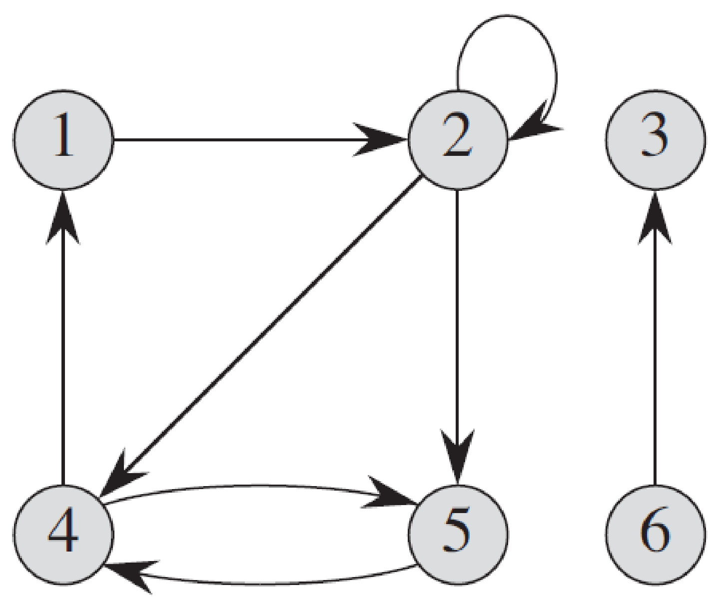
EEECE7205: Fundamentals of Computer Engineering

Graph Algorithms

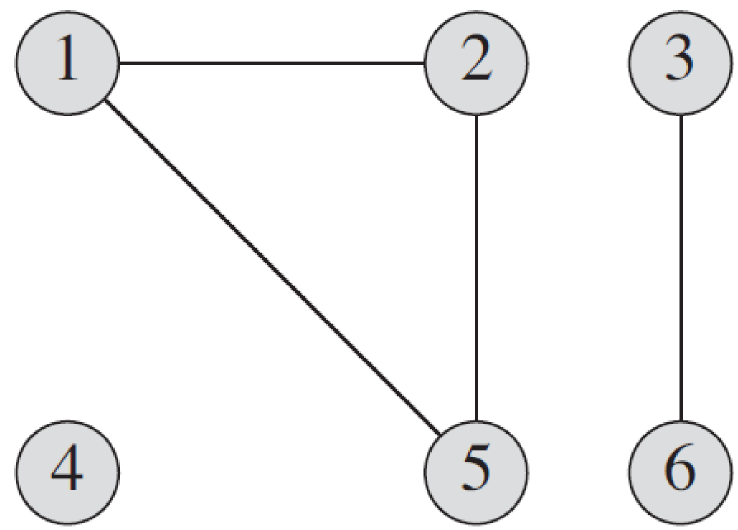
Directed Graph

- A ***directed graph*** (or ***digraph***) G is a pair (V, E), where V is a finite set and E is a binary relation on V.
- The set V is called the ***vertex set*** of G, and its elements are called ***vertices*** (singular: ***vertex***).
- The set E is called the ***edge set*** of G, and its elements are called ***edges***.
- Vertices are represented by circles, and edges are represented by arrows.
- In the shown example:
  $V = \{1, 2, 3, 4, 5, 6\}$
  $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$.
- The edge (2, 2) is a self-loop.



Undirected Graph

- In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices, rather than ordered pairs.
- An edge is a set $\{u, v\}$ where $u, v \in V$ and $u \neq v$.
- In the shown example:
  $V = \{1, 2, 3, 4, 5, 6\}$
  $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$.
- In an undirected graph, **self-loops** and **multiple edges between the same pair** of nodes are not allowed.



Graph Definitions (1 of 3)

- In a directed graph, an edge (u, v) is ***incident from*** or ***leaves*** vertex u and is ***incident to*** or ***enters*** vertex v.
- In an undirected graph, an edge (u, v) is ***incident on*** vertices u and v.
- In a graph with an edge (u, v) means that vertex v is ***adjacent*** to vertex u. In an undirected graph, the adjacency relation is symmetric.
- The ***degree*** of a vertex in an undirected graph is the number of edges incident on it. A vertex with degree 0 is called ***isolated***.
- In a directed graph, the ***out-degree*** of a vertex is the number of edges leaving it, and the ***in-degree*** of a vertex is the number of edges entering it.
  - The ***degree*** of a vertex in a directed graph is its in-degree plus its out-degree.

Graph Definitions (2 of 3)

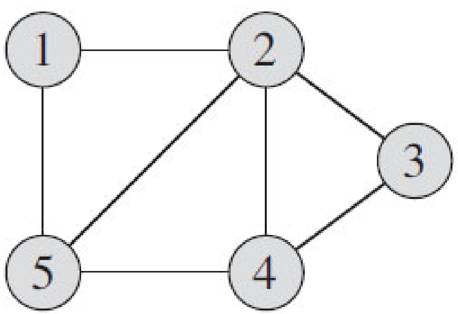
- A ***path*** of ***length*** k from a vertex v_0 to a vertex v_k in a graph is the sequence of vertices $(v_0, v_1, v_2, \ldots , v_k)$
  where $(v_{i-1}, v_i) \in E$ for $i = 1,2, \ldots ,k$.
- The length of the path is the number of edges in the path.
- If there is a path p from u to v, we say that v is ***reachable*** from u via p.
- A path is ***simple*** if all vertices in the path are distinct.
- A ***subpath*** of path $p = (v_0, v_1, v_2, \ldots , v_k)$ is a contiguous subsequence of its vertices.
- In a directed graph, a path $(v_0, v_1, v_2, \ldots , v_k)$ forms a ***cycle*** if $v_0 = v_k$ and it is a ***simple cycle*** if all its other vertices are distinct.
- A self-loop is a cycle of length 1. A directed graph with no self-loops is ***simple***.
- In an undirected graph, a path $(v_0, v_1, v_2, \ldots , v_k)$ forms a ***cycle*** if $k \geq 3$ and $v_0 = v_k$ and it is a ***simple cycle*** if all its vertices are distinct.
- A graph with no cycles is ***acyclic***.

Graph Definitions (3 of 3)

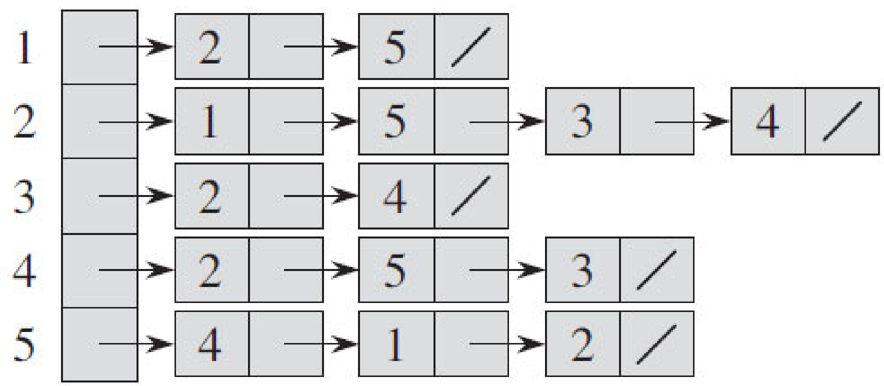
- An undirected graph is ***connected*** if every vertex is reachable from all other vertices.
- A directed graph is ***strongly connected*** if every two vertices are reachable from each other.
- A ***complete graph*** is an undirected graph in which every pair of vertices is adjacent.
- An acyclic, undirected graph is a ***forest***.
- A connected, acyclic, undirected graph is a ***tree***.
- 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 $|V|^2$.

Representing Undirected Graphs

- The adjacency-list representation is usually the method of choice for ***sparse*** graphs.
- The adjacency-matrix representation is usually the method of choice for ***dense*** graphs or when we need to be able to tell quickly if there is an edge connecting two given vertices.
- The sum of the lengths of all the adjacency lists is $2|E|$
- Observe the symmetry along the main diagonal of the adjacency matrix.



An undirected graph G



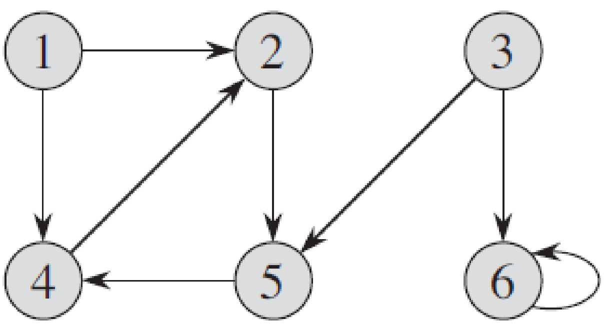
An adjacency-list representation of G

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

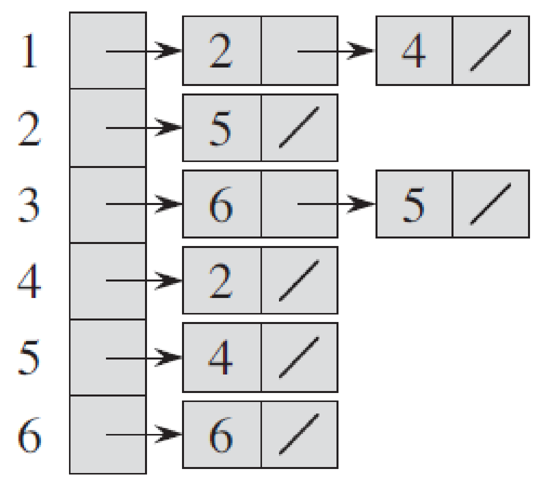
An adjacency-matrix representation of G

Representing Directed Graphs

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



A directed graph G



An adjacency-list representation of G

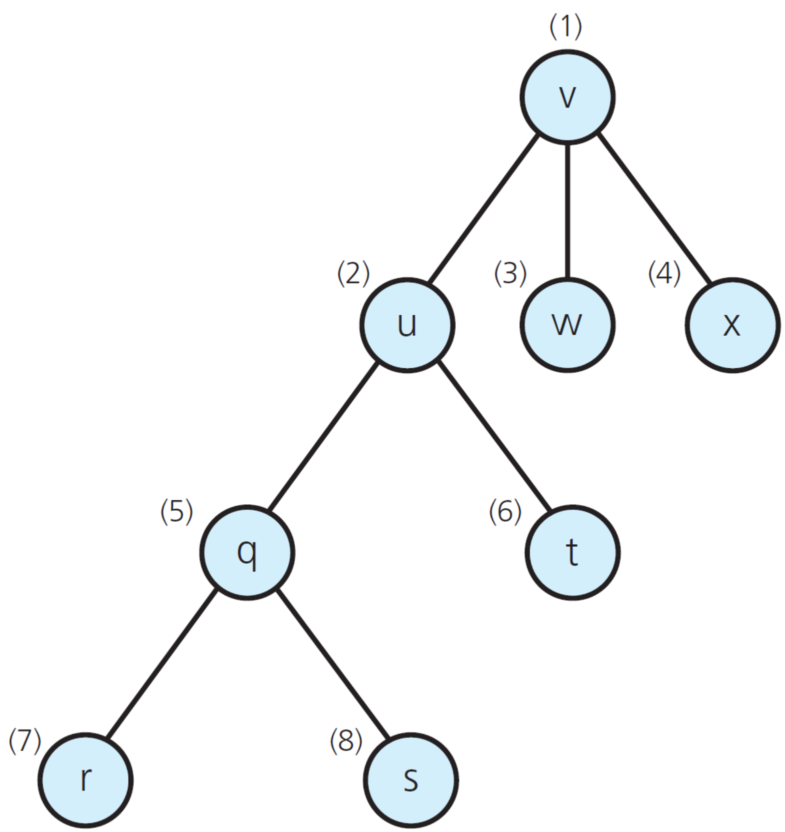
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

An adjacency-matrix representation of G

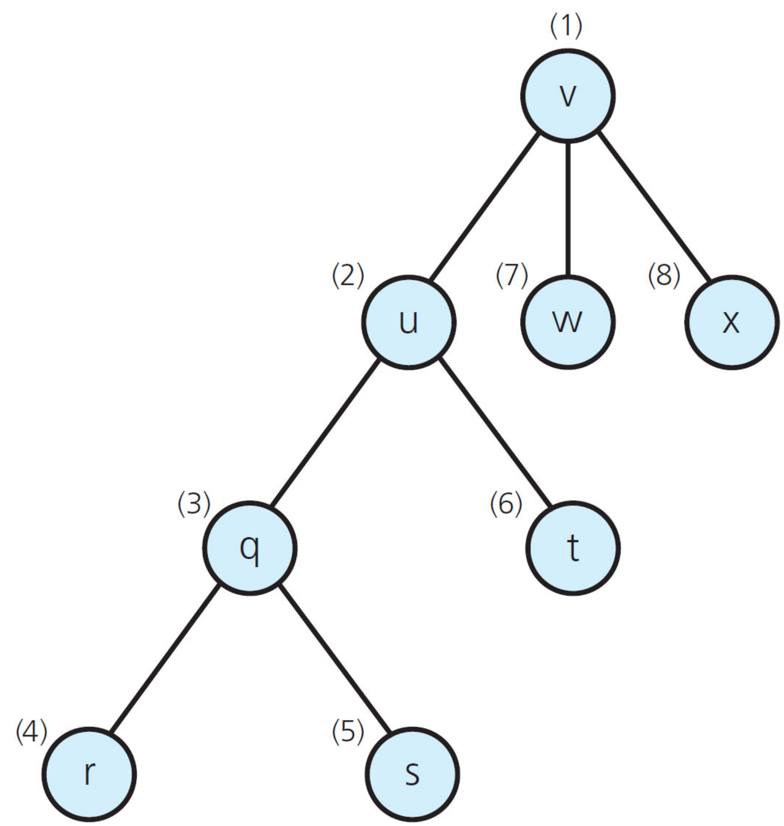
Graph Traversals

- 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

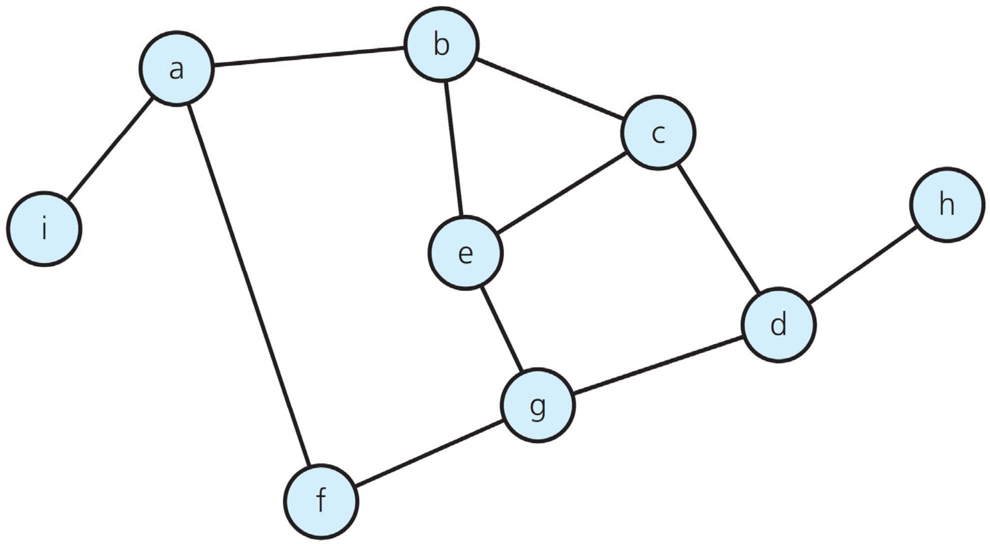
Breadth-First Search (1 of 2)

- BFS strategy: discovers all vertices at distance *k* from *s* before discovering any vertices at distance *k* + 1
- This strategy is reflected in using a **queue** as in the following algorithm:

```
bfs(v: Vertex)
 q = 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)

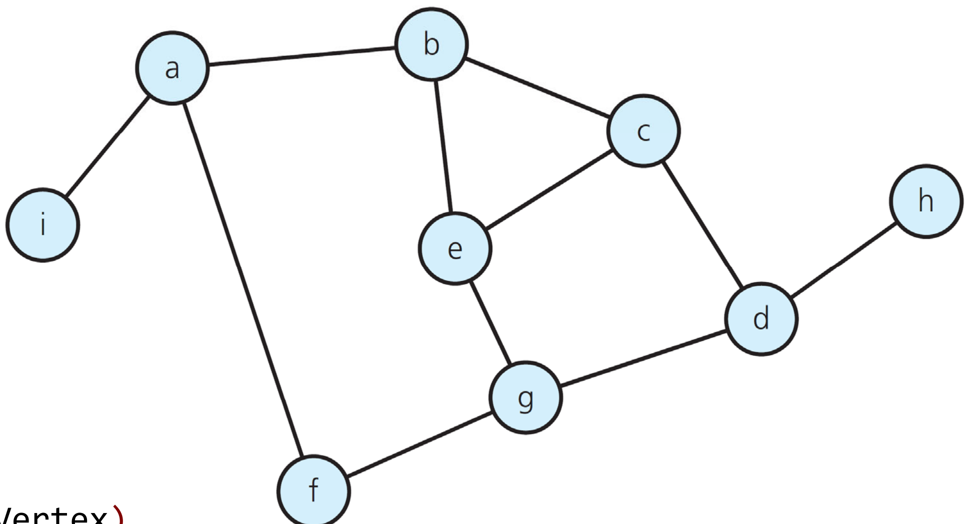
Depth-First Search (1 of 2)

- DFS strategy: searches “deeper” in the graph whenever possible before backing up.
- 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)

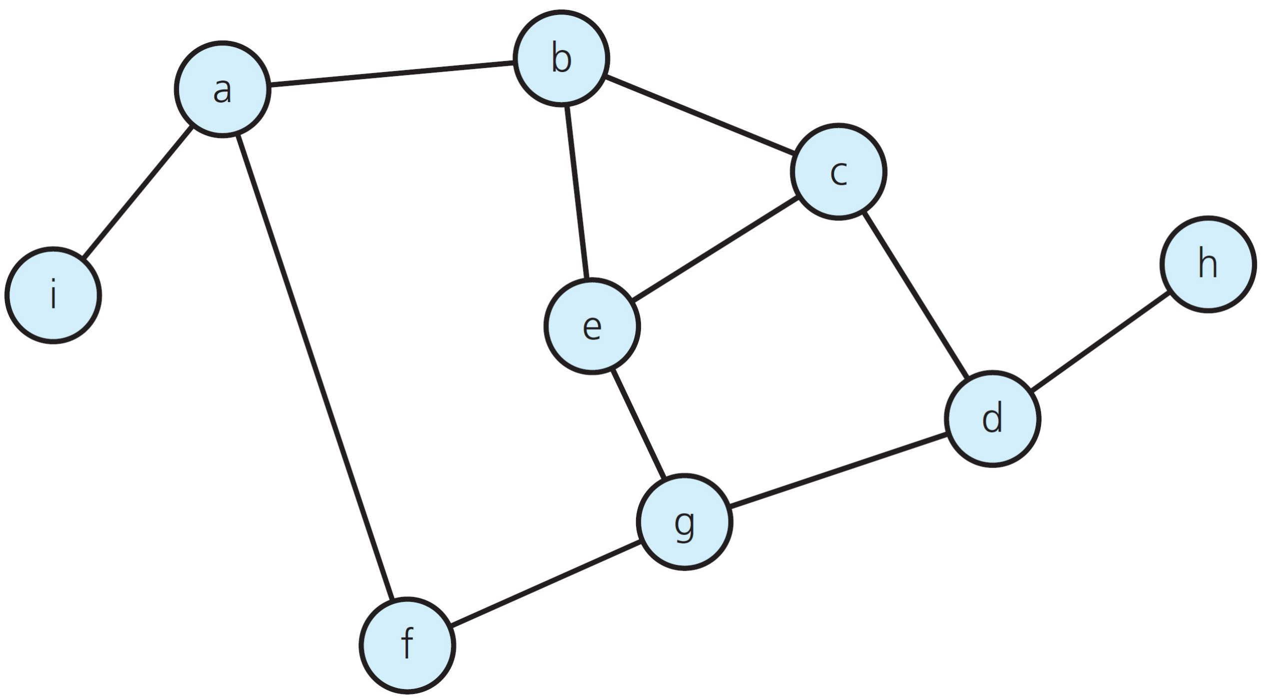
- Recall that a connected undirected graph cannot contain a cycle if it has *n* vertices and exactly n – 1 edges,.
- The following is a modified DFS algorithm to determine whether a general graph contains a cycle.
- In this algorithm, each vertex has the following attributes:
  - **`visited`**: to indicate 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
              u.predecessor = top
              s.push(u)                         }
         else { top.finished=true
                 s.pop()}
 }
 return false
}
```

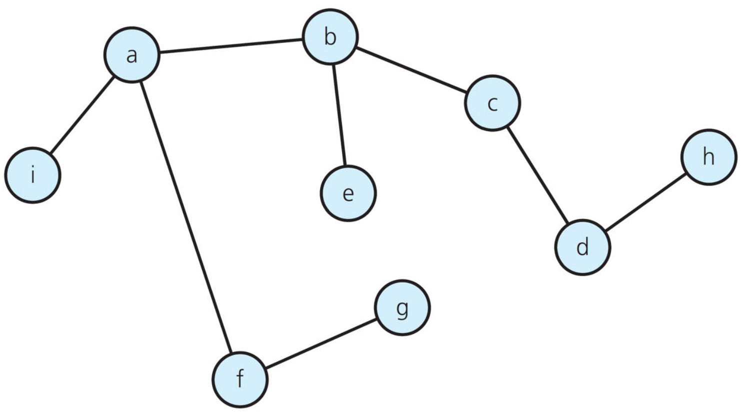
Detect Cycles in a Graph (3 of 3)

- 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` and is not e's predecessor (vertic *b*), then the algorithm will return `true`.

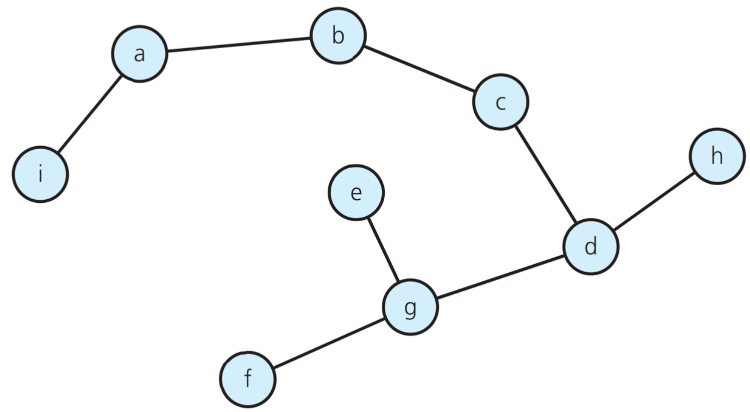


BFS and DFS Trees

- The predecessor of each vertex in the BFS and DFS can be maintained as explained in the detect cycle algorithm.
- The BFS and DFS predecessor subgraph forms a tree as shown.



BFS Predecessor Subgraph
Tree rooted at vertex *a*

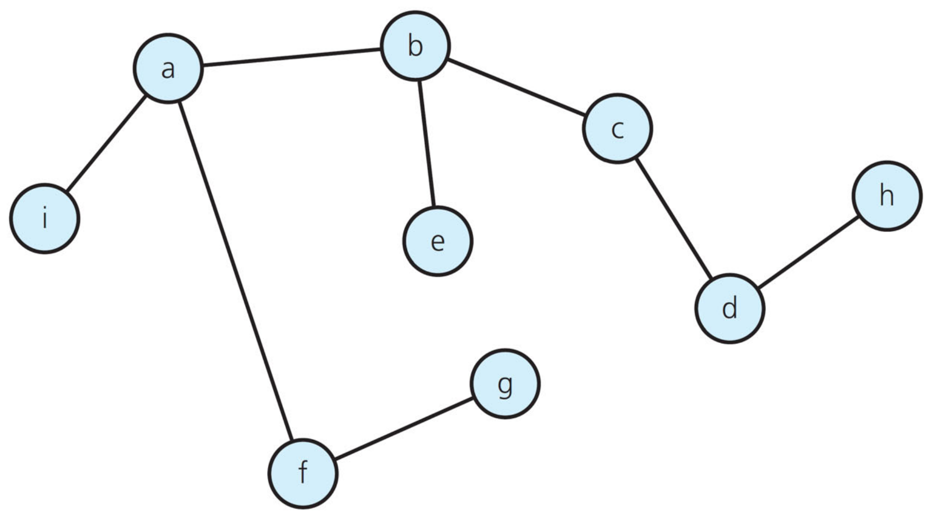


DFS Predecessor Subgraph
Tree rooted at vertex *a*

Printing BFS and DFS Paths

- The following procedure prints out the vertices on a path from vertex s to v, on the BFS and DFS predecessor subgraph .
- Assume all predecessors are initialized to NIL

```
PrintPath(G: Graph, s: Vertex, v: Vertex)
 if v == s
       print s
 else if v.predecessor == NIL
              print no path from s to v exists
       else PrintPath(G, s, v.predecessor)
            print v
}
```



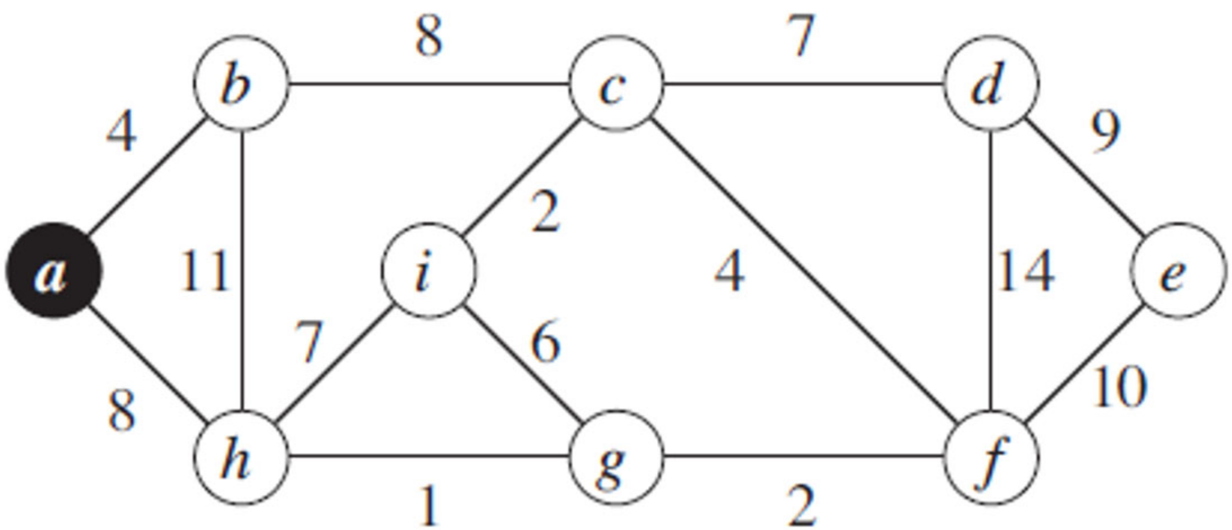
Spanning Trees

- Application example:

  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. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

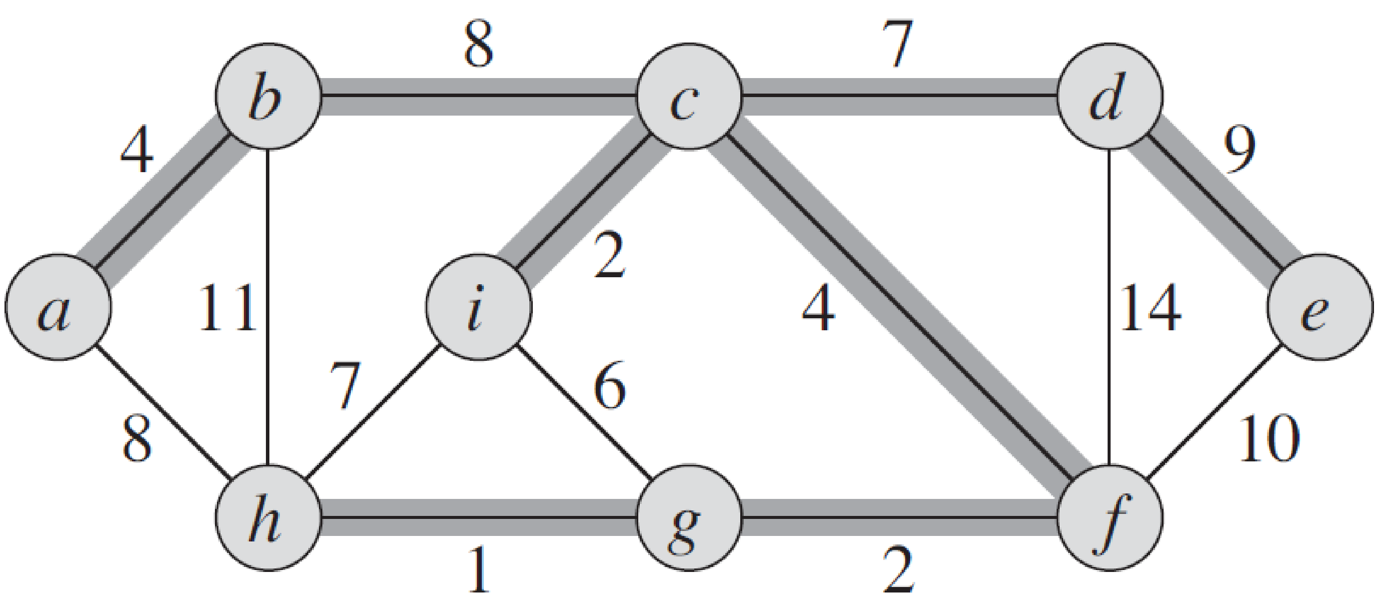
Minimum Spanning Trees

- We can model the wiring problem with a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins.
- For each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect u and v.
- We then wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight is minimized.



Spanning Tree Example

- Since *T* is acyclic and connects all of the vertices, it must form a tree, which we call a ***spanning tree*** since it "spans" the graph *G*.
- We call the problem of determining the tree *T* the ***minimum-spanning-tree problem*** (MST).



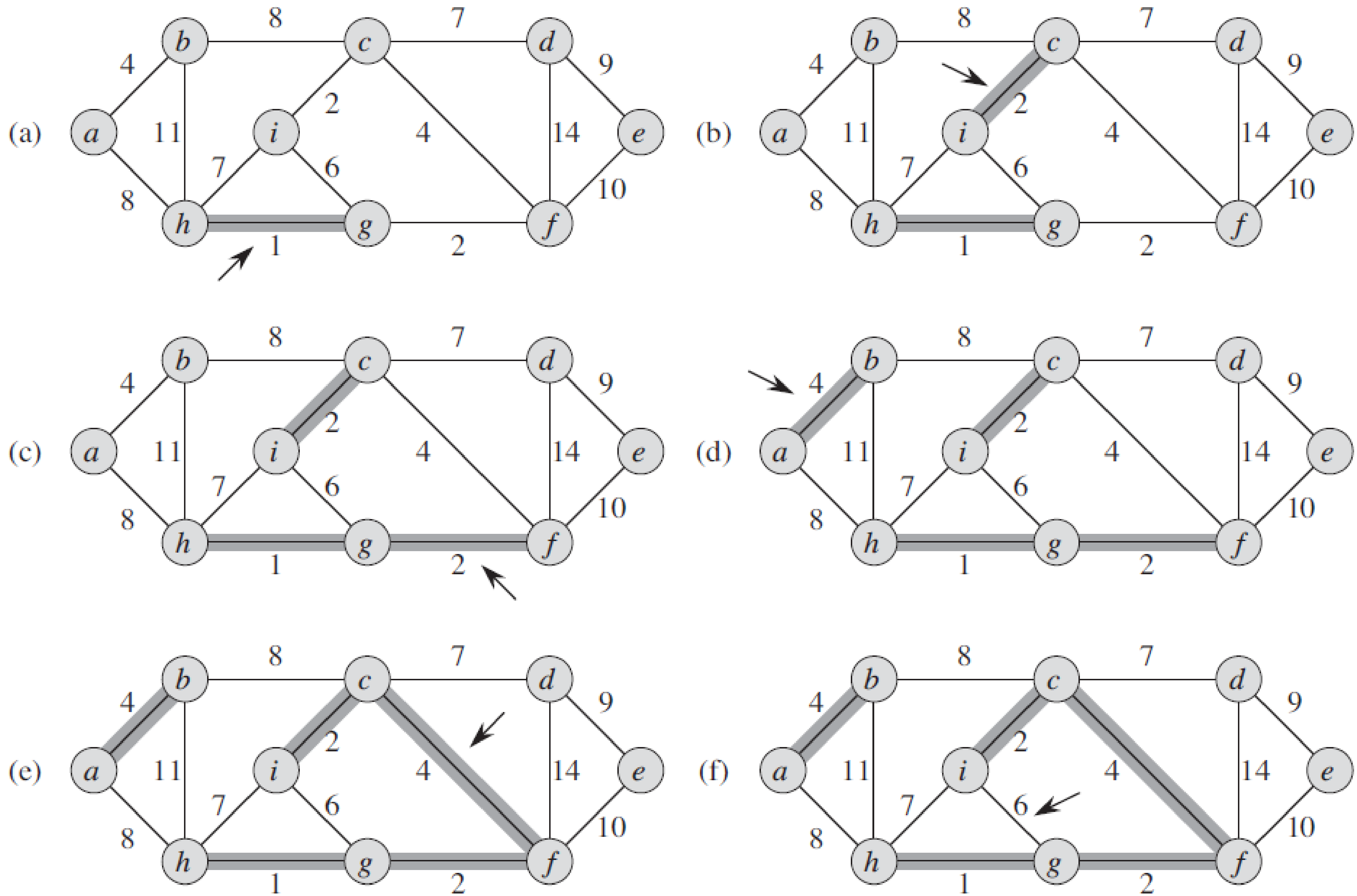
MST: Kruskal's Algorithm

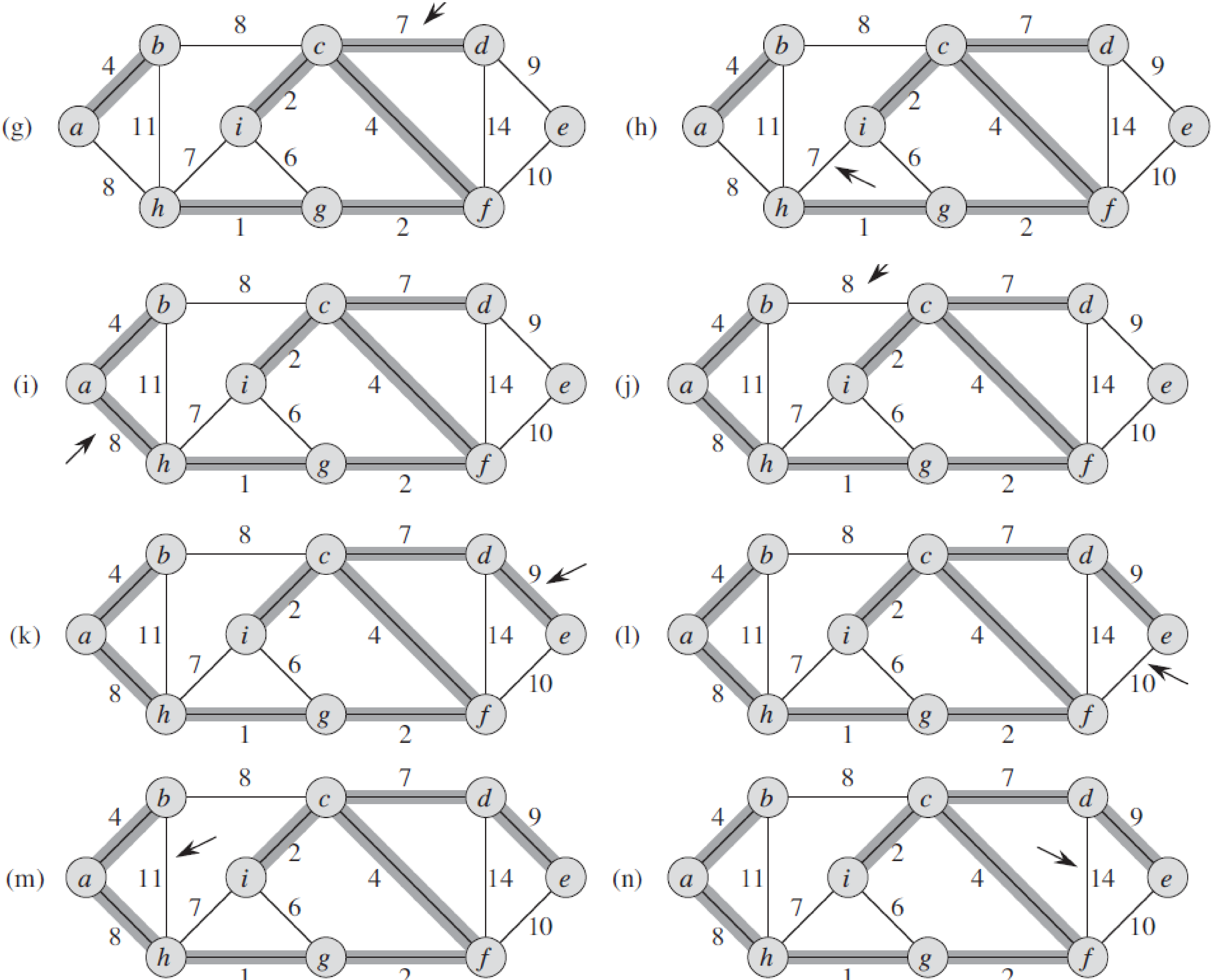
```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

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

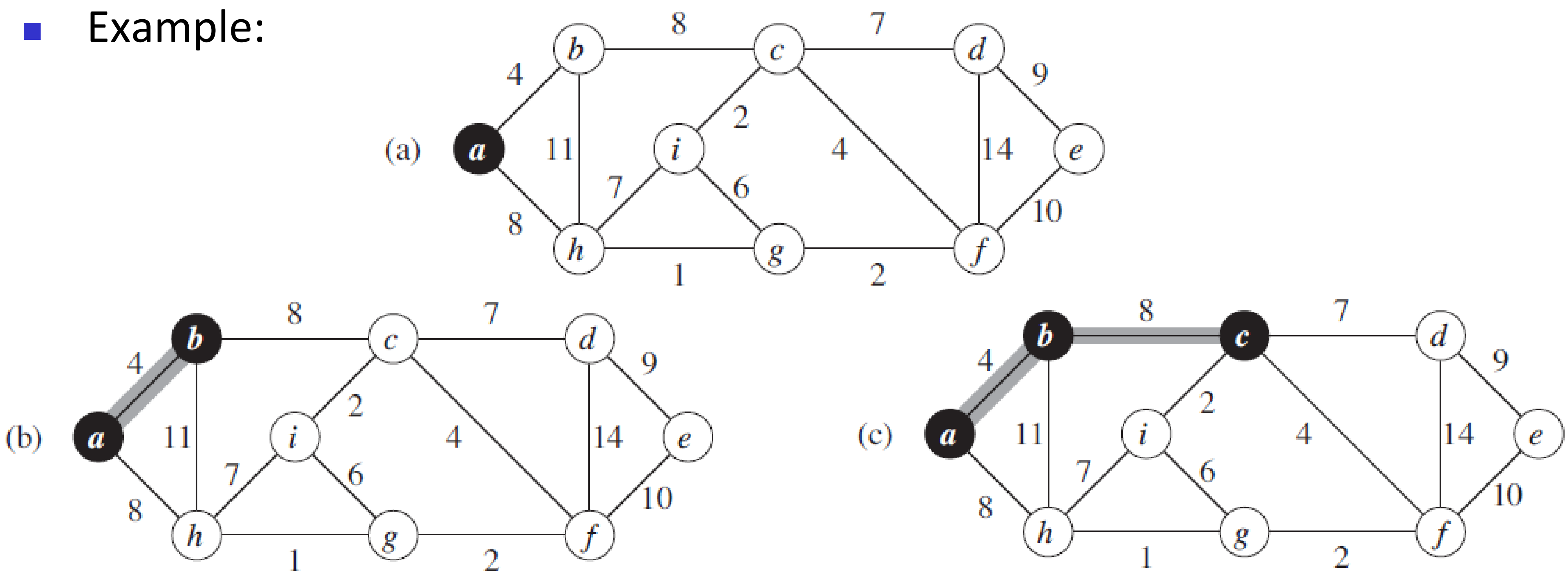
Kruskal's Algorithm Example



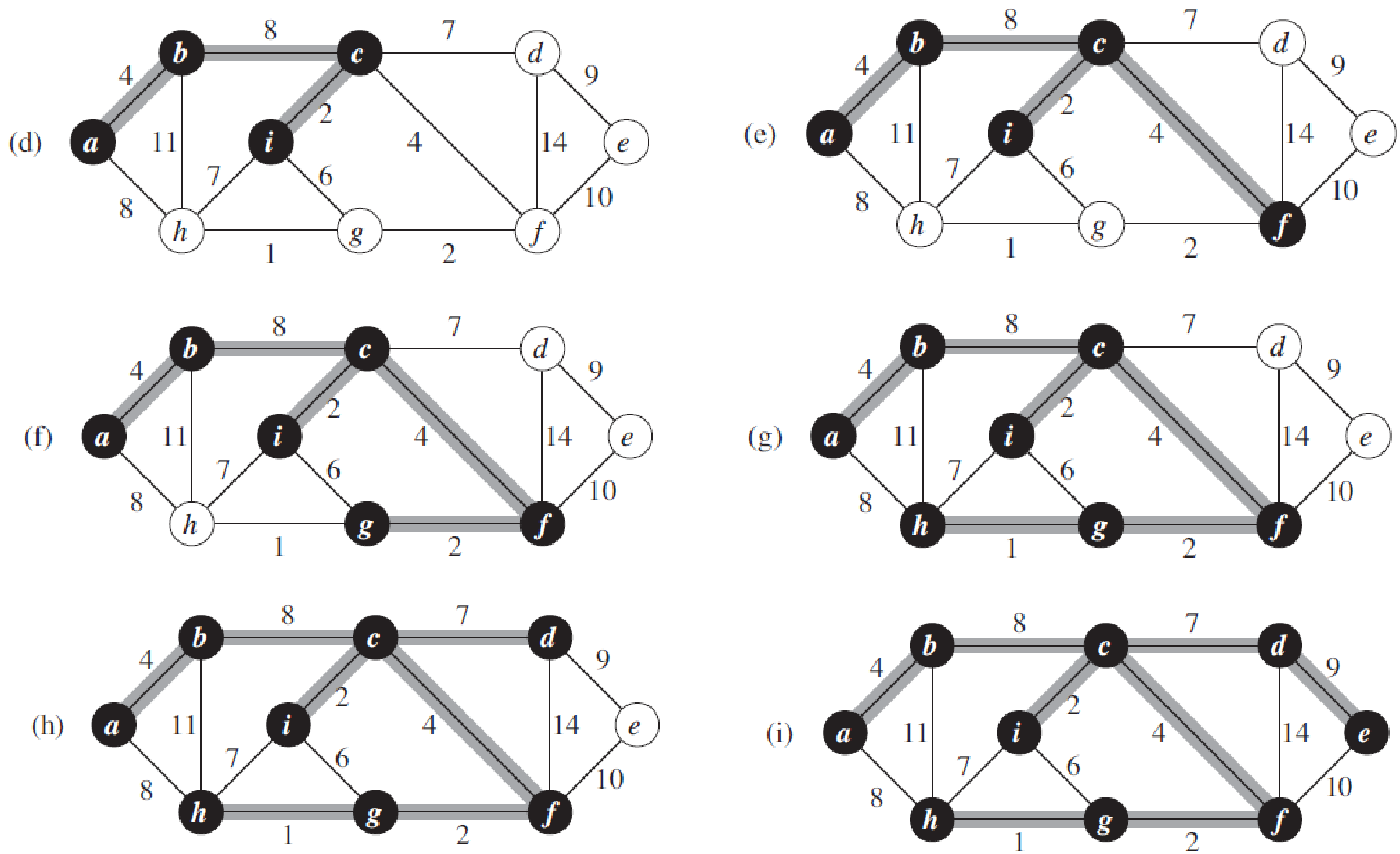

(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)

MST: Prim's Algorithm

- Prim's algorithm has the property that the edges in the set *A* always form a single tree.
- The tree starts from an arbitrary root vertex *r* and grows until the tree spans all the vertices in *V*.
- Each step adds to the tree *A* the "lightest" edge that connects *A* to an isolated vertex—one on which no edge of *A* is incident.
- Example:

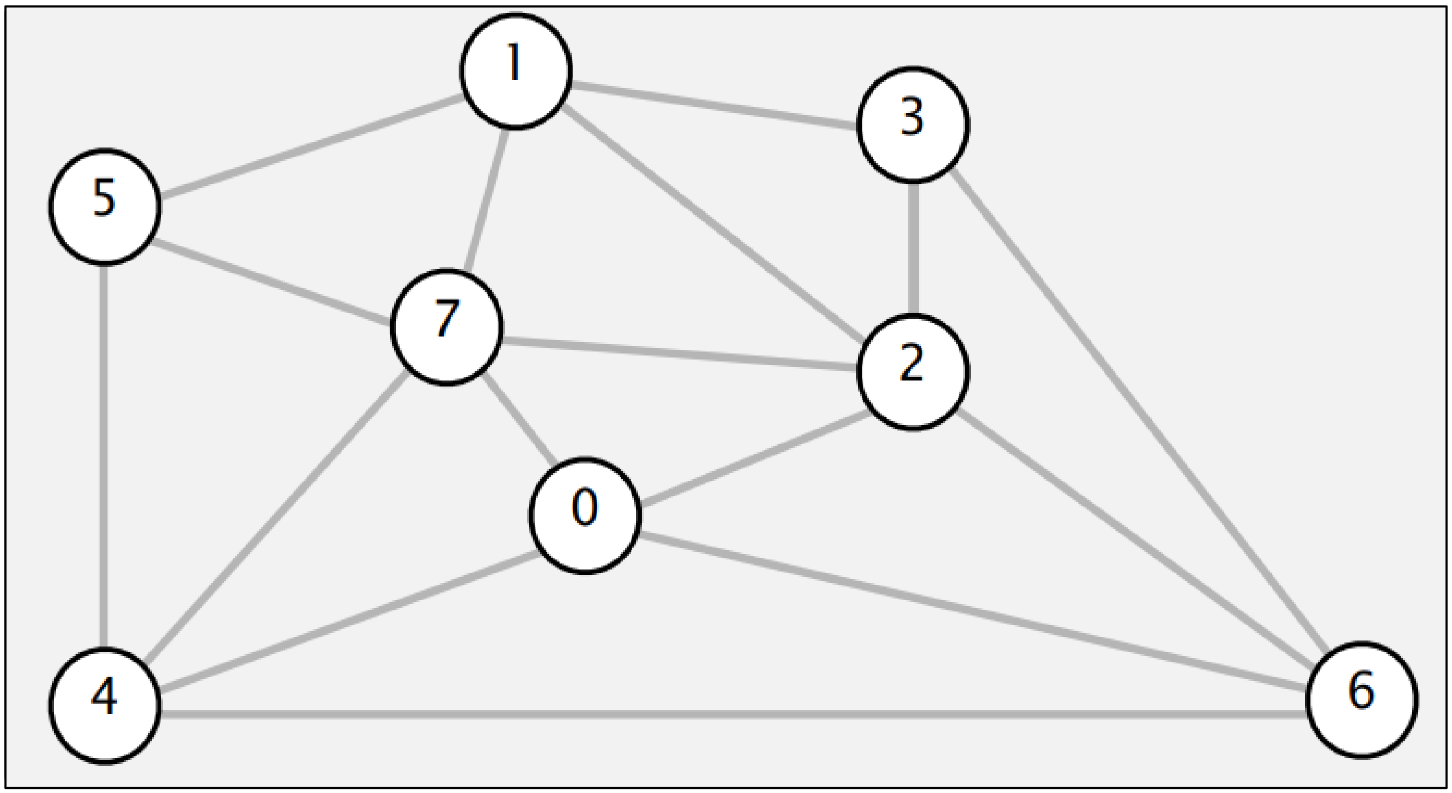


Prim's Algorithm Example (Cont'd)



MST Exercise

- Find the MST of the following graph using both Kruskal's Algorithm and Prim's Algorithm.



graph edges sorted by weight

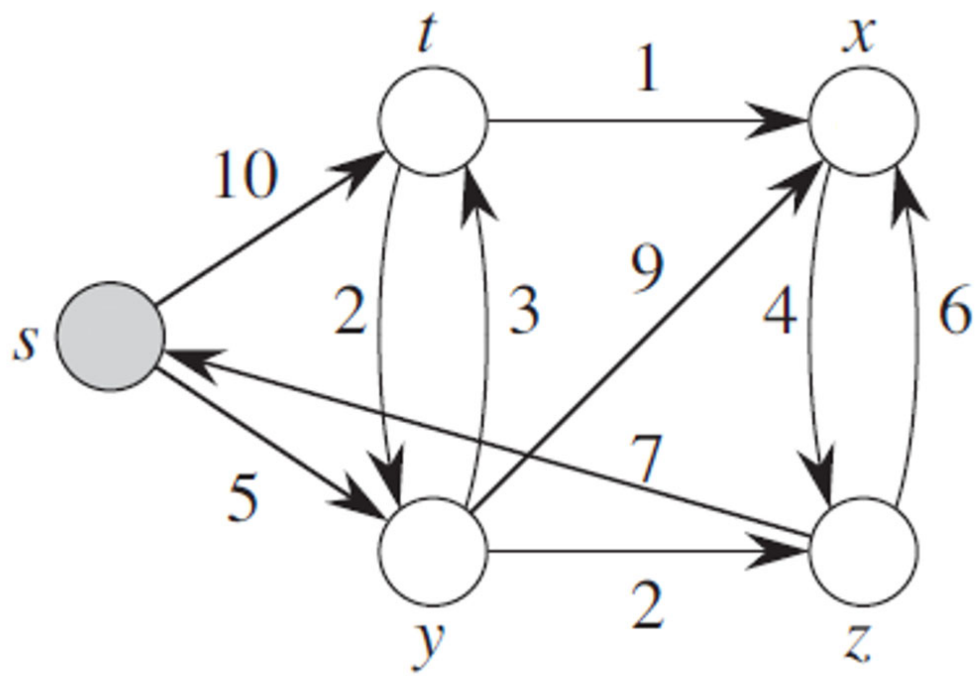
0-7	16
2-3	17
1-7	19
0-2	26
5-7	28
1-3	29
1-5	32
2-7	34
4-5	35
1-2	36
4-7	37
0-4	38
6-2	40
3-6	52
6-0	58
6-4	93

Shortest Paths

- Application example:

  Finding the shortest possible route from *Boston* to *Providence*.

  Given a road map of the *United States* on which the distance between each pair of adjacent intersections is marked.



Shortest Paths

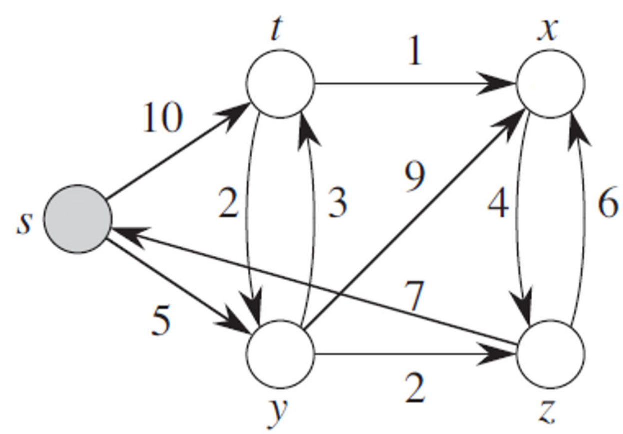
- Consider a weighted, directed graph $G = (V, E)$ with edge-weight function : $w : E \rightarrow \mathbb{R}$.
- The ***weight*** of path $p = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$ is defined to be

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

- A ***shortest path*** from u to v is a path of minimum weight from u to v. The ***shortest- path weight*** from u to v is defined as

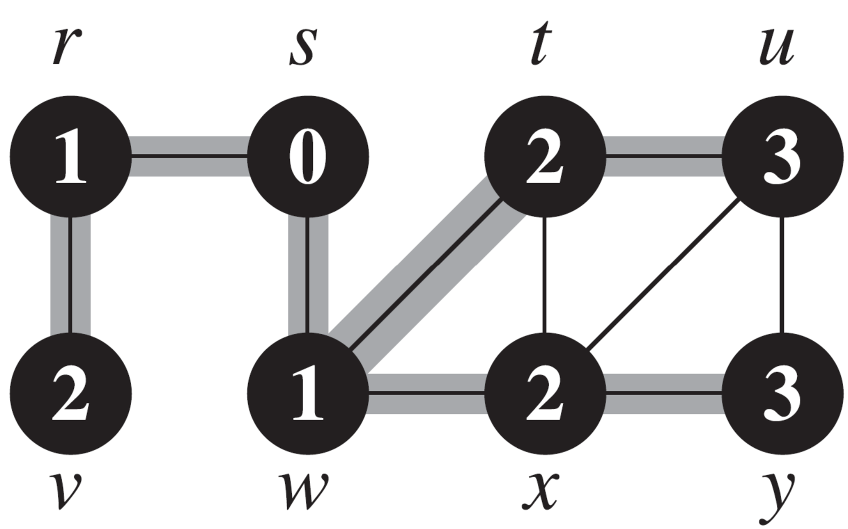
$$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}.$$

- **Note:** $\delta(u, v) = \infty$ if no path from u to v exists.



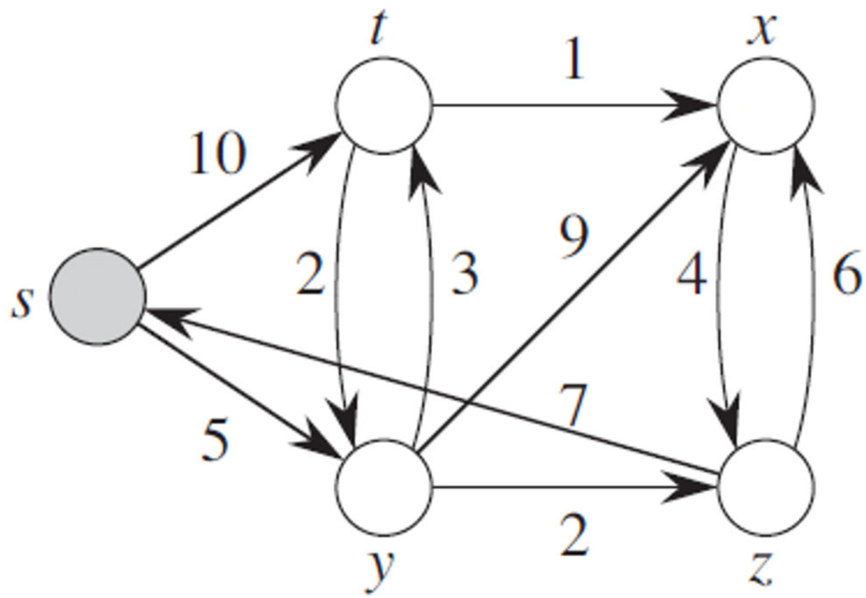
Edge Weights

- Edge weights can represent metrics such as distances, time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that we would want to minimize.
- The breadth-first-search (BFS) algorithm, as shown below, is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge has unit weight.



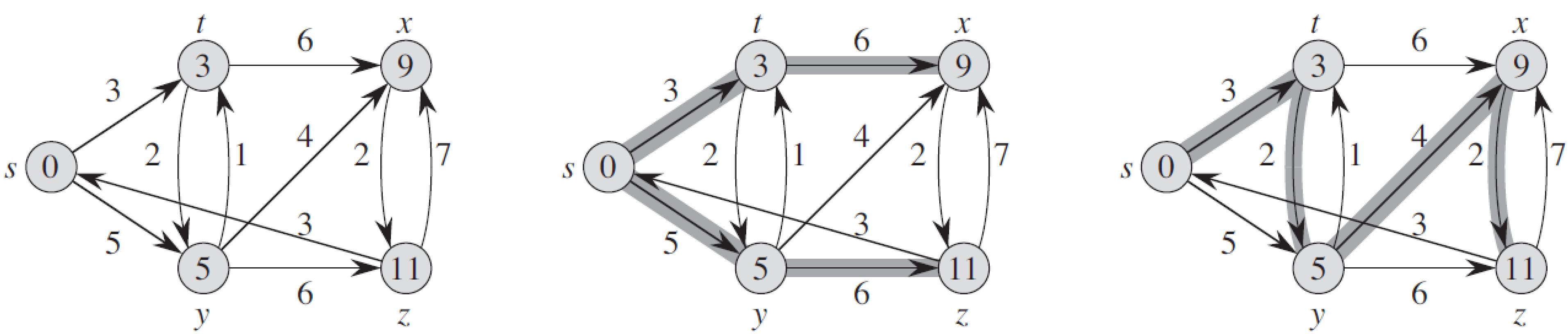
Single-Source Shortest Paths (1 of 2)

- The ***single-source shortest-paths problem***: given a graph $G = (V, E)$, we want to find a shortest path from a given ***source*** vertex $s \in V$ to each vertex $v \in V$.
- The algorithm for the single-source problem can solve many other problems, including the following variant:
  - **Single-destination shortest-paths problem:** Find a shortest path to a given ***destination*** vertex t from each vertex v. By reversing the direction of each edge in the graph and then solve the single-source problem.



Single-Source Shortest Paths (2 of 2)

- Shortest paths are not necessarily unique.
- *Example*: The following figure shows a weighted, directed graph and two shortest-paths trees with the same root.

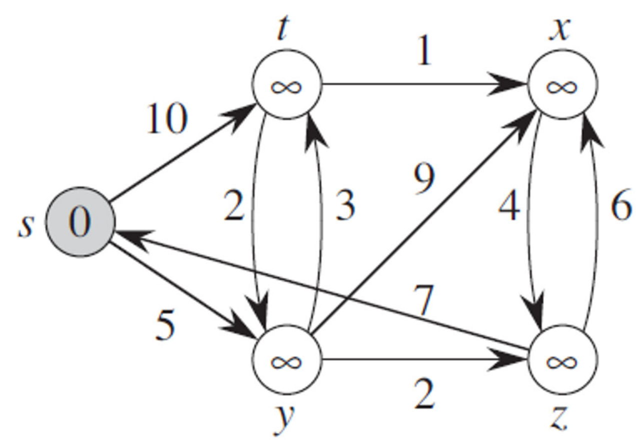


(a) (b) (c)

Initialization

- The shortest paths algorithms start with the initialization process where for each vertex $v \in V$:
  - $v.d$ is the upper bound on the weight of a shortest path from source s to v and it is initialized to ∞.
  - $v.d$ is called a ***shortest-path estimate***.
  - $v.\pi$ is vertex v's ***predecessor***.

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

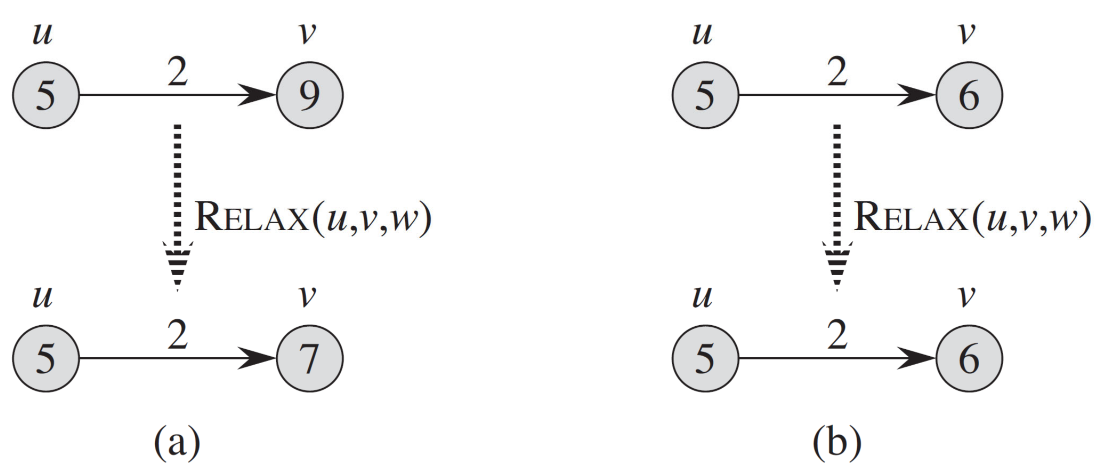


Relaxation (1 of 2)

- The process of ***relaxing*** an edge (*u*, *v*) consists of testing whether we can improve the shortest path to *v* that has been found so far by going through *u* and, if so, updating *v.d* and *v.π*.

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

- *Example:*



Relaxation (2 of 2)

- Shortest path algorithms calls INITIALIZE-SINGLE-SOURCE once and then repeatedly relaxes edges.
- Relaxation is the only means by which shortest path estimates and predecessors change.
- The algorithms differ in how many times they relax each edge and the order in which they relax edges.

Dijkstra's Algorithm (1 of 2)

- 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 *S*** of vertices whose final shortest-path weights from the source s have already been determined.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.
- The algorithm uses a **min-priority queue *Q*** of vertices, keyed by their d values (the shortest-path estimate).

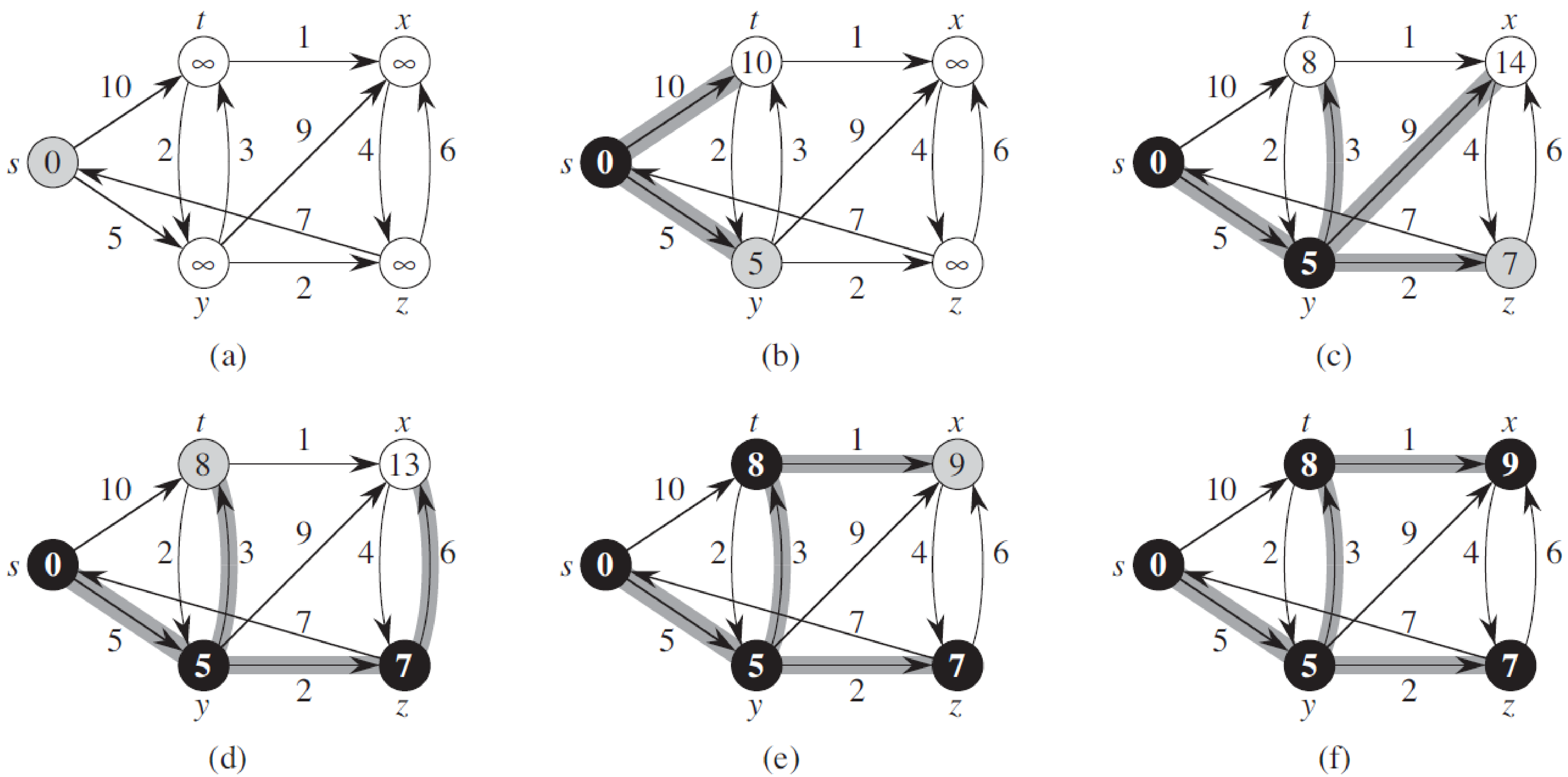
Dijkstra's Algorithm (2 of 2)

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

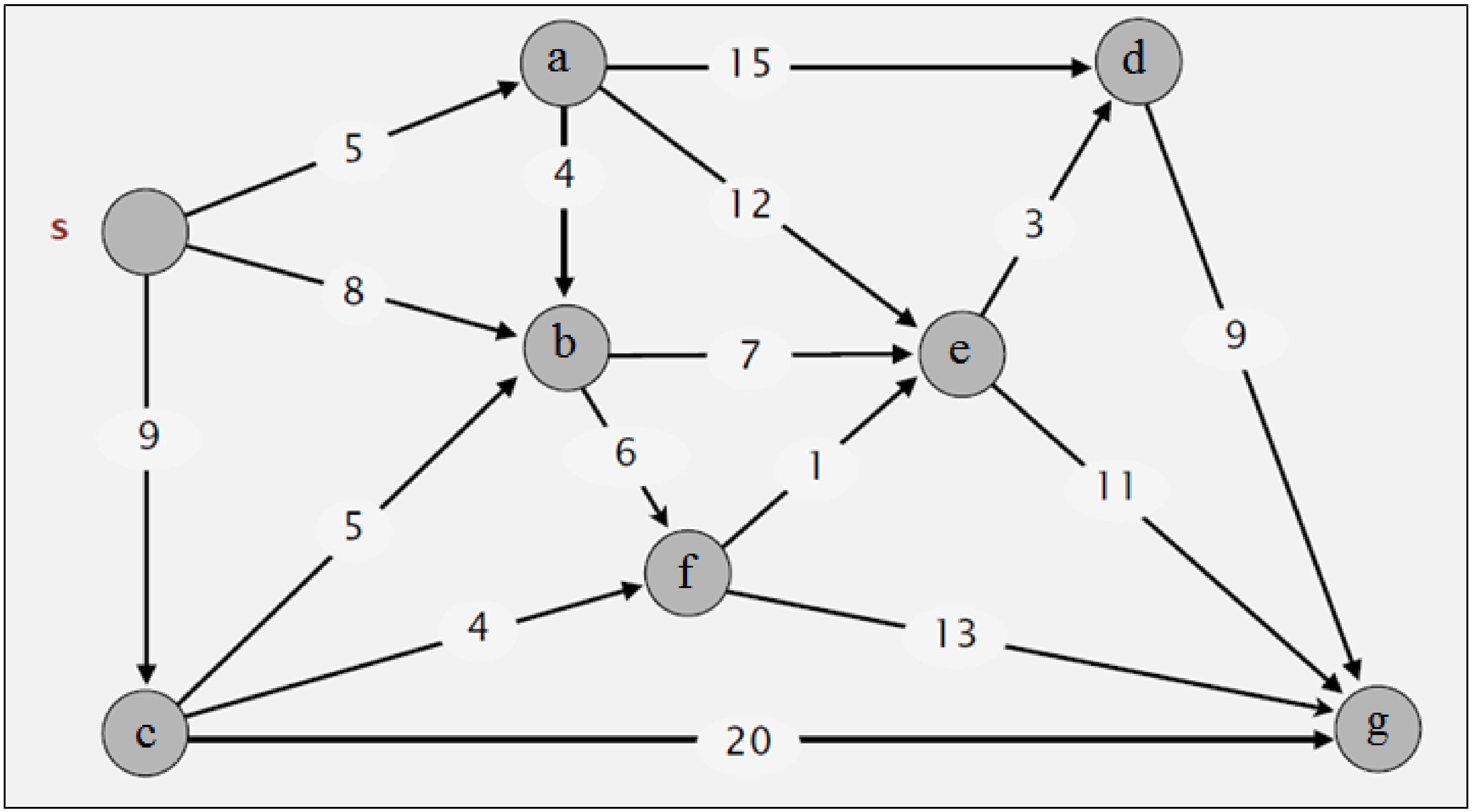
Dijkstra's Algorithm Example

- Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. The shaded vertex is vertex u with the minimum d value.



Dijkstra’s Algorithm Exercise

- Find the single-source shortest-paths from vertex *s* in the shown weighted, directed graph.

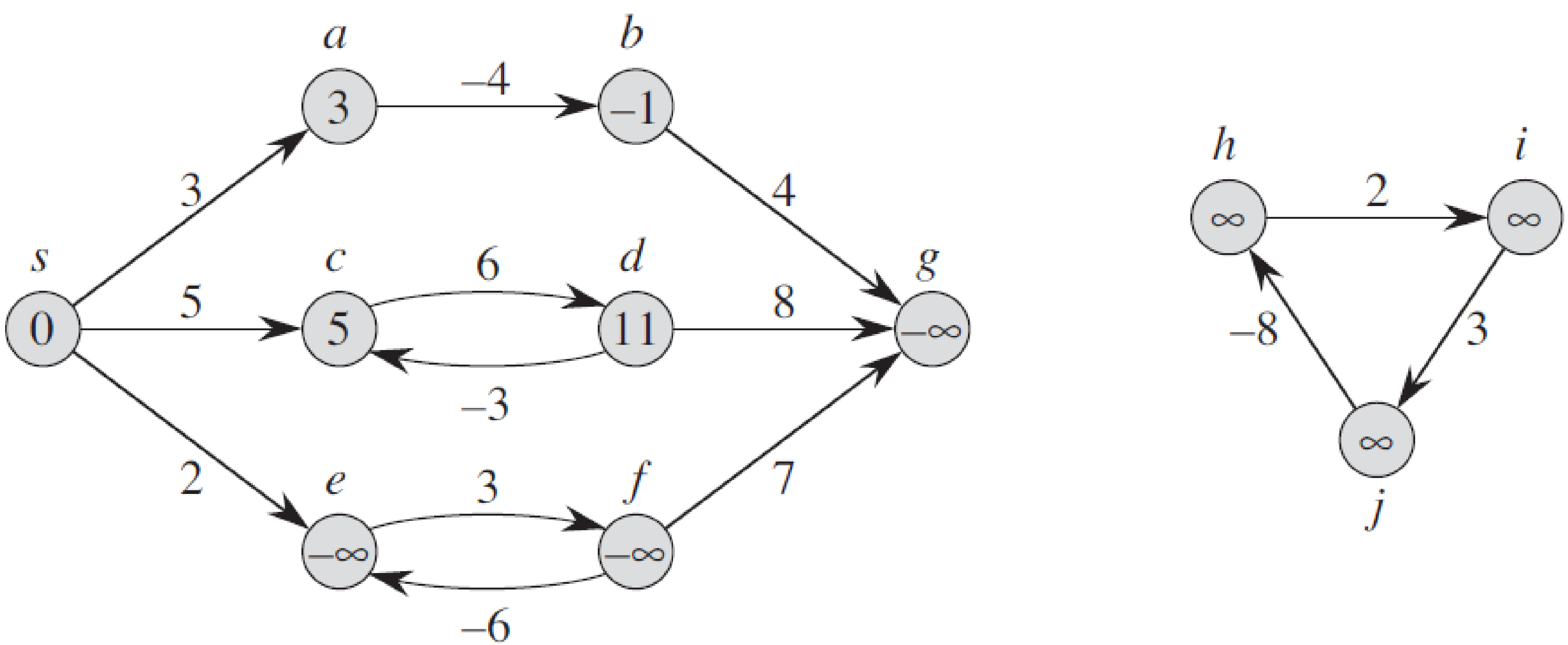


Negative-Weight Edges

- 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 $\delta(s, v) = -\infty$.

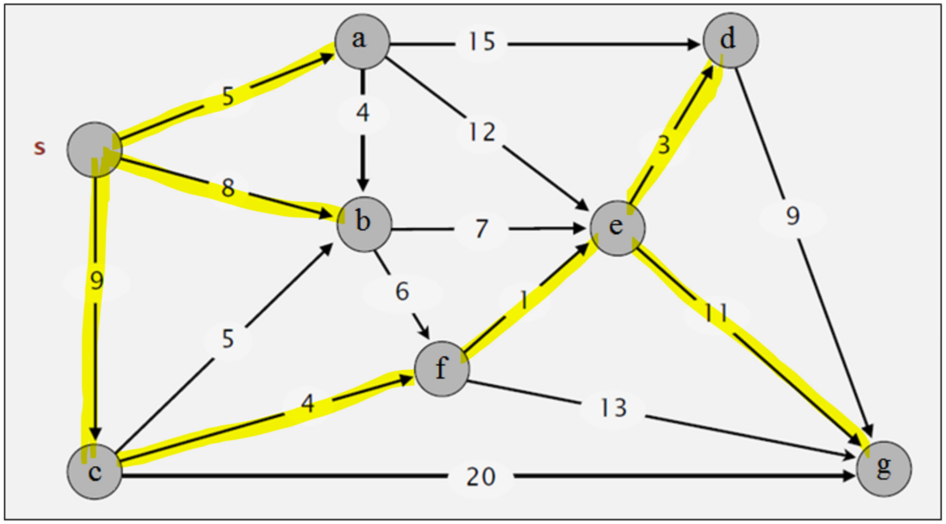
Negative-Weight Cycle Example

- 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.



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 $v.d \ge \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ 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 $v.d = \delta(s, v) = \infty$.
- **Predecessor-subgraph property**
  - Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.



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)

```
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.

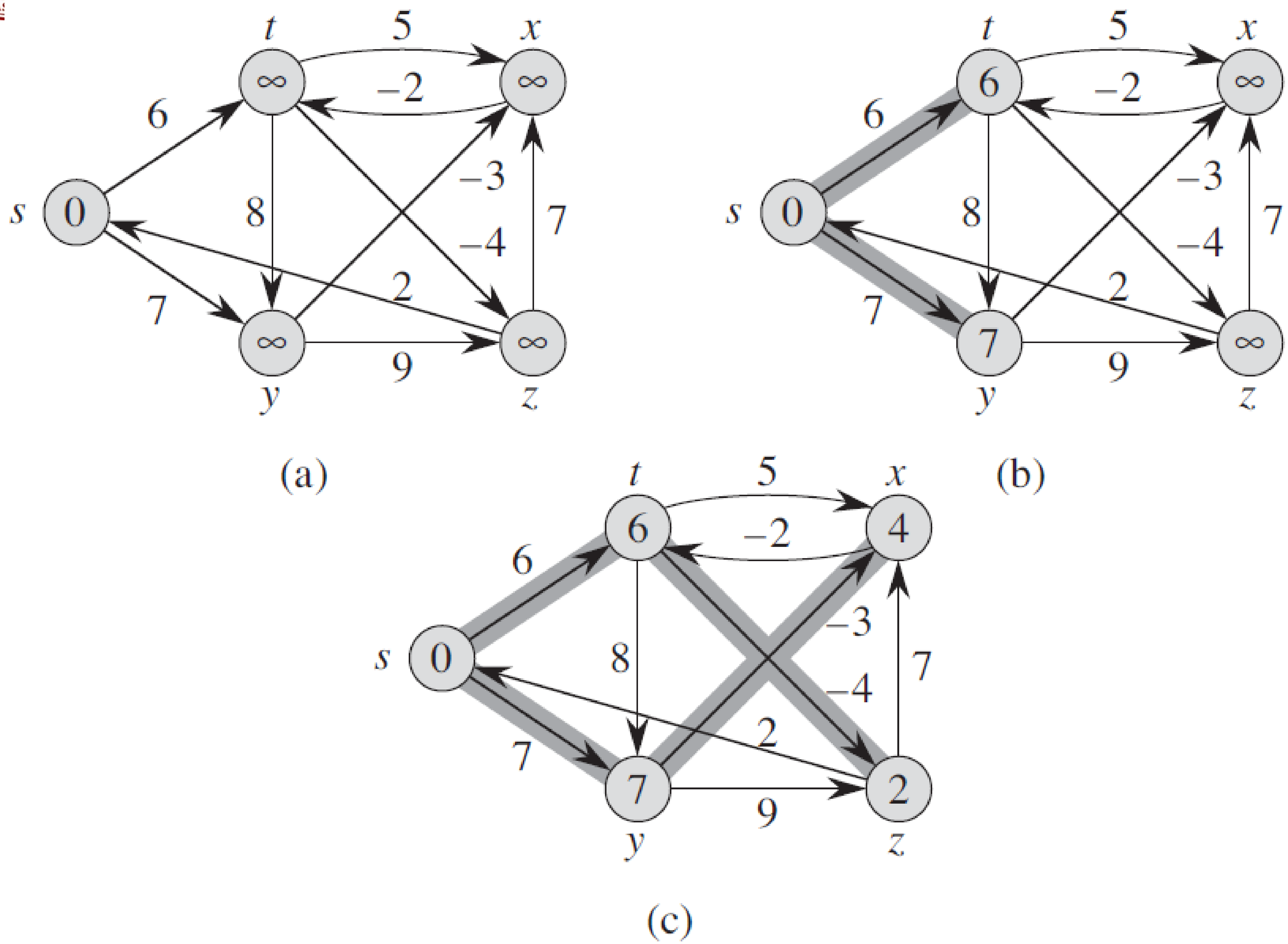
It checks for a negative-weight cycle and returns the appropriate Boolean value.

Triangle inequality

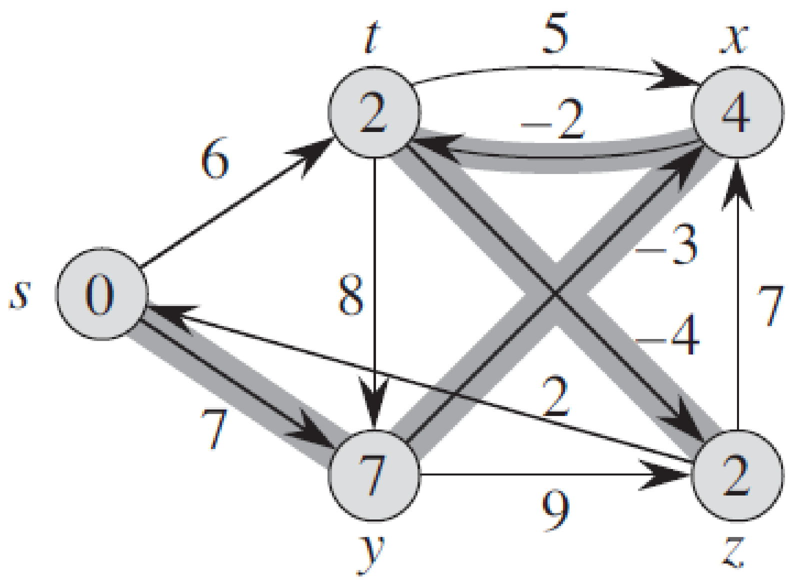
The Bellman-Ford Algorithm (3 of 3)

- 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* has entry *False* in array *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 (to be discussed later in the course).

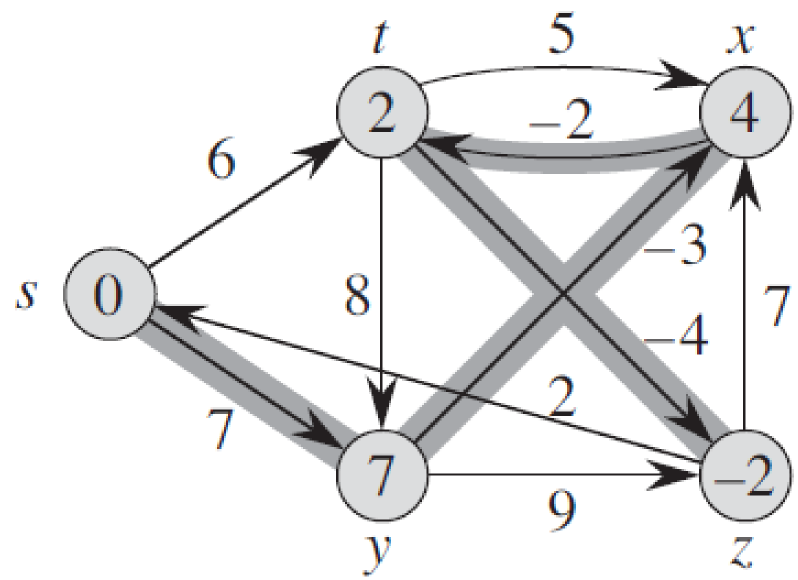
Bellman-Ford Example (1 of 2)



Bellman-Ford Example (2 of 2)



(d)

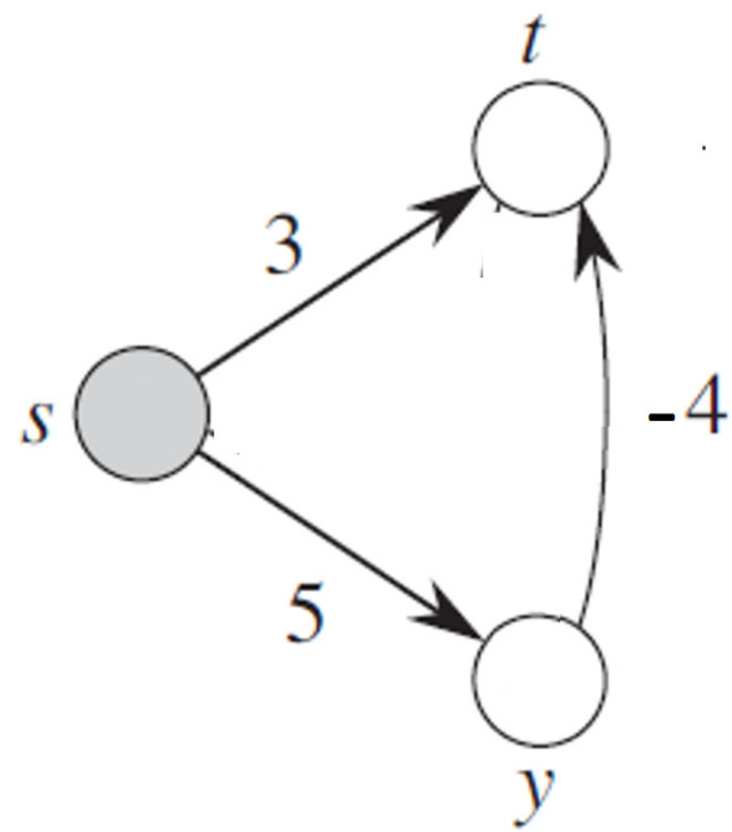


(e)

- 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.

Dijkstra's vs. Bellman-Ford

- Use both Dijkstra's and Bellman-Ford to solve the single-source shortest-paths problem for the shown graph where node ***s*** is the source.
- Using Dijkstra's will not produce the correct answer while Bellman-Ford will result in the correct answer.
- So, Dijkstra's does not work when edges have negative weight regardless if there is a negative cycle or not.

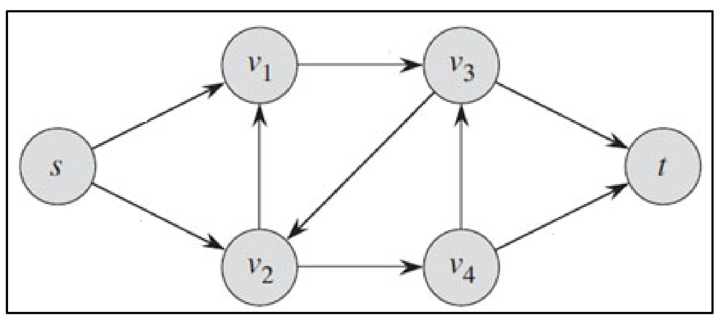


Flow Network

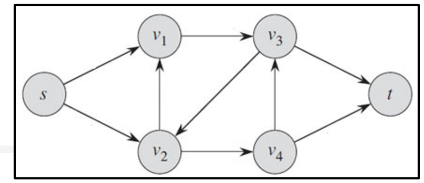
- We modelled a road map as a directed graph in order to find the shortest path from one point to another, we can also interpret a directed graph as a “flow network” and use it to answer questions about material flows.
- Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed.
- The source produces the material at some steady rate, and the sink consumes the material at the same rate.
- The “flow” of the material at any point in the system is intuitively the rate at which the material moves.
- Flow networks can model many problems, including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks.

Flow Networks Properties

- A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v)$.
- It is required that **self-loops are not allowed**.
- If E contains an edge (u, v) then there is **no edge (*v*, *u*) in the reverse direction**.
- Two vertices are distinguished in a flow network: a ***source s*** and a ***sink t***.
- *Assumption:* each vertex lies on some path from the source to the sink. The graph is therefore **connected** and each vertex other than s has at least one entering edge.



Flow Definition



- Let $G = (V, E)$ be a flow network with a capacity function ***c*** and let ***s*** be the source of the network, and let ***t*** be the sink. A ***flow*** in G is a real-valued function $f : V \times V \rightarrow R$ that satisfies the following two properties:

1. **Capacity constraint:** For all $u, v \in V$, we require

$$0 \leq f(u, v) \leq c(u, v)$$

which says that the flow from one vertex to another must be nonnegative and must not exceed the given capacity

2. **Flow conservation:** For all $u \in V - \{s,t\}$, we require "*flow in equals flow out*"

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) .$$

When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v)=0$.

The Maximum Flow Problem

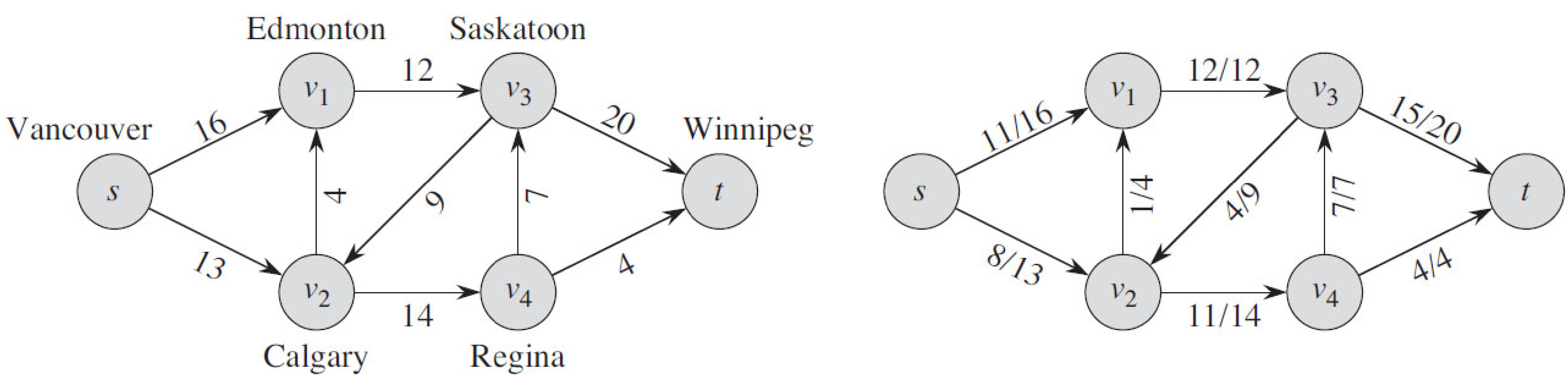
- In the maximum-flow problem, we wish to compute the greatest rate at which we can ship material from the source to the sink without violating any capacity constraints.
- The total flow, ***f***, out of the source ***s***, is defined as:

$$|f| = \sum_{v \in V} f(s, v)$$

- 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.

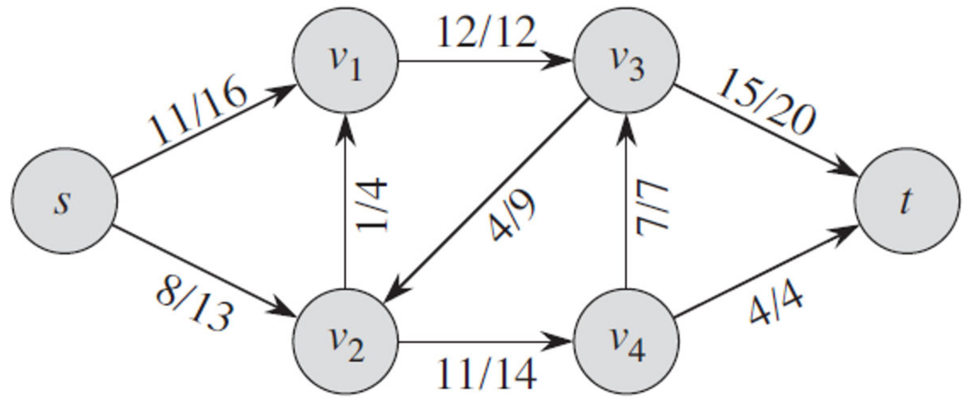
Flow Network Example (1 of 2)

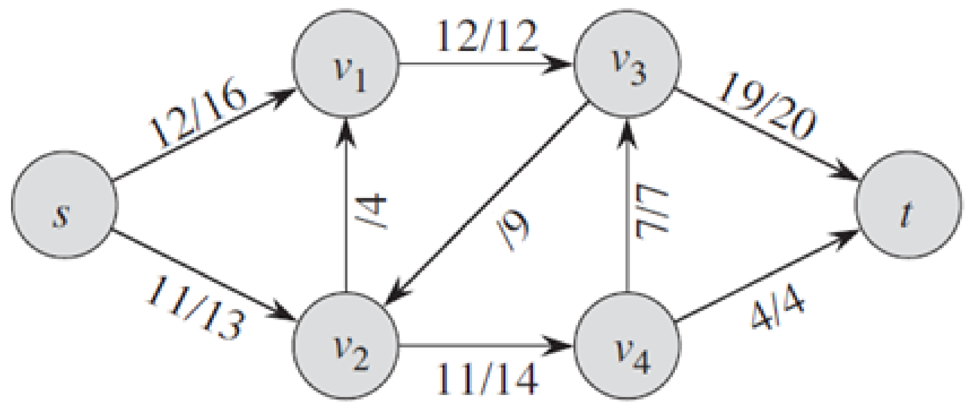
- The left figure model the trucking problem of a company *LP*. *LP* has a factory (source ***s***) in Vancouver that manufactures furniture, and it has a warehouse (sink ***t***) in Winnipeg that stocks them.
- The trucks ships the furniture over specified routes (edges) between cities (vertices). Routes have a limited capacity of at most $c(u, v)$ furniture containers per day between each pair of cities ***u*** and ***v***.
- *LP* needs to determine the largest number ***p*** of containers per day that they can ship. *LP* is not concerned with how long it takes for the furniture to get from the factory to the warehouse; they care only in maximizing ***p***.
- The right figure represent one possible solution. Is it maximum?



Flow Network Example (2 of 2)

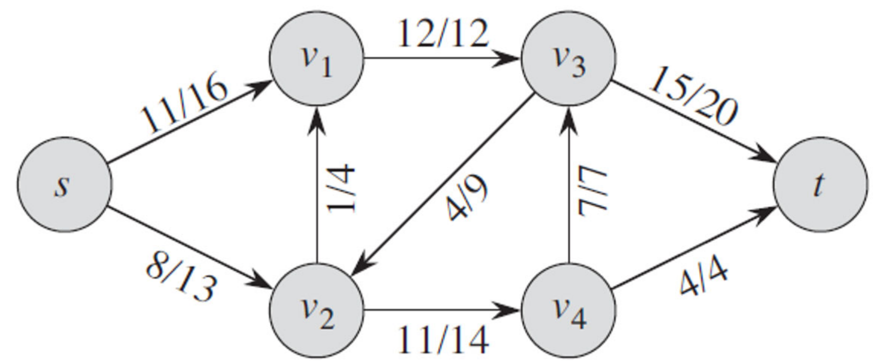
- The top solution has a flow of 19, while the bottom solution has a better flow of 23.
- But how can we improve the top solution to reach the bottom one?
- As you see, we had to remove an already assigned flows (v_2 to v_1 and v_3 to v_2).



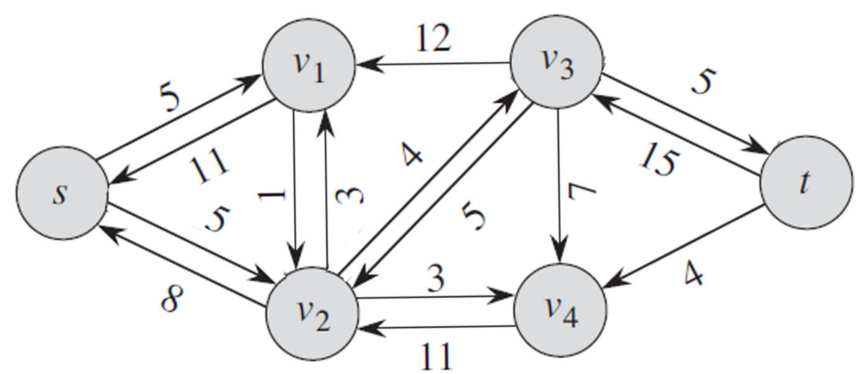


Residual Networks

- The figures show the flow network G followed by its residual network G_f .
- Edges in G_f have the following “residual capacity” c_f in any edge (u, v)



$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \ , \\ f(v,u) & \text{if } (v,u) \in E \ , \\ 0 & \text{otherwise} \ . \end{cases}$$



- Edges with $c_f = 0$ are not included in G_f.
- Observe that the residual network G_f does not satisfy the definition of a flow network because it may contain both an edge (u, v) and its reversal (v, u).

Augmentation

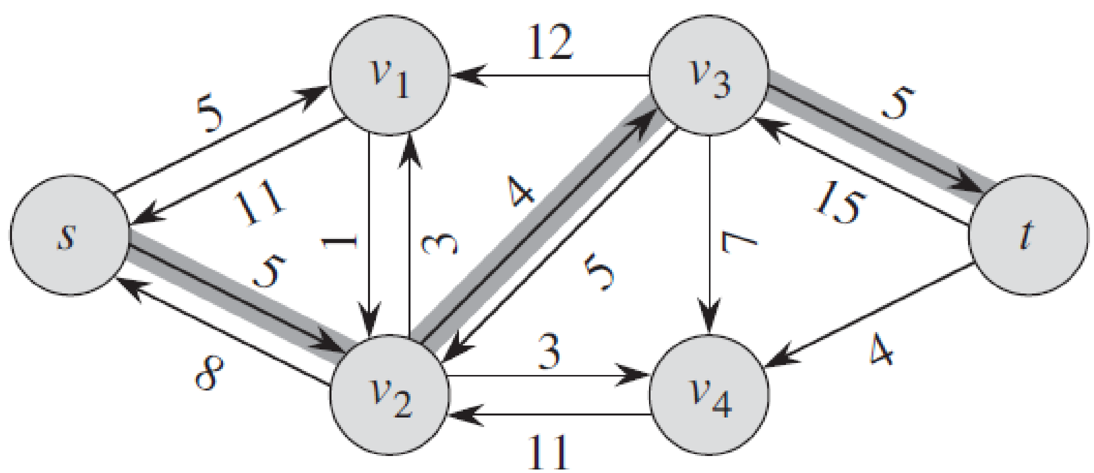
- We define $f \uparrow f'$, the ***augmentation*** of flow f by f', to be:

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

- The flow on (u, v) is increased by $f'(u, v)$ but is decreased by $f'(v, u)$ because pushing flow on the reverse edge in the residual network signifies decreasing the flow in the original network.
- Pushing flow on the reverse edge in the residual network is also known as ***cancellation***.

Augmenting Paths

- Given a residual network $G_f = (V, E_f)$ and a flow ***f***, an *augmenting path* ***p*** is a simple path from ***s*** to ***t*** in G_f.
- The shaded path in the shown figure is an augmenting path.
- Since the smallest residual capacity on this path is $c_f(v_2, v_3) = 4$, the flow through each edge of this path can be increased by up to 4 units without violating a capacity constraint.



Residual Capacity

- The ***residual capacity*** of an augmenting path ***p*** is the maximum amount by which we can increase the flow on each edge in ***p*** and it is given by:

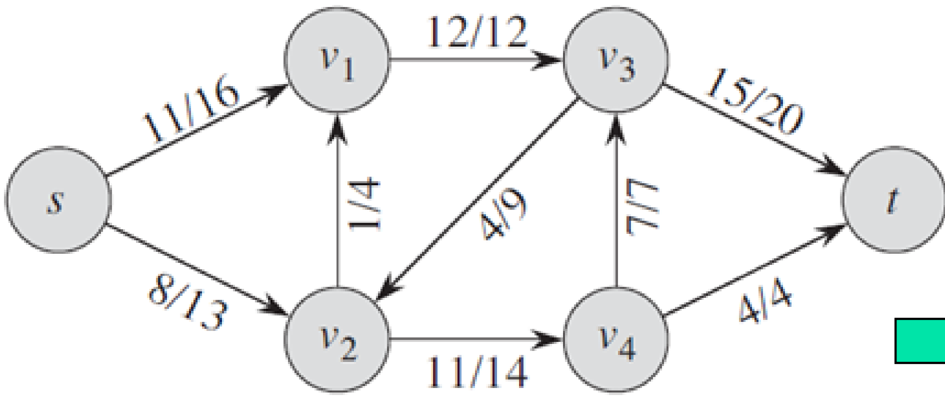
$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$$

- Leading to the following definition:

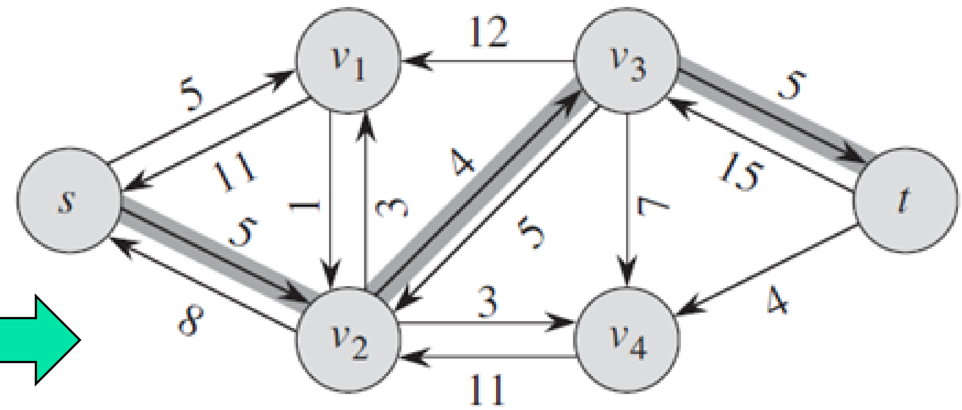
$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p , \\ 0 & \text{otherwise} . \end{cases}$$

- *Corollary*: If we augment the current flow ***f*** by $\boldsymbol{f_p}$, we get another flow in ***G*** whose value is closer to the maximum.

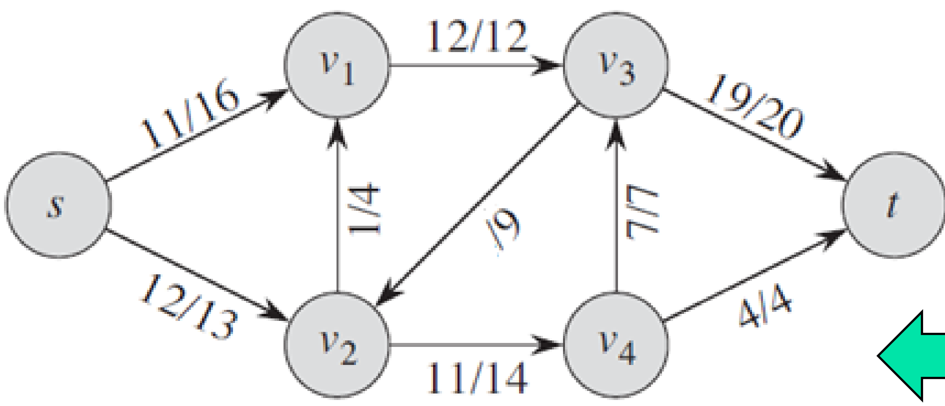
Augmentation Example



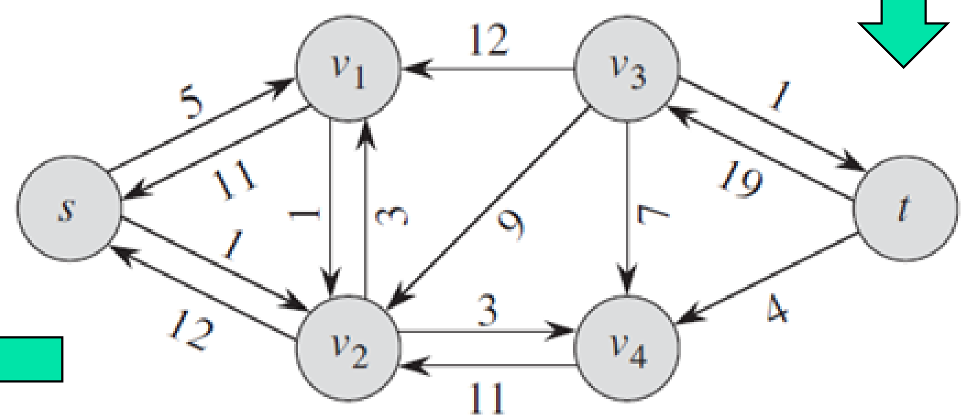
G with flow *f*



G_f with flow $f_p = 4$



G after augmenting *f* with f_p



Updated G_f

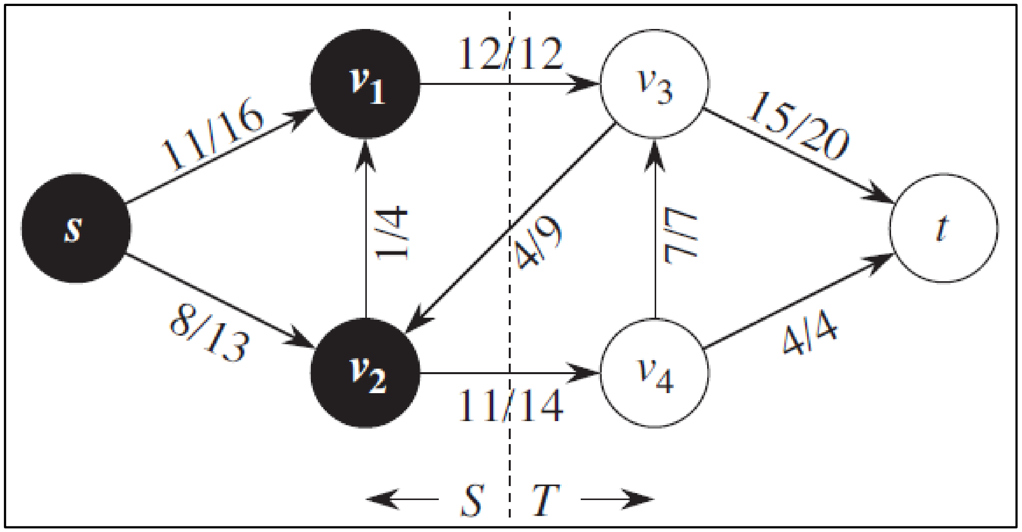
Cuts of Flow Networks

- 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*** ∈ ***S*** and ***t*** ∈ ***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

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



- *Example*:
  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.
- Let ***f*** be a flow in a flow network ***G*** with source ***s*** and sink ***t***, then *net flow* across any cut is **| *f* |**.
- *Corollary*: The value of any flow ***f*** in a flow network ***G*** is bounded from above by the capacity of any cut of ***G***.
- The value of a *maximum flow* in a network equals the capacity of a *minimum cut* of the network.
- The *max-flow min-cut* theorem tells us that a flow is maximum if and only if its residual network contains no augmenting path.

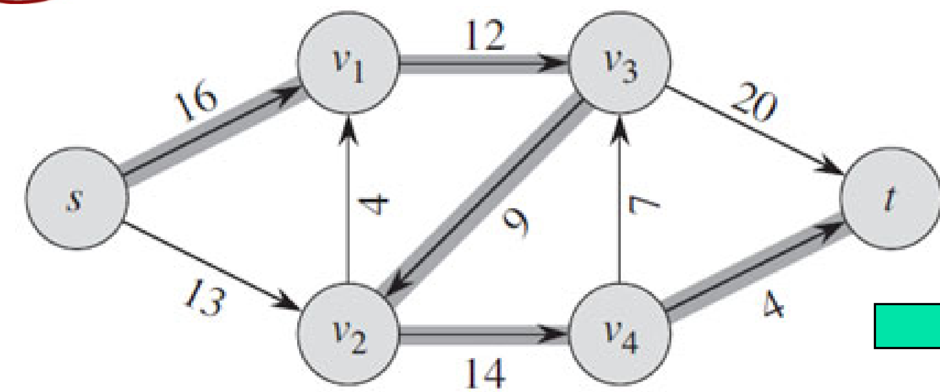
Ford-Fulkerson Algorithm

- The *Ford-Fulkerson* method repeatedly augments the flow along augmenting paths until it has found a maximum flow.

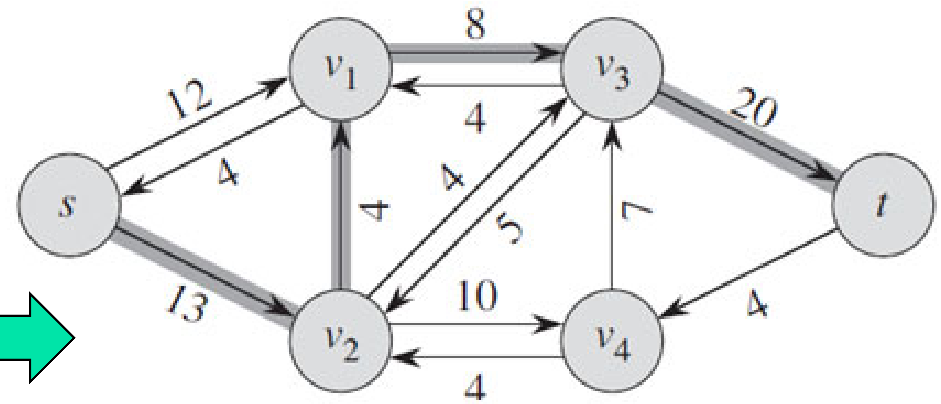
```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min{c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
```

- Lines 6–8 update the flow by adding flow when the residual edge is an edge in the original network and subtracting it otherwise.
- When no augmenting paths exist, the flow ***f*** is a maximum flow.

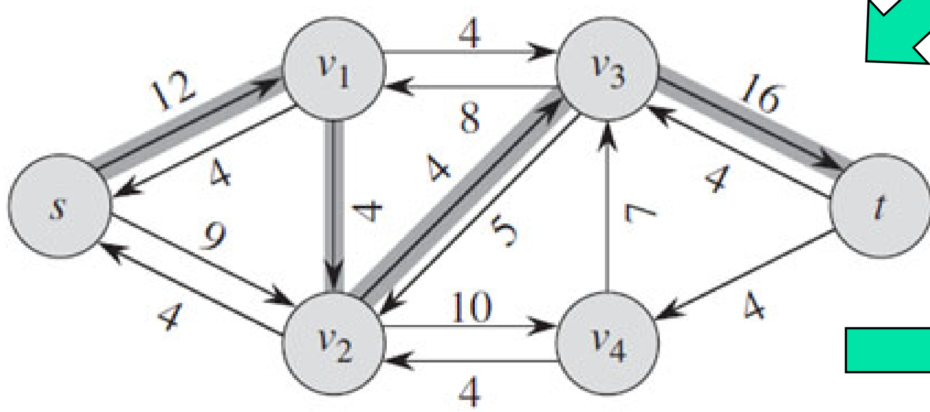
Ford-Fulkerson Example (1 of 3)



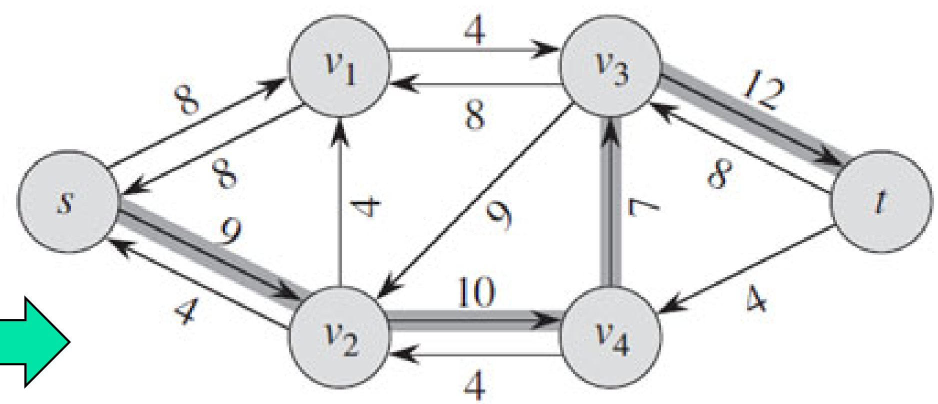
Input G_f with augmenting path p shaded and $c_f = 4$



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

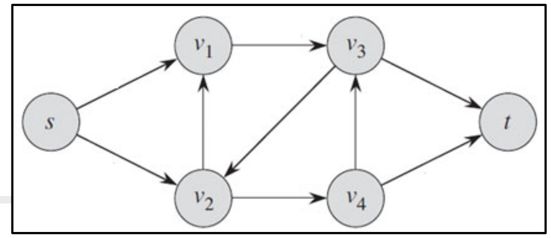


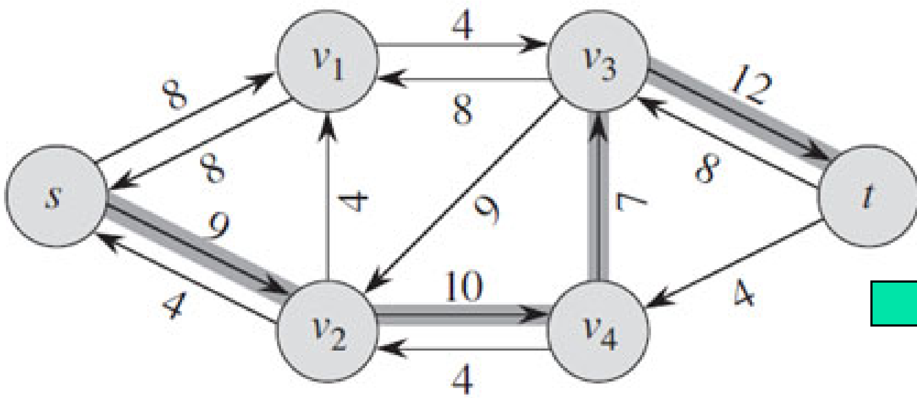
Updated G_f with augmenting path p shaded and $c_f = 4$



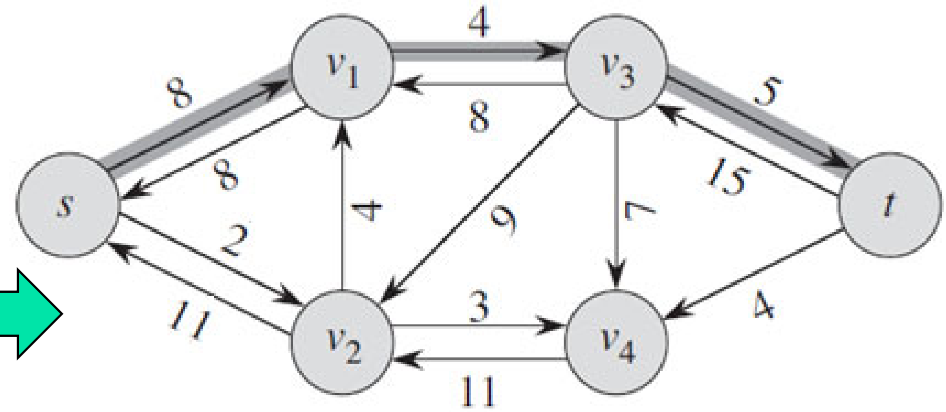
Updated G_f with augmenting path p shaded and $c_f = 7$

Example (2 of 3)

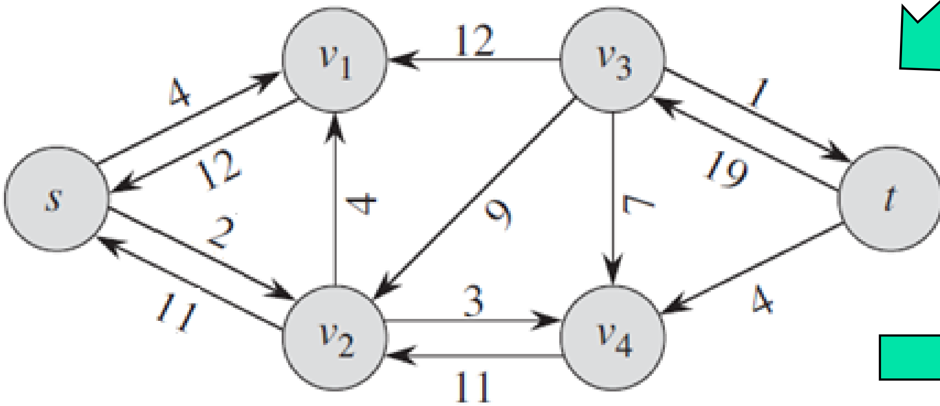




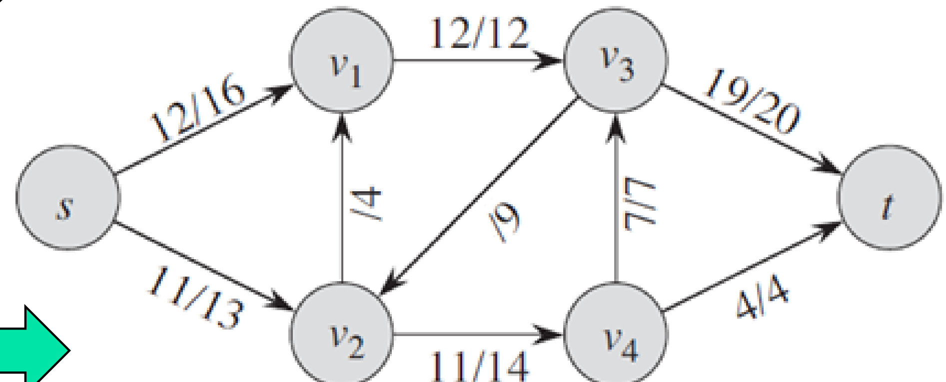
(Repeated)



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 $f = 23$

Example (3 of 3)

- As you see the ***minimum cut*** of the given flow network *G* (which is a cut whose capacity is minimum over all cuts of the network) has ***capacity*** of 23, which matches the maximum flow found by the *Ford-Fulkerson* algorithm.