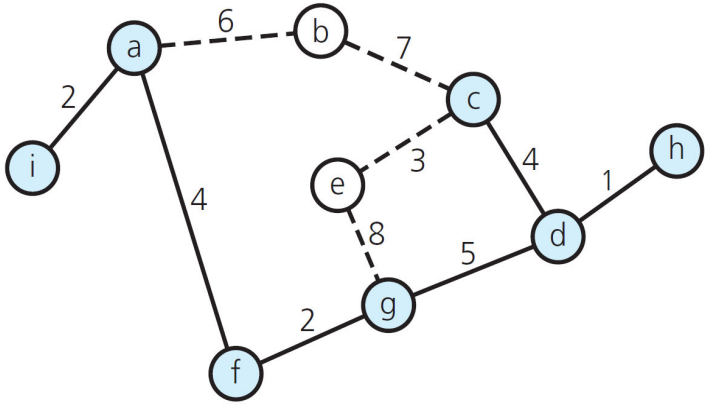




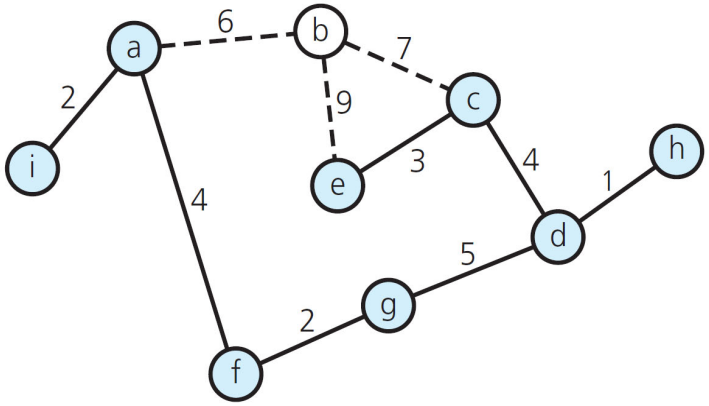
(e) Mark d, include edge (g, d)



(f) Mark h, include edge (d, h)

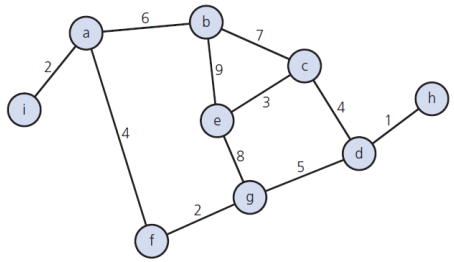


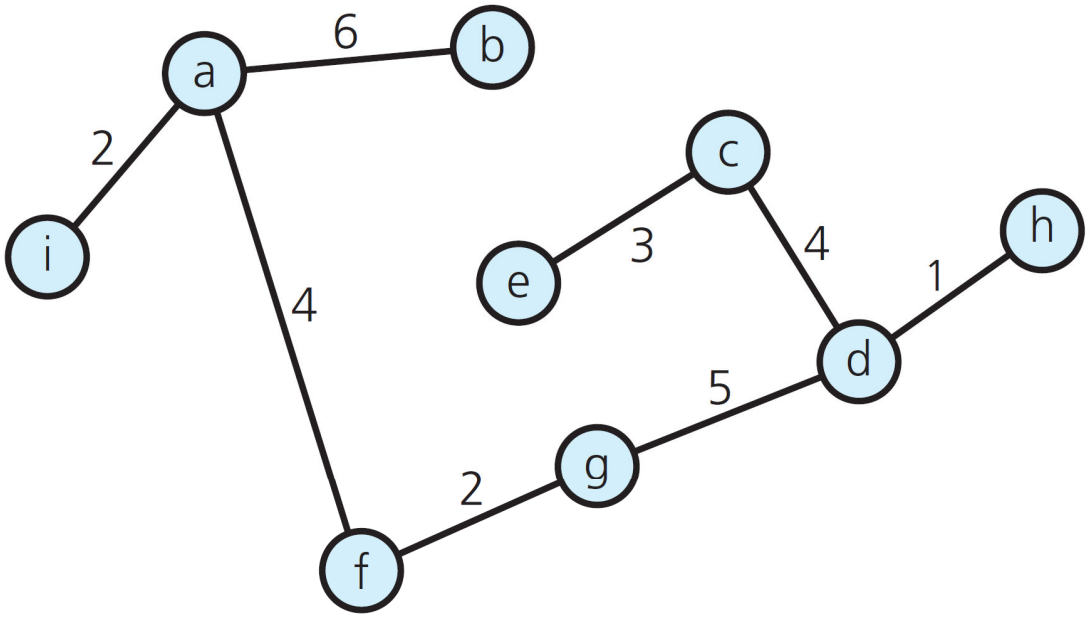
(g) Mark c, include edge (d, c)



(h) Mark e, include edge (c, e)

MST Prim Example (4 of 4)





(i) Mark b, include edge (a, b)

Tree cost = 27

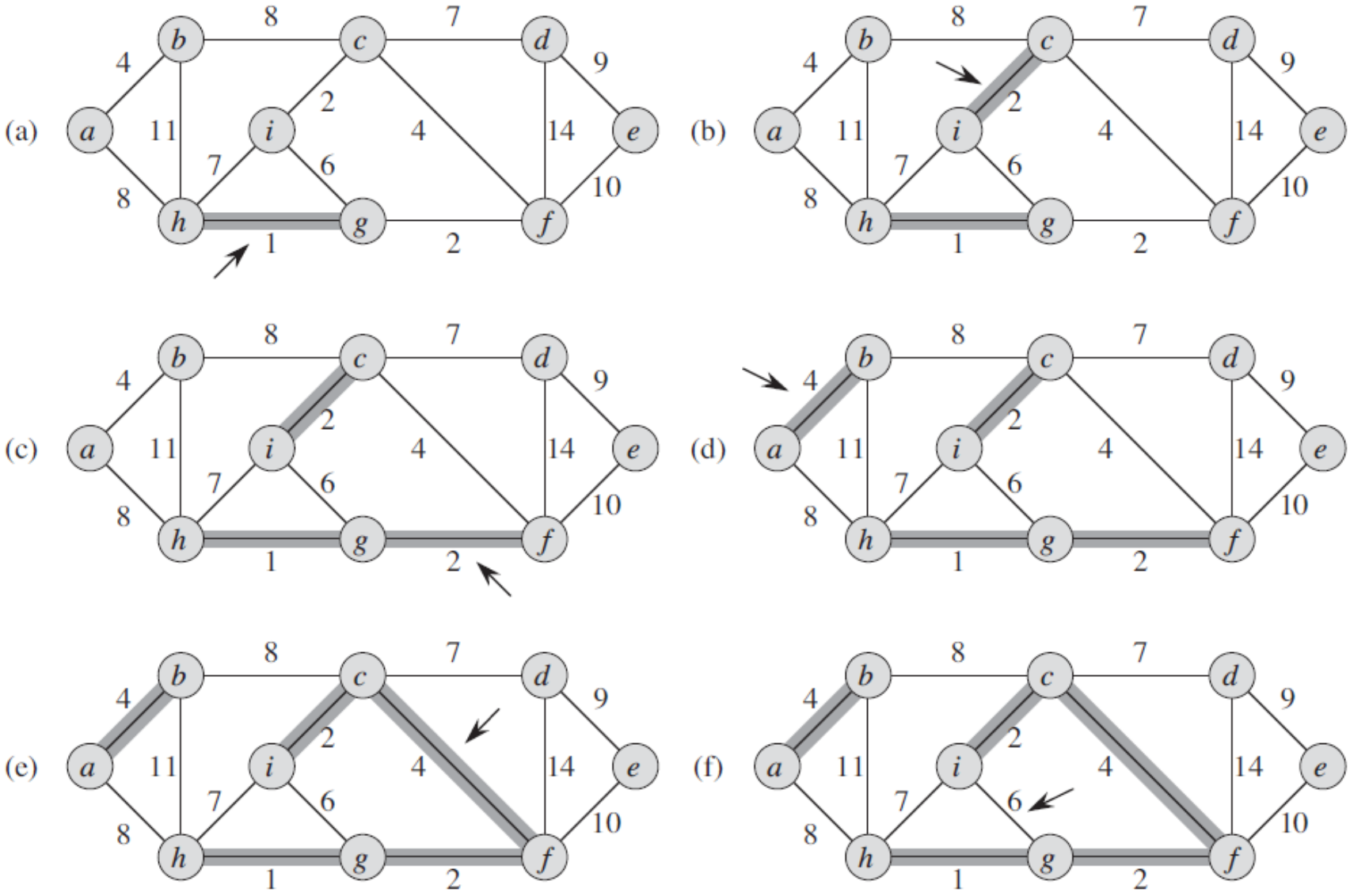
MST: Kruskal's Algorithm

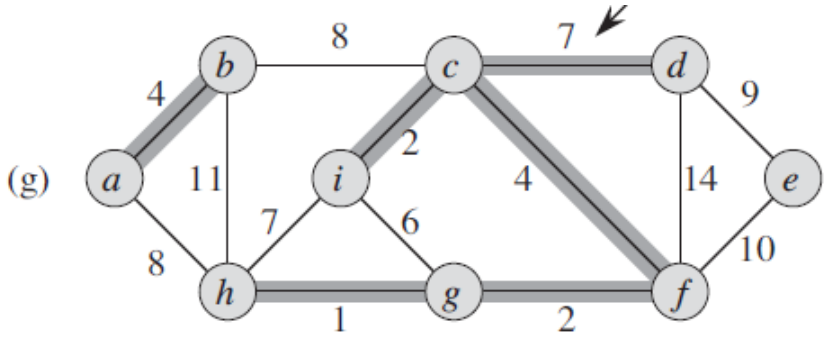
```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

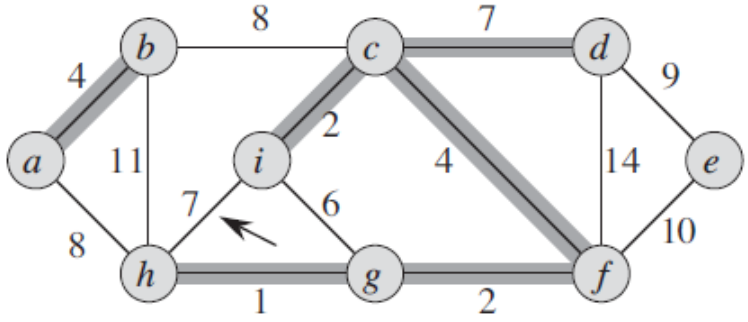
To combine two sets to form one tree

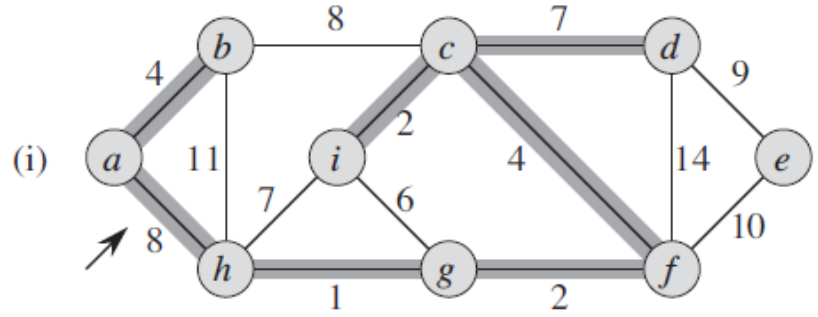
Comparing u's set with v's set to determine whether these vertices *u* and *v* belong to the same tree

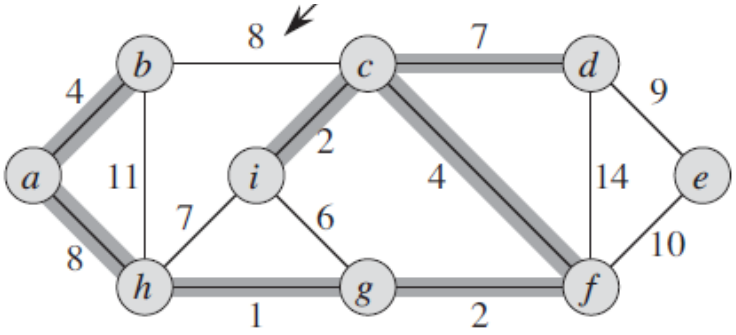
MST Kruskal Example

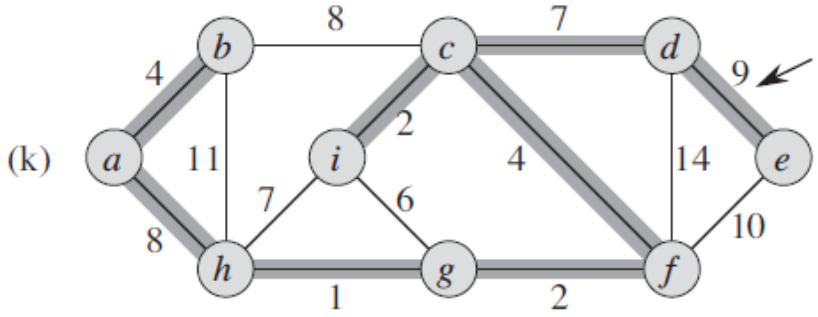


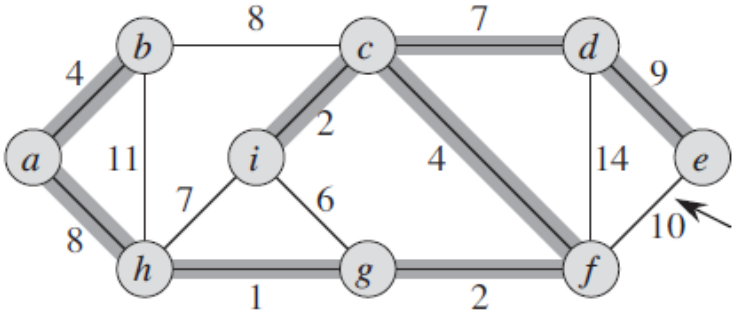

(g)
a b c d e f g h i
4 8 8 7 9 11 7 6 2 4 14 10 8 1 2

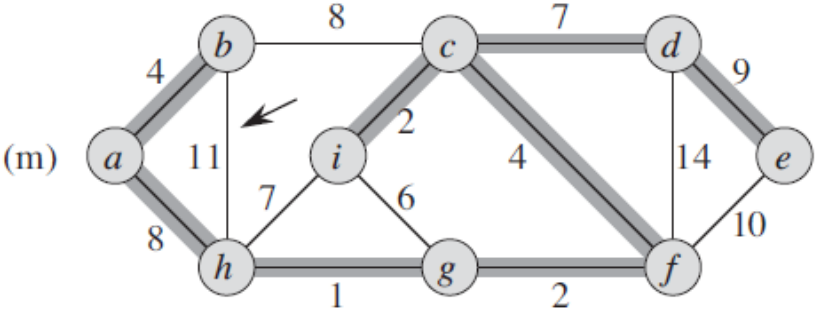

(h)
a b c d e f g h i
4 8 8 7 9 11 7 6 2 4 14 10 8 1 2

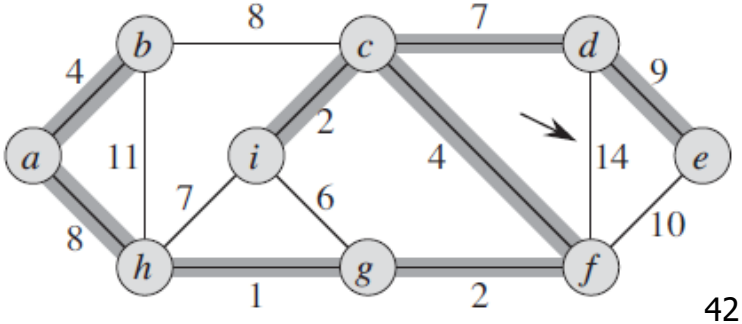

(i)
a b c d e f g h i
4 8 8 7 9 11 7 6 2 4 14 10 8 1 2


(j)
a b c d e f g h i
4 8 8 7 9 11 7 6 2 4 14 10 8 1 2


(k)
a b c d e f g h i
4 8 8 7 9 11 7 6 2 4 14 10 8 1 2


(l)
a b c d e f g h i
4 8 8 7 9 11 7 6 2 4 14 10 8 1 2


(m)
a b c d e f g h i
4 8 8 7 9 11 7 6 2 4 14 10 8 1 2


(n)
a b c d e f g h i
4 8 8 7 9 11 7 6 2 4 14 10 8 1 2

Greedy Algorithms

- Minimum-spanning-tree algorithms furnish a classic example of the greedy algorithms.
- A ***greedy algorithm*** always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.
- The greedy method is quite powerful and works well for a wide range of problems.

Shortest Paths

- The **shortest path** between two given vertices in a weighted graph is the path that has the smallest sum of its edge weights.
  - For each edge (*u*, *v*) in the weighted graph, we have a weight ***w*(*u*, *v*)** specifying the cost to connect *u* and *v*
  - The sum of the weights of the edges of a path is called the path's **length** or **weight** or **cost**
  - The shortest-path weight from *s* to *v* is defined as **δ(*s*, *v*)**
  - The weight of a vertex *v* is the weight of the current known path from the source to *v*, it is also called *v*'s shortest-path estimate, and it is denoted as **weight[*v*]** or ***v.d***
  - The vertex that precedes *v* on the current known path from the source to *v* is called v's predecessor and is denoted as **predecessor[v]** or ***v.π***

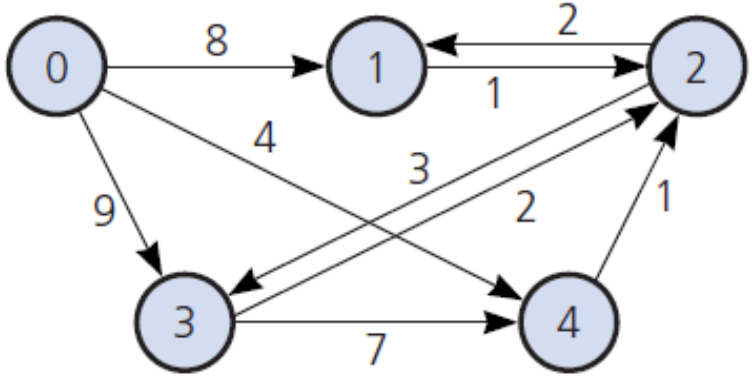
Shortest Paths Application Example

- Consider a map of airline routes. A weighted directed graph can represent this map: The vertices are cities, and the edges indicate existing flights between cities. The edge weights represent the mileage between cities (vertices); as such, the weights are not negative. A shortest paths algorithm can find the shortest routes between any two cities in the map.

Shortest Path Example

- In the following weighted directed graph, the shortest path from vertex 0 to vertex 1 is not the edge between 0 and 1—its cost is 8—but rather the path 0 → 4 → 2 → 1, with a cost of 7.

(a)



(b)

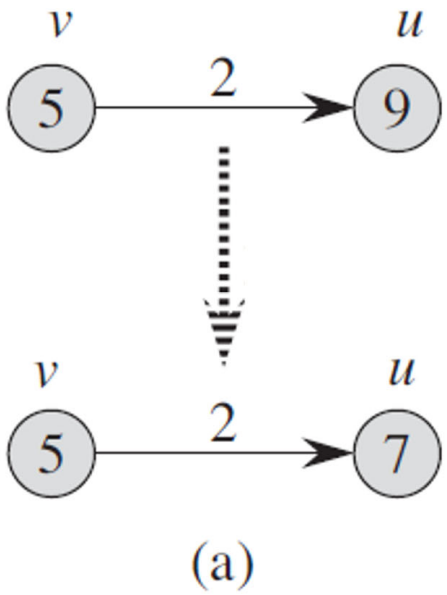
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Dijkstra's Shortest Path Algorithm

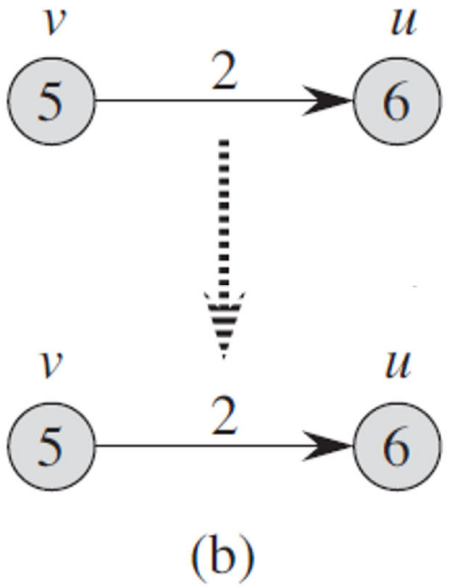
- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are **nonnegative**.
- Dijkstra's algorithm maintains a set `vertexSet` of vertices whose final shortest-path weights from the source *0* have already been determined.
- The algorithm repeatedly selects the vertex $u \in V$ - `vertexSet` with the minimum shortest-path estimate (weight[*u*]), adds *u* to `vertexSet`, and **relaxes** all edges leaving *u*.

Relaxation

- The process of ***relaxing*** an edge (v, u) consists of testing whether we can improve the shortest path to u found so far by going through v and, if so, updating `weight[u]`.
  - After relaxation, u's predecessor vertex ($u.\pi$) must be updated.
- Example:



(a)

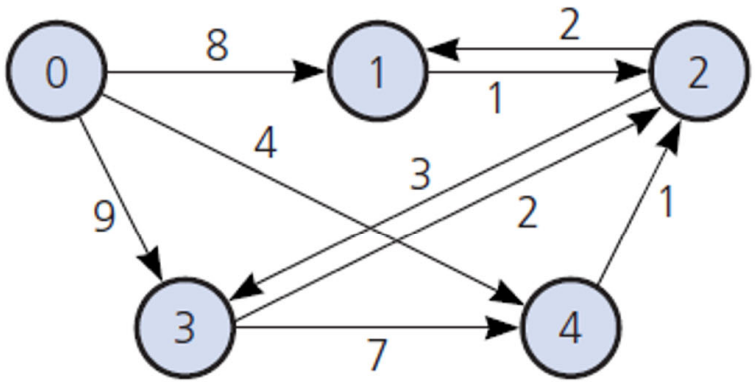


(b)

Dijkstra's Algorithm

```
// Finds the minimum-cost paths between an origin vertex
// (vertex 0) and all other vertices in a weighted directed
// graph theGraph; theGraph's weights are nonnegative.
shortestPath(theGraph: Graph, weight: WeightArray)
// initialization
 Create a set vertexSet that is empty
 n = number of vertices in theGraph
 for each vertex in theGrah {
   weight[v] = ∞
   predecessor[v] = -1 } // unknown predecessor
 weight[0] = 0
 for (step = 1 through n) {
  //A priority queue using heap can be used for the following step
   Find the smallest weight[v] such that v is not in vertexSet
   Add v to vertexSet
   // Relaxation: Check weight[u] for all u not in vertexSet
   for (all vertices u not in vertexSet)
      if (weight[u] > weight[v] + matrix[v][u]) {
          weight[u] = weight[v] + matrix[v][u]
          predecessor[u] = v }
}
```

Dijkstra's Algorithm Example



matrix

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

Properties of Shortest Paths

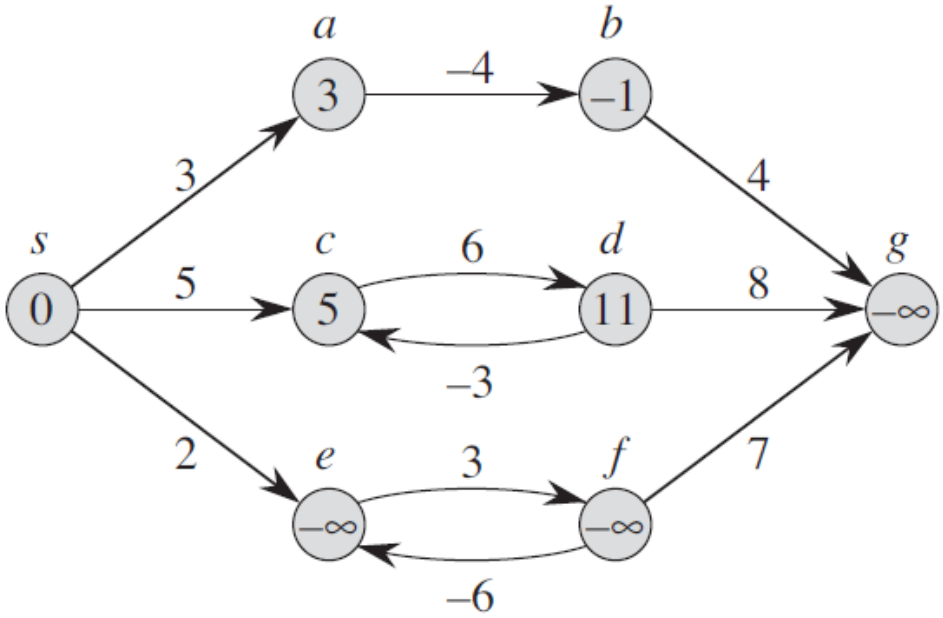
- **Triangle inequality**
  - For any edge $(u, v) \in E$, we have $\delta(s, v) \le \delta(s, u) + w(u, v)$.
- **Upper-bound property**
  - We always have weight[v] $\ge \delta(s, v)$ for all vertices $v \in V$, and once weight[v] achieves the value $\delta(s, v)$, it never changes.
- **No-path property**
  - If there is no path from s to v, then we always have weight[v] = $\delta(s, v) = \infty$.
- **Predecessor-subgraph property**
  - Once weight[v] = $\delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

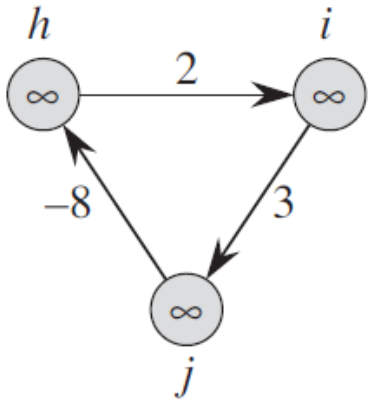
Negative-Weight Edges

- Some instances of the single-source shortest-paths problem may include edges whose weights are negative.
- If the graph $G = (V, E)$ contains no negative-weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value.
- If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined. No path from s to a vertex on the cycle can be a shortest path.
- If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

Negative-Weight Cycle Example

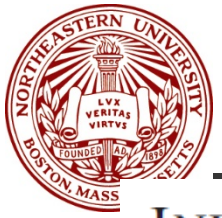
- In the following example, The shortest-path weight from source *s* appears within each vertex.
- Vertices *e*, *f* , and *g* are reachable from *s* through a negative-weight cycle (cycle *e*, *f*) ,they have shortest-path weights of $-\infty$.
- Vertices *h*, *i*, and *j* are not reachable from *s*, and so their shortest-path weights are ∞, even though they lie on a negative-weight cycle.





The Bellman-Ford Algorithm (1 of 3)

- The ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the general case in which edge weights may be negative.
- It returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source.
  - If there is such a cycle, the algorithm indicates that no solution exists.
  - If there is no such cycle, the algorithm produces the shortest paths and their weights.



The Bellman-Ford Algorithm (2 of 3)

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

Note: in this algorithm, *v.d* is weight[*v*]

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

The algorithm makes $|V|$-1 passes over the edges of the graph. Each pass consists of relaxing each edge of the graph once.

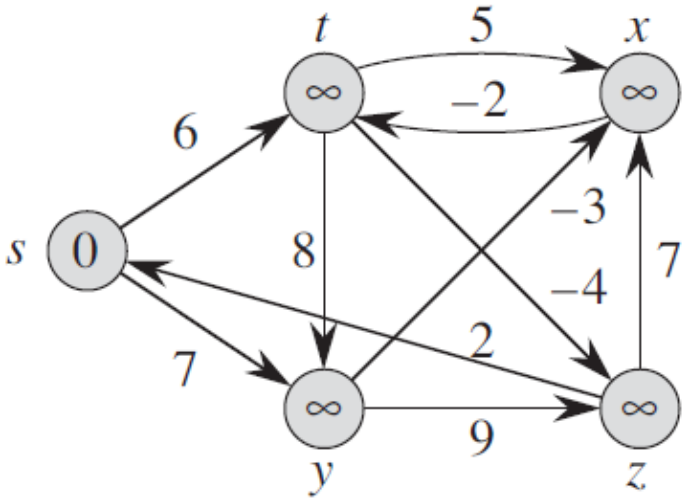
Triangle inequality

It checks for a negative-weight cycle and returns the appropriate Boolean value.

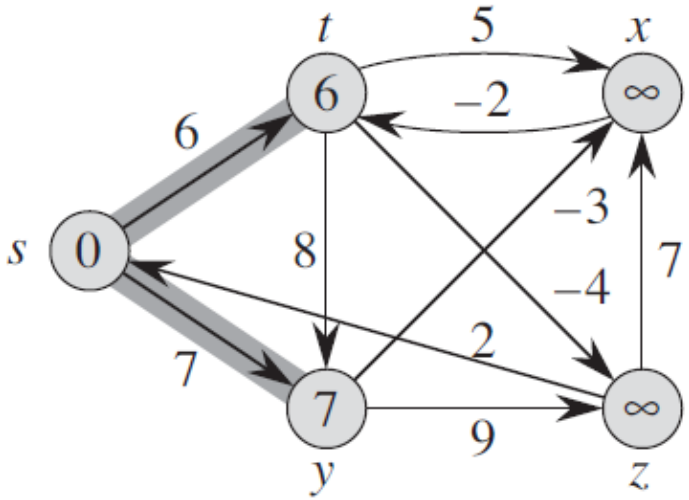
The Bellman-Ford Algorithm (3 of 3)

- The previous algorithm can be optimized by:
  - Maintaining an array *D* with entries *True* or *False* for whether the weight of each vertex has been changed or not (initially this array contain all *False* except a *True* entry for source node *s*)
  - Skipping the *Relax*(*u, v, w*) step if *u.d* has entry *False* in *D*.
  - After each call to *Relax*(*u, v, w*), update *v*'s entry in *D* to indicate whether *v*'s weight has been changed or not.
  - No need to continue the first *for* loop if all entries in *D* are false.
- This optimization of the algorithm uses the dynamic programming approach.

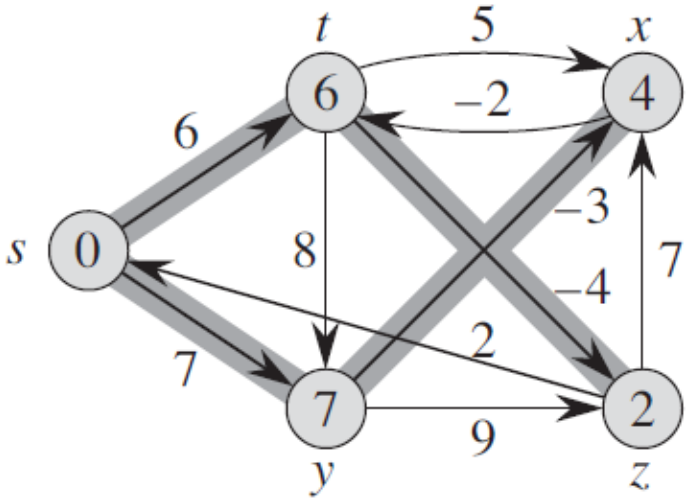
Bellman-Ford Example (1 of 2)



(a)

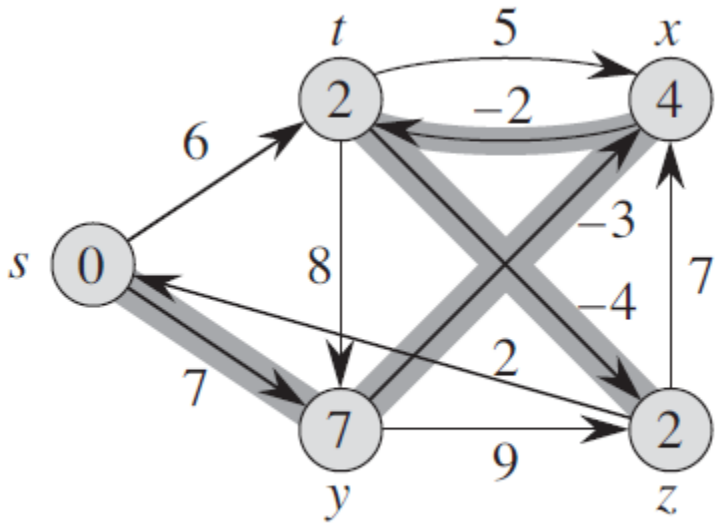


(b)

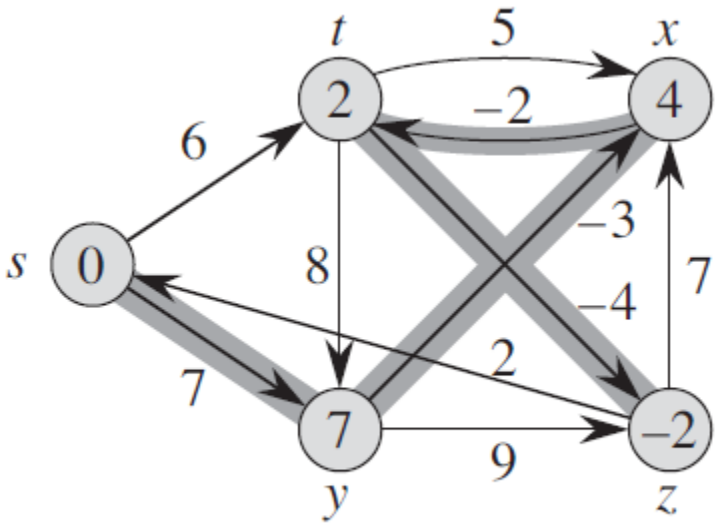


(c)

Bellman-Ford Example (2 of 2)



(d)



(e)

- As the triangle inequality ($\delta(s, v) \leq \delta(s, u) + w(u, v)$) can be verified on all edges, then the Bellman-Ford algorithm returns TRUE in this example.

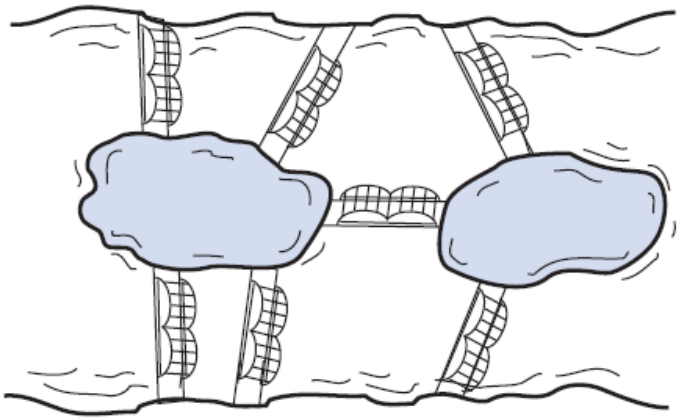
Circuits

- A **circuit** is another name for type of cycle common in statement of certain types of problems.
- A cycle in a graph is a path that begins and ends at the same vertex.
- Typical circuits either visit every vertex once or every edge once.

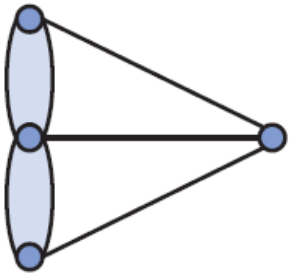
Euler Circuit (1 of 2)

- In 1700s Euler proposed a bridge problem.
- Two islands in a river are joined to each other and to the river banks by several bridges, as shown in (a)
- The bridges correspond to the edges in the multigraph in (b) and the land masses correspond to the vertices.
- The problem asked whether you can begin at a vertex *v,* pass through every edge exactly once, and terminate at *v* .
- Euler demonstrated that no solution exists for this particular configuration of edges and vertices.

(a)



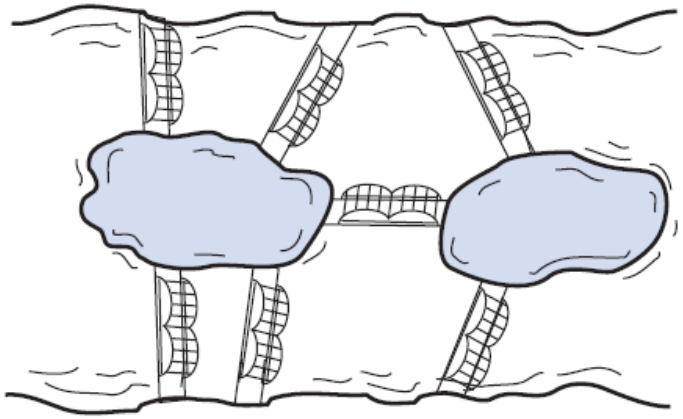
(b)



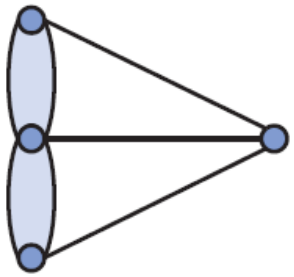
Euler Circuit (2 of 2)

- A path in an undirected graph that begins at a vertex *v,* passes through every edge in the graph exactly once, and terminates at *v* is called an **Euler circuit**.
  - It can be used to check for the graph's connectivity.
- Euler showed that an Euler circuit exists if and only if each vertex touches an even number of edges.

(a)



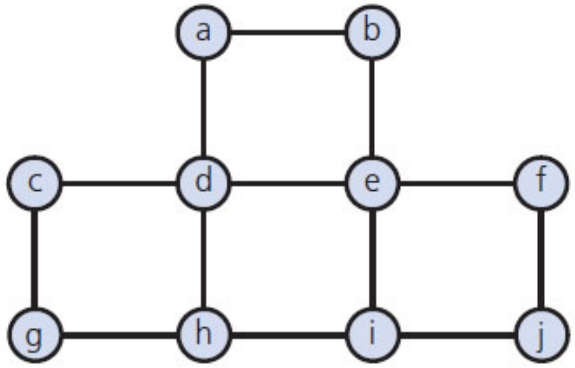
(b)



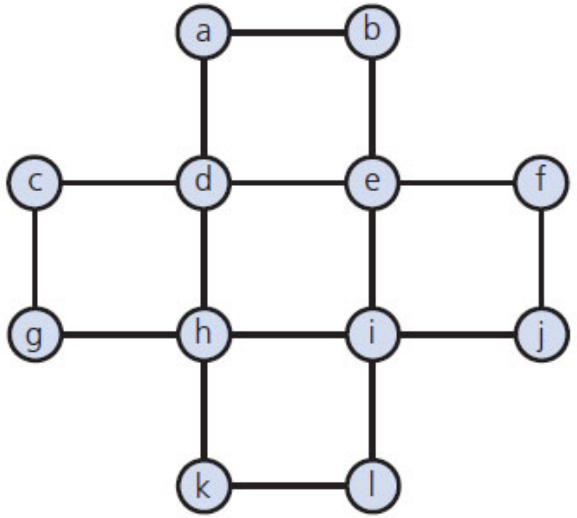
Euler Circuit Example (1 of 2)

- Finding an Euler circuit is like drawing the following diagrams without lifting your pencil or redrawing a line, and ending at your starting point.
- For diagram (a), vertices *h* and *i* each touch an odd number of edges (three), so no Euler circuit is possible.
- On the other hand, each vertex in diagram (b) touches an even number of edges, making an Euler circuit feasible.

(a)



(b)

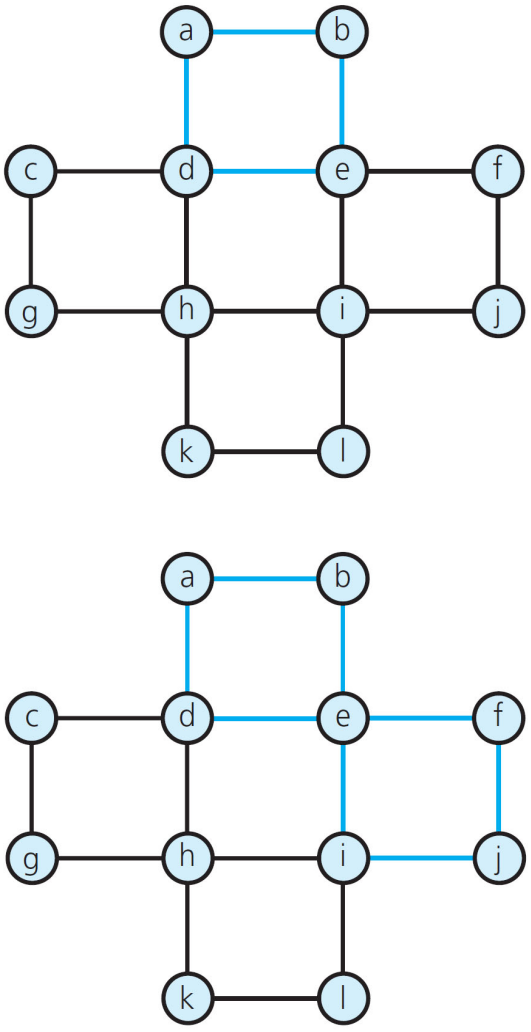


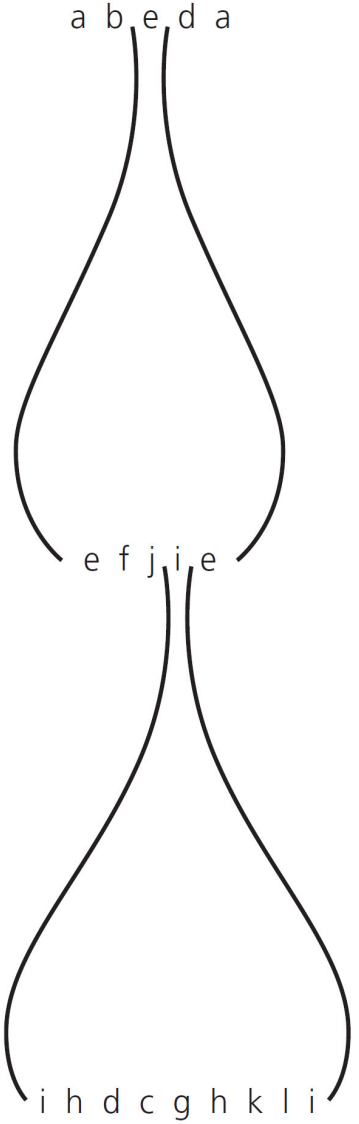
Euler Circuit Example (2 of 2)

1. The strategy is to use a depth-first search that marks edges instead of vertices.
2. Stop when you have a cycle. To continue, find the first vertex along the cycle that touches an unvisited edge.
3. Repeat applying depth-first search from that vertex and repeat step 2.

- Euler circuit is:

  a b e f j i h d
  c g h k l i e d a



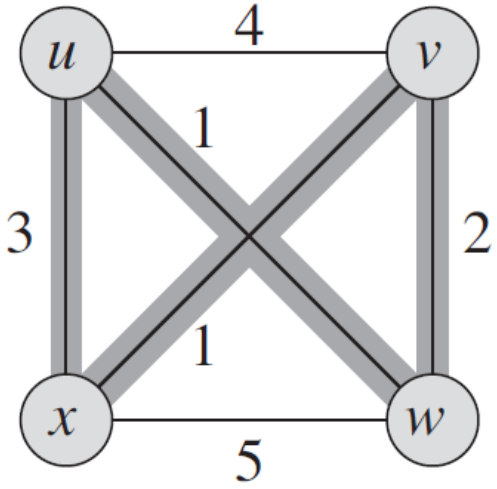


Hamilton Circuit

- **Hamilton circuit**
  - Begins at vertex *v*
  - Passes through every vertex exactly once
  - Terminates at *v*
- Variation is "**traveling salesperson problem**"
  - Visit every city on his/her route exactly once
  - Edge (road) has associated cost (mileage)
  - Goal is determine least expensive circuit

The Traveling-Salesperson Problem

- The salesperson wishes to make a ***tour*** visiting each city exactly once and finishing at the city she/he starts from.
- The salesperson incurs a nonnegative integer cost $c(i, j)$ to travel from city i to city j , and the salesperson wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.
- In the figure, a minimum-cost tour is (u, w, v, x, u), with cost 7.
- ***Theorem:***
  The traveling-salesperson problem (**TSP**) is NP-complete



Polynomial-Time Algorithms

- Almost all the algorithms we have studied thus far have been ***polynomial-time algorithms***: on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k.
- Not *all* problems can be solved in polynomial time.
  - Problems that are solvable by polynomial-time algorithms are tractable.
  - Problems that require superpolynomial time are intractable and they cannot be solved in a reasonable amount of time unless for very small n.
  - Problems solvable in logarithmic time are solvable in polynomial time as well.

NP-Complete Problems

- The status of the "Nondeterministic Polynomial time" or "NP-complete" problems is unknown.
  - No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them.

- The *Hamilton Circuit* problem is NP-complete while the *Euler Circuit* can be solved in $O(E)$ time.

Approximation Algorithms

- Many problems of practical significance are NP-complete, yet they are too important to abandon merely because we don't know how to find an optimal solution in polynomial time.
- There are at least three ways to get around NP-completeness:
  1. If the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
  2. We may be able to isolate important special cases that we can solve in polynomial time.
  3. We might come up with approaches to find near-optimal solutions in polynomial time. In practice, near optimality is often good enough.
- We call an algorithm that returns near-optimal solutions an ***approximation algorithm***.

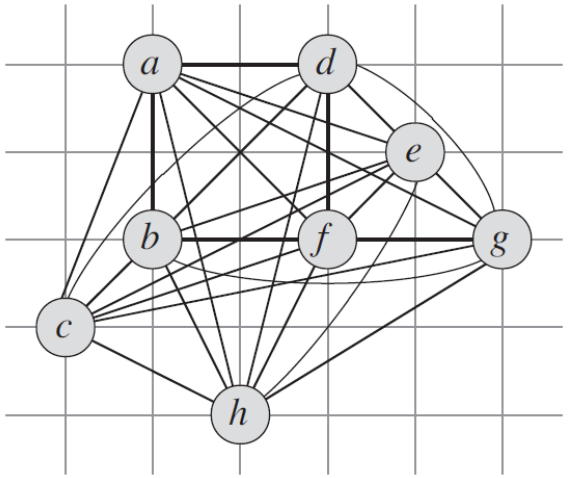
The TSP Approximation

- In the traveling-salesperson problem, we are given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, and we need to find a hamiltonian cycle (a tour) of G with minimum cost.
- The following is a TSP approximation algorithm:

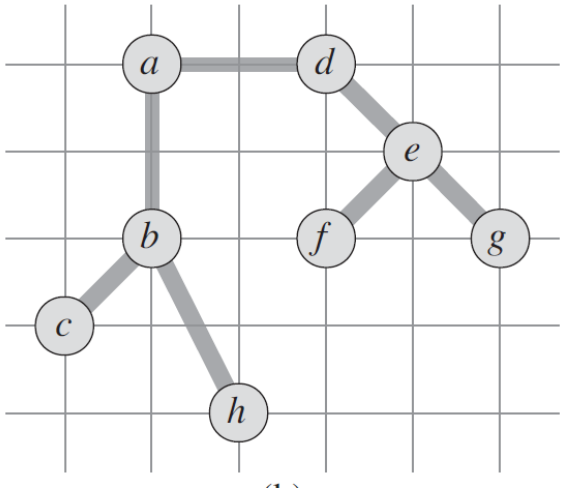
```
APPROX-TSP-TOUR(G, c)
1  select a vertex r ∈ G.V to be a "root" vertex
2  compute a minimum spanning tree T for G from root r
       using MST-PRIM(G, c, r)
3  let H be a list of vertices, ordered according to when they are first visited
       in a preorder tree walk of T
4  return the hamiltonian cycle H
```

TSP Example (1 of 3)

- A complete undirected graph. Vertices lie on intersections of integer grid lines.



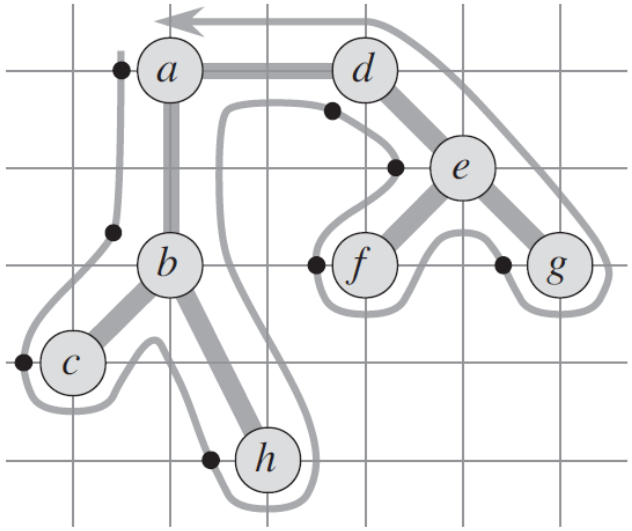
- A minimum spanning tree *T* of the complete graph, as computed by MST-PRIM. Vertex *a* is the root vertex.
- The weight of this MST gives a lower bound on the length of an optimal traveling-salesperson tour.

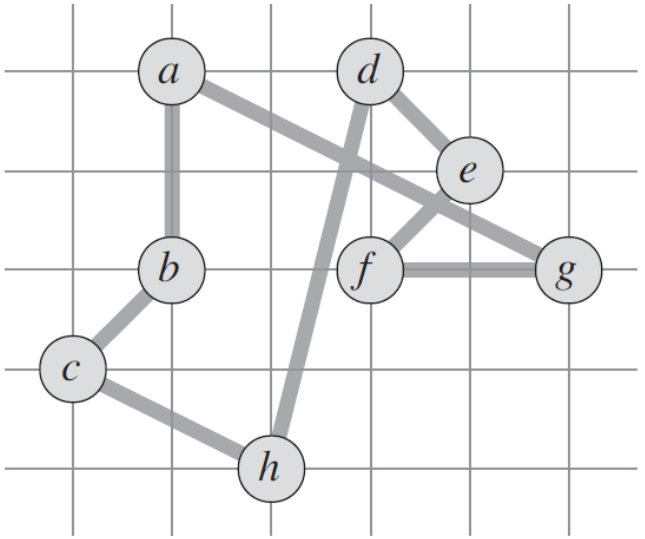


TSP Example (2 of 3)

- A **preorder** traversal of *T* is indicated by the dot next to each vertex, yielding the ordering *a,b,c,h,d,e,f,g*.

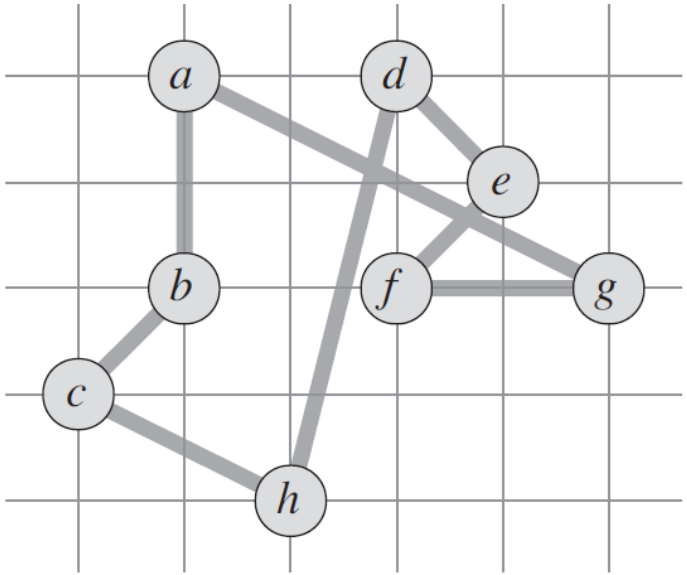
- A tour obtained by visiting the vertices in the order given by the preorder traversal, which is the tour *H* returned by APPROX-TSP-TOUR. Its total cost is approximately 19.



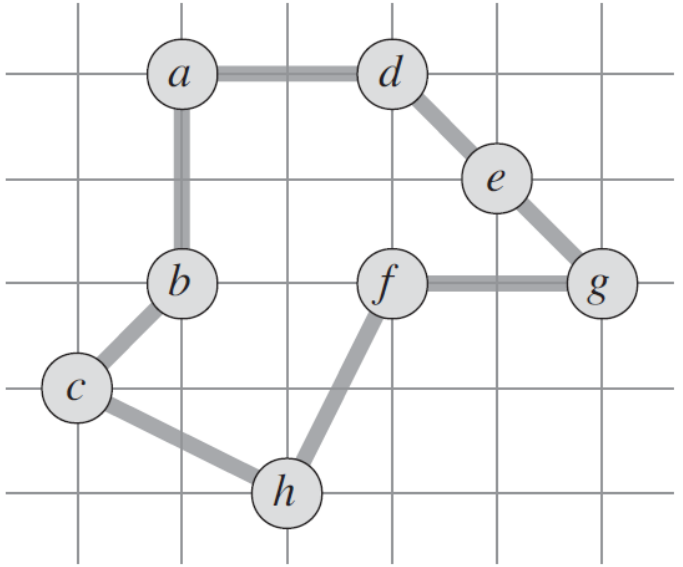


TSP Example (3 of 3)

- An optimal tour H^* for the original complete graph. Its total cost is approximately 14.7, which is about 23% shorter than the approximate solution.



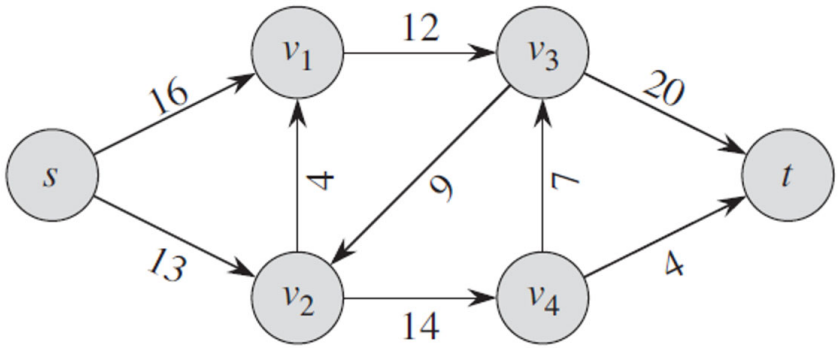
Cost ≈ 19

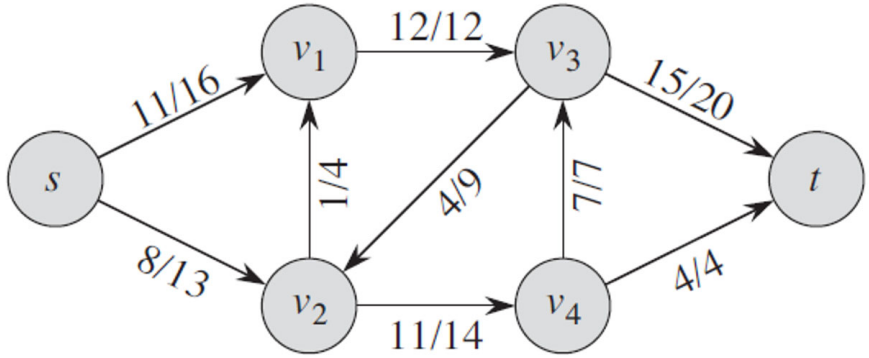


Cost ≈ 14.7

Maximum-Flow Problem

- A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v)$.
- Two vertices are distinguished in a flow network: a ***source s*** and a ***sink t***.
- In the ***maximum-flow problem***, we are given a flow network G with source s and sink t , and we wish to find a flow f of maximum value that meets the condition: for any vertex, except s and t, **flow in = flow out**.
- Flow networks can model many problems, including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks.





Net flow = 19

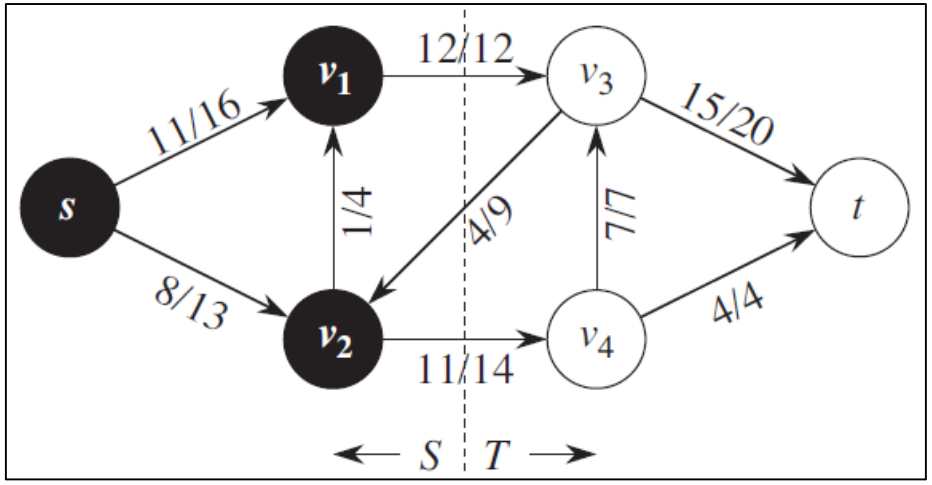
Cuts of Flow Networks

- A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and T *where* $T = V - S$ such that $s \in S$ and $t \in T$.
- If f is a flow, then the ***net flow*** $f(S, T)$ across the cut (S, T) is defined to be:

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) .$$

- The ***capacity*** of the cut is

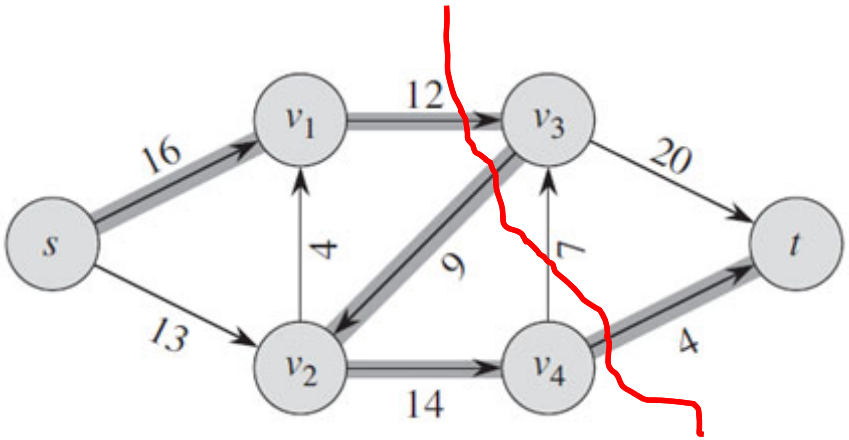
$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) .$$

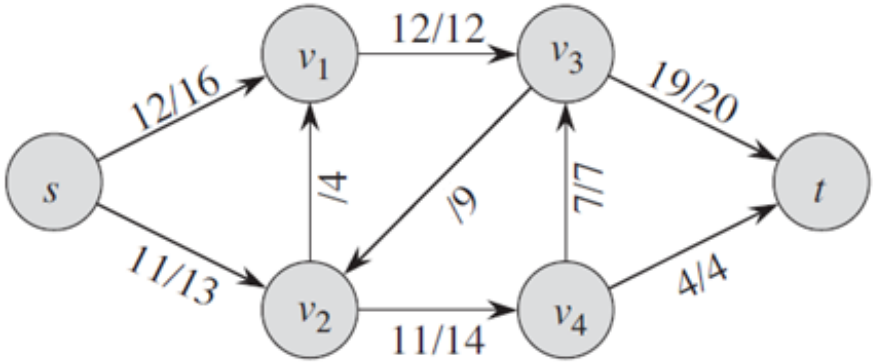


- For the shown graph $f(S, T) = 19$ and $c(S, T) = 26$
- *Notes*:
  - ***capacity*** counts only the capacities on edges going from S to T.
  - ***net flow*** considers flows in edges on both directions and it is the same for all cuts.

Minimum Cut of a Flow Network

- A ***minimum cut*** of a network is a cut whose ***capacity*** is minimum over all cuts of the network.
- The value of a maximum flow in a network is bounded from above by the capacity of a minimum cut of the network.



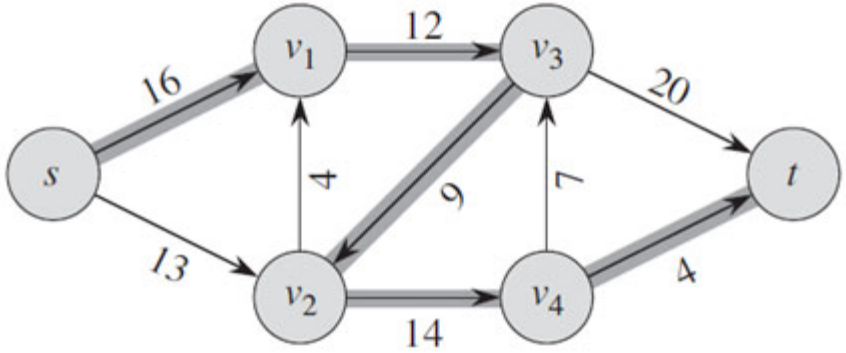


Net flow = 23

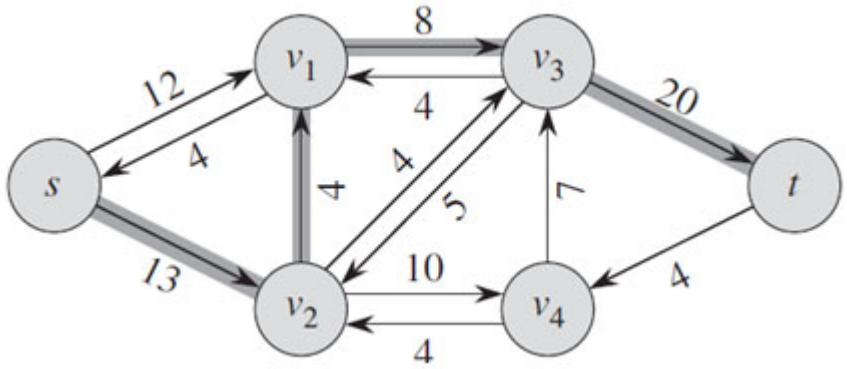
Ford-Fulkerson Algorithm

- The Ford-Fulkerson method creates, from the flow network *G, a* residual network G_f .
- It repeatedly finds an ***augmenting path*** *p*, which is a simple path from *s* to *t* in G_f and augments the path's residual capacity c_f in the edges along this path.
- When no augmenting paths exist, the final flow *f* from G_f is a maximum flow.

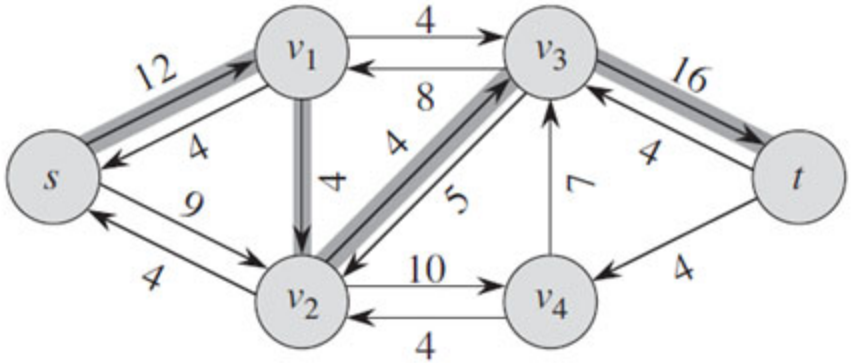
Ford-Fulkerson Example (1 of 2)



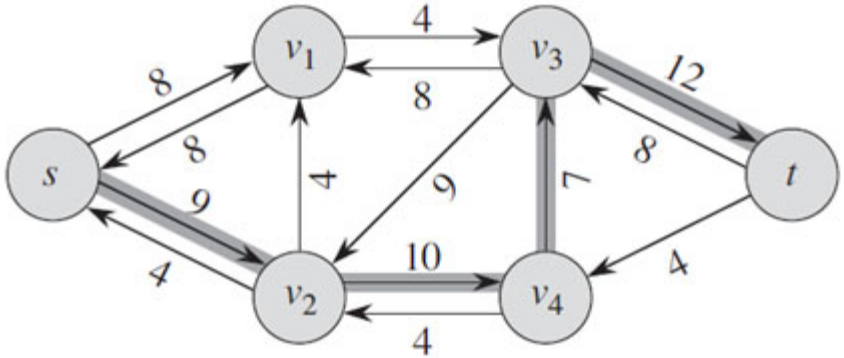
Input G_f with augmenting path p shaded and $c_f = 4$



Updated G_f with augmenting path p shaded and $c_f = 4$

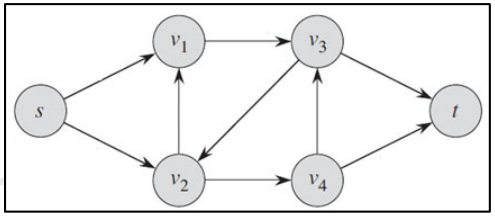


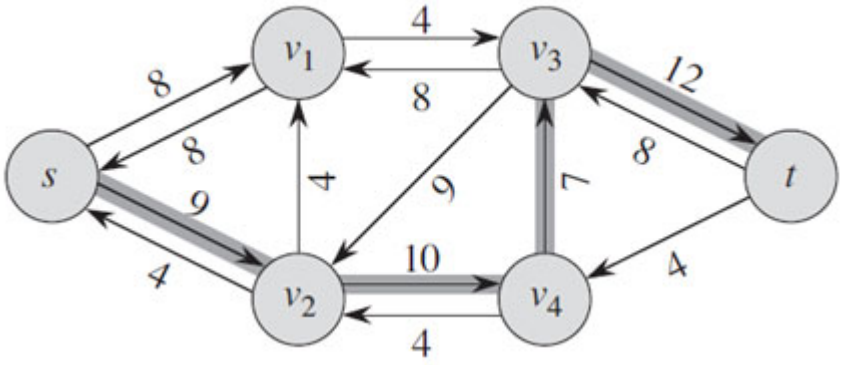
Updated G_f with augmenting path p shaded and $c_f = 4$



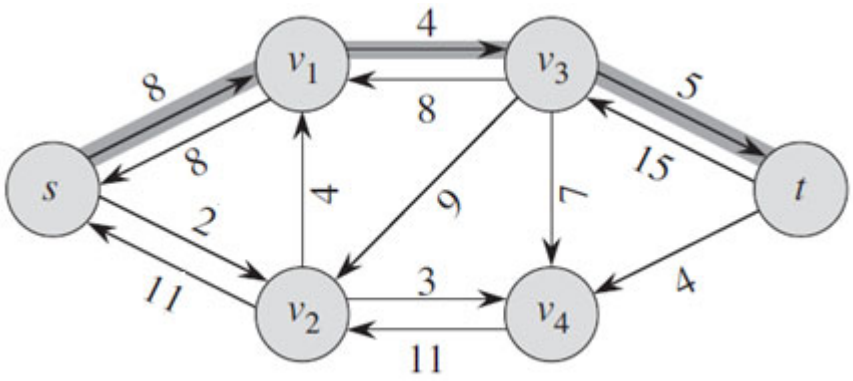
Updated G_f with augmenting path p shaded and $c_f = 7$

Example (2 of 2)

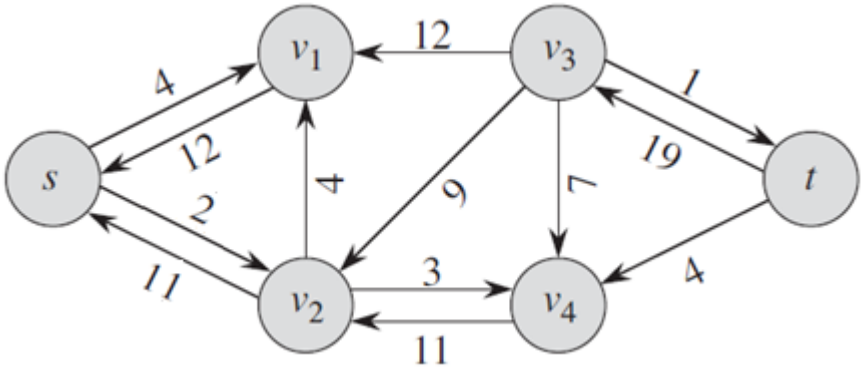




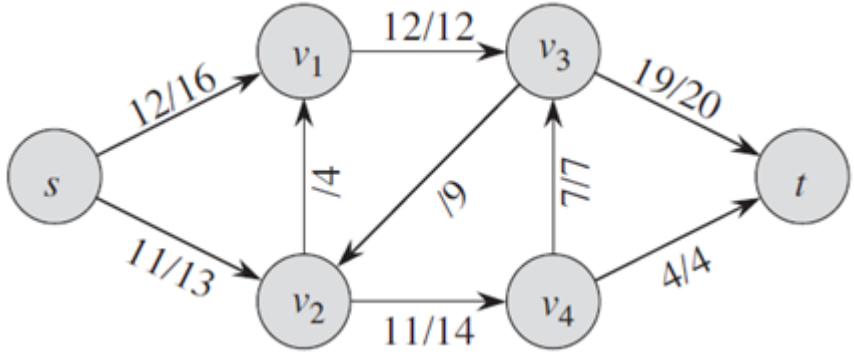
(Repeated)



Updated G_f with augmenting path p shaded and $c_f = 4$



Updated G_f with no augmenting paths exist



Updated G with the maximum flow $f = 23$