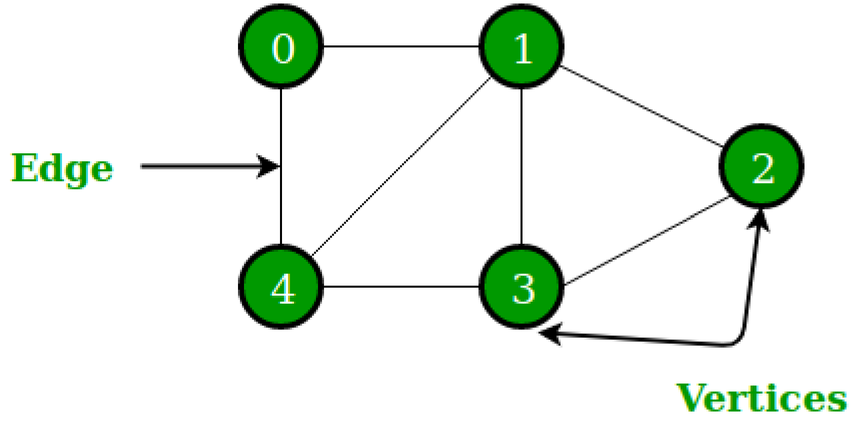
Design and Analysis of Algorithm

Mahesh Shirole

VJTI, Mumbai-19

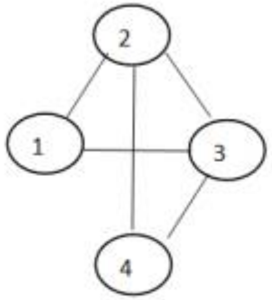
Slides are prepared from
1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Graph

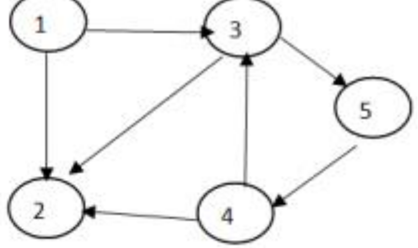


- Graphs are one of the most versatile structures used in computer programming
- The sorts of problems that graphs can help to solve are generally quite different
- A graph is a way of representing relationships that exist between pairs of objects
- A graph is a set of objects, called ***vertices***, together with a collection of pairwise connections between them, called ***edges***
- The circles are vertices, and the lines are edges. The vertices are usually labeled with alphabets or numbers.
- Graphs have applications in modeling many domains, including mapping, transportation, computer networks, and electrical engineering

Graph



Undirected graph

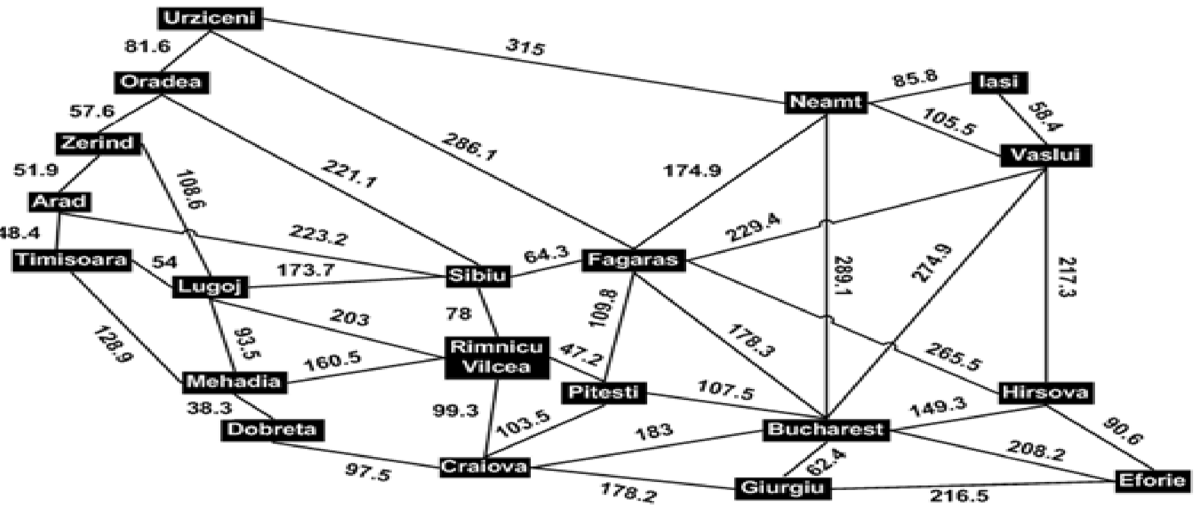


Directed graph.

- A graph ***G (V, E)*** is simply a set ***V*** of vertices and a collection ***E*** of pairs of vertices from ***V***, called edges
- Thus, a graph is a way of representing connections or relationships between pairs of objects from some set ***V***
- Edges in a graph are either *directed* or *undirected*
- An edge ***(u,v)*** is said to be directed from ***u*** to ***v*** if the pair ***(u,v)*** is ordered, with ***u*** preceding ***v***.
- An edge ***(u,v)*** is said to be undirected if the pair ***(u,v)*** is not ordered
- If all the edges in a graph are *undirected*, then we say the graph is an ***undirected graph***
- If all the edges in a graph are *directed*, then we say the graph is an ***directed graph***, also called a ***digraph***
- A graph that has both directed and undirected edges is often called a ***mixed graph***

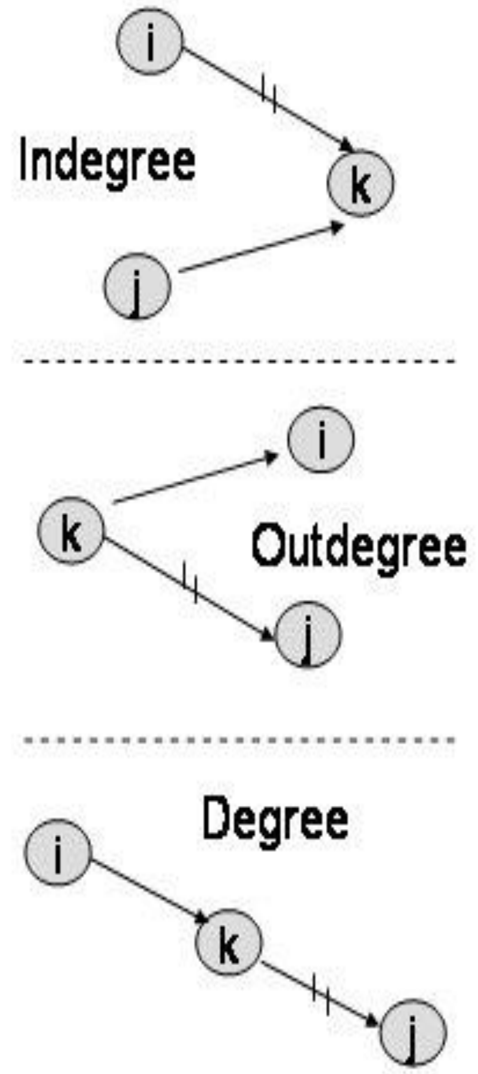
Graph Examples

- **A city map** can be modeled as a graph whose vertices are intersections or dead ends, and whose edges are stretches of streets without intersections. This graph has both undirected edges, which correspond to stretches of two-way streets, and directed edges, which correspond to stretches of one-way streets. Thus, in this way, a graph modeling a city map is a mixed graph



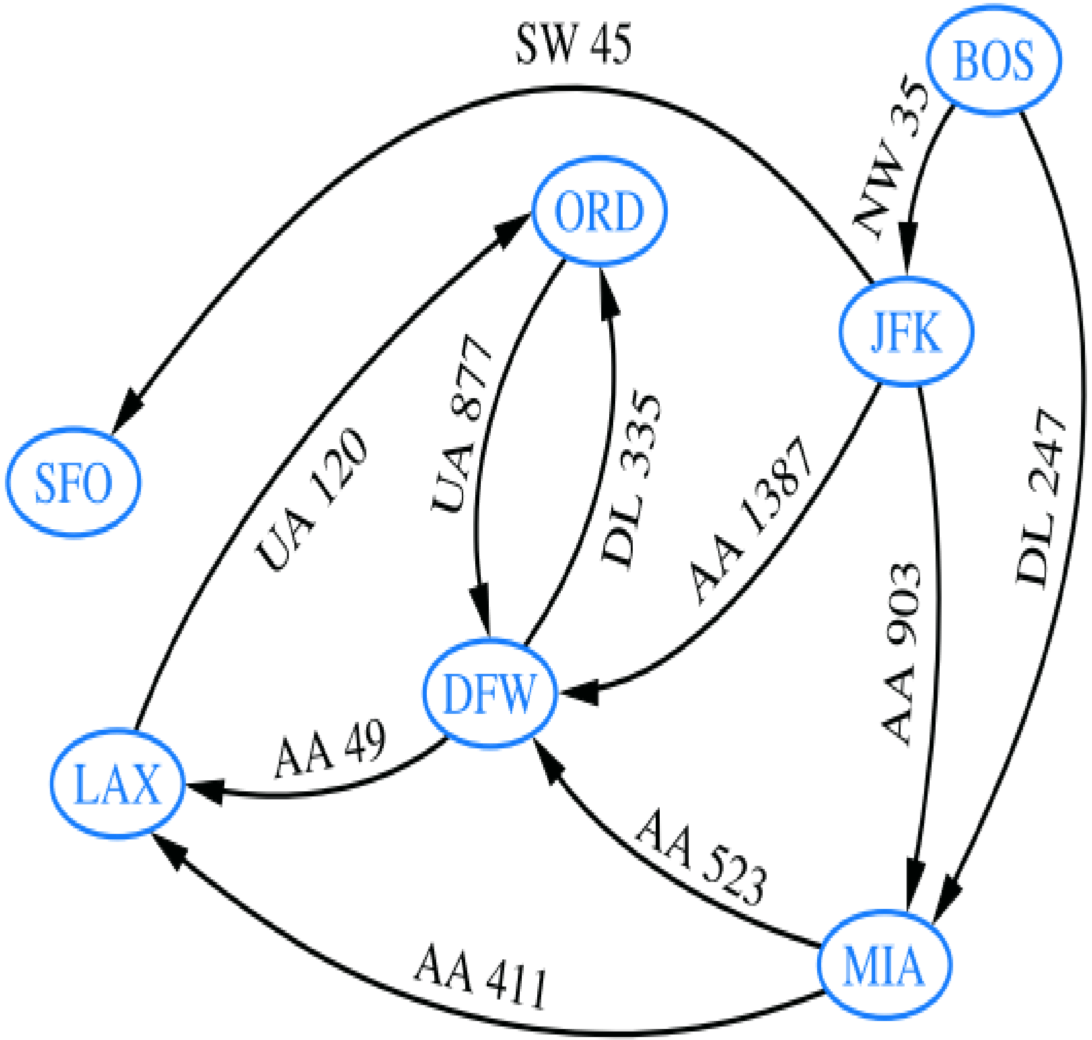
Graph

- The two vertices joined by an edge are called the end vertices (or ***endpoints***) of the edge
- If an edge is directed, its first end point is its ***origin*** and the other is the ***destination*** of the edge
- Two vertices ***u*** and ***v*** are said to be **adjacent** if there is an edge whose end vertices are ***u*** and ***v***
- An edge is said to be **incident** to a vertex if the vertex is one of the edge's endpoints
- The outgoing edges of a vertex are the directed edges whose origin is that vertex
- The incoming edges of a vertex are the directed edges whose destination is that vertex
- The degree of a vertex ***v***, denoted ***deg(v)***, is the number of incident edges of ***v***
- The in-degree and out-degree of a vertex ***v*** are the number of the incoming and outgoing edges of ***v***, and are denoted ***indeg(v)*** and ***outdeg(v)***, respectively

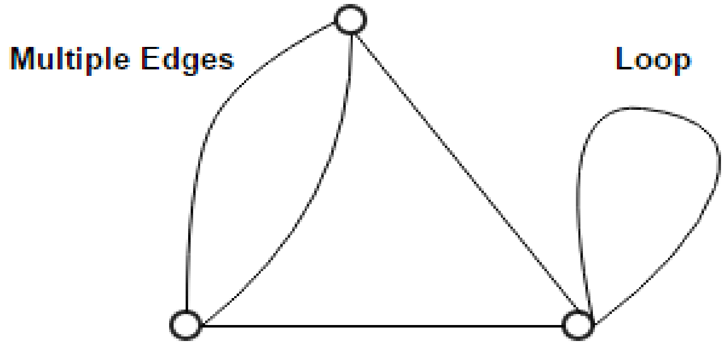


Graph Example

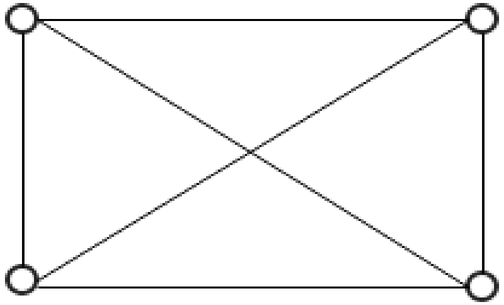
- We can study air transportation by constructing a graph ***G***, called a flight network, whose vertices are associated with airports, and whose edges are associated with flights
- Two airports are adjacent in ***G*** if there is a flight that flies between them, and an edge ***e*** is incident to a vertex ***v*** in ***G*** if the flight for ***e*** flies to or from the airport for ***v***
- The in-degree of a vertex ***v*** of ***G*** corresponds to the number of inbound flights to ***v***'s airport, and the out-degree of a vertex ***v*** in ***G*** corresponds to the number of out bound flights



Graph



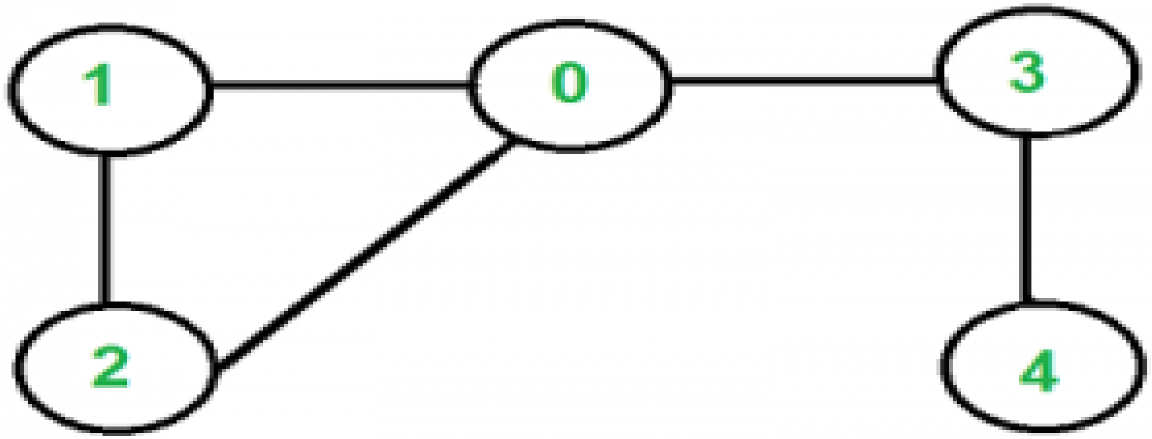
Not a Simple Graph



Simple Graph

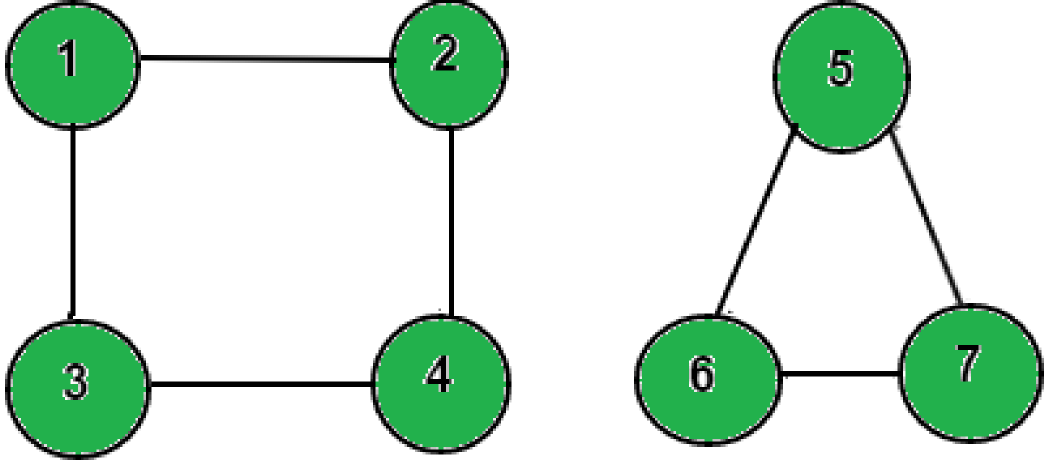
- The definition of a graph refers to the group of edges as a collection, not a set, thus allowing two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination
- Such edges are called **parallel edges** or **multiple edges**
- Another special type of edge is one that connects a vertex to itself
- Namely, we say that an edge (undirected or directed) is a **self-loop** if its two endpoints coincide
- Graphs do not have *parallel edges* or *self-loops* are said to be **simple graphs**

Graph



- A **path** is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex
- A **cycle** is a path that starts and ends at the same vertex, and that includes at least one edge
- We say that a *path* is **simple** if *each vertex in the path is distinct*
- We say that a *cycle* is **simple** if *each vertex in the cycle is distinct, except for the first and last one*
- A **directed path** is a path such that all edges are directed and are traversed along their direction
- A *directed graph* is **acyclic** if it has no directed cycles

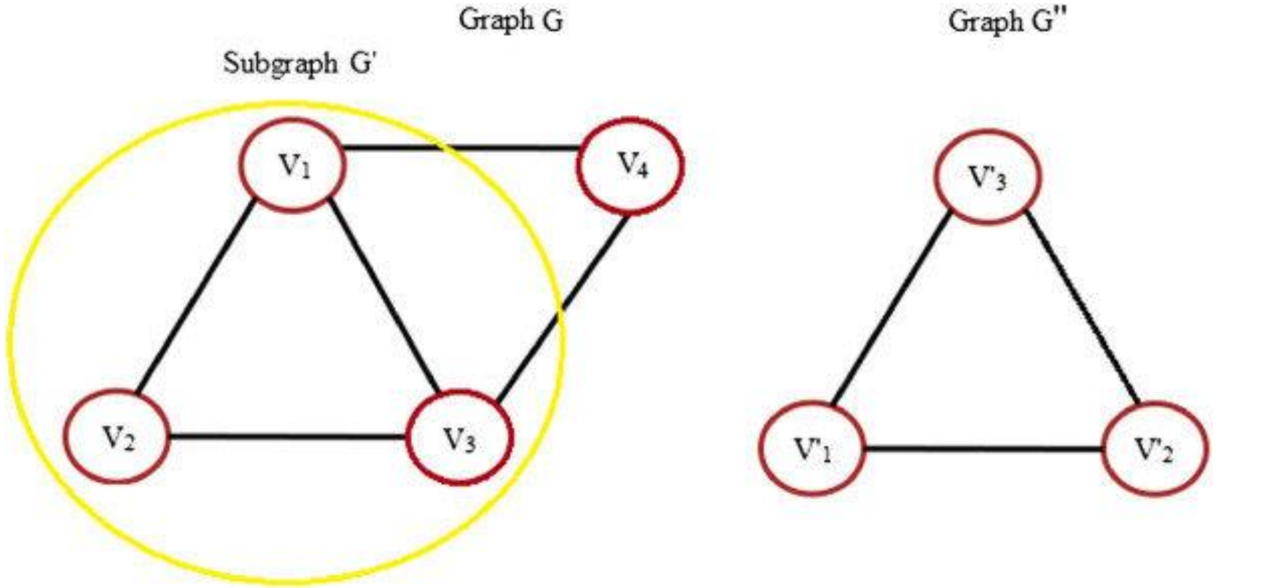
Graph

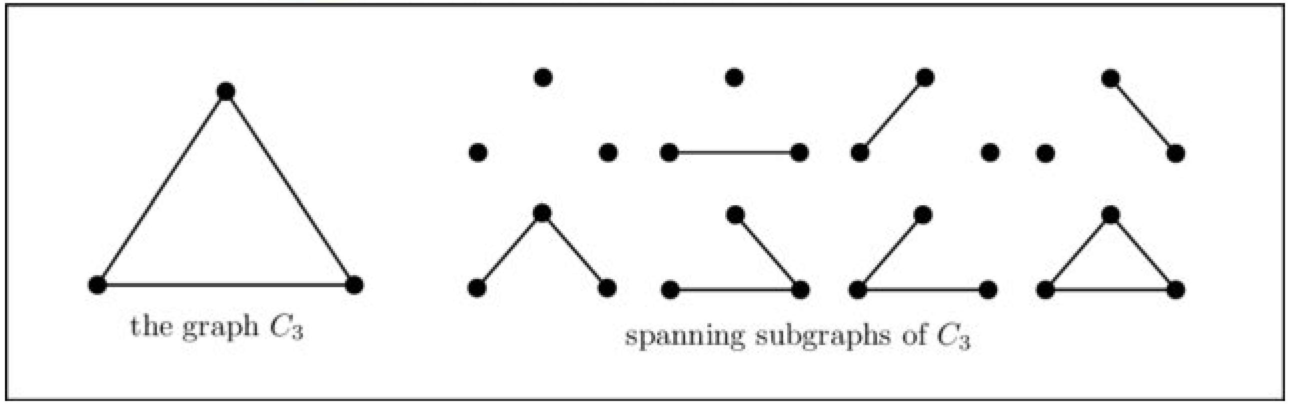


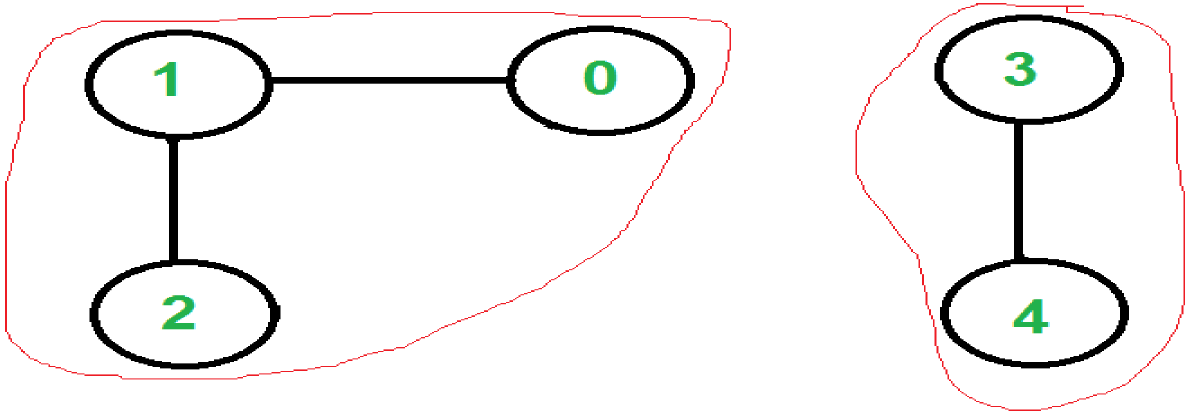
- Given vertices ***u*** and ***v*** of a (directed) graph ***G***, we say that ***u*** reaches ***v***, and that ***v*** is reachable from ***u***, if ***G*** has a (directed) **path** from ***u*** to ***v***
- In an undirected graph, the notion of reachability is symmetric, that is to say, ***u*** reaches ***v*** if an only if ***v*** reaches ***u***
- However, in a directed graph, it is possible that ***u*** reaches ***v*** but ***v*** does not reach ***u***, because a directed path must be traversed according to the respective directions of the edges
- A graph is **connected** if, for any two vertices, there is a path between them
- A directed graph G is **strongly connected** if for any two vertices ***u*** and ***v*** of ***G***, ***u*** reaches ***v*** and ***v*** reaches ***u***

Graph

- A **subgraph** of a graph ***G*** is a graph ***H*** whose vertices and edges are subsets of the vertices and edges of G, respectively
- A **spanning subgraph** of ***G*** is a subgraph of ***G*** that contains all the vertices of the graph ***G***
- If a graph ***G*** is not connected, its maximal connected subgraphs are called the connected components of ***G***
- A **forest** is a graph without cycles
- A **tree** is a connected forest, that is, a connected graph without cycles
- A **spanning tree** of a graph is a spanning subgraph that is a tree







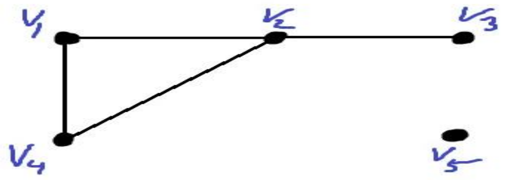
Graph

Proposition 14.8: *If G is a graph with m edges and vertex set V, then*

$$\sum_{v \text{ in } V} deg(v) = 2m.$$

Justification: An edge (u, v) is counted twice in the summation above; once by its endpoint u and once by its endpoint v. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges. ■

Proposition 14.9: *If G is a directed graph with m edges and vertex set V, then*



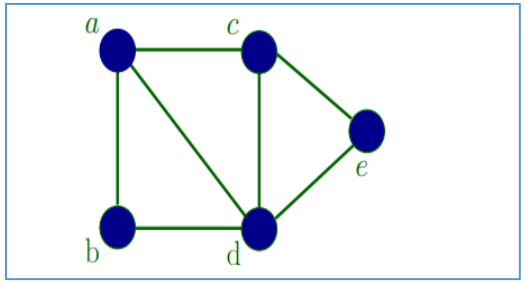
$deg(v_1) = 2$
$deg(v_2) = 3$
$deg(v_4) = 2$
$deg(v_3) = 1$
$deg(v_5) = 0$

$$\sum_{v \text{ in } V} indeg(v) = \sum_{v \text{ in } V} outdeg(v) = m.$$

Justification: In a directed graph, an edge (u, v) contributes one unit to the out-degree of its origin u and one unit to the in-degree of its destination v. Thus, the total contribution of the edges to the out-degrees of the vertices is equal to the number of edges, and similarly for the in-degrees. ■

Proposition 14.10: *Let G be a simple graph with n vertices and m edges. If G is undirected, then $m \leq n(n-1)/2$, and if G is directed, then $m \leq n(n-1)$.*

Graph

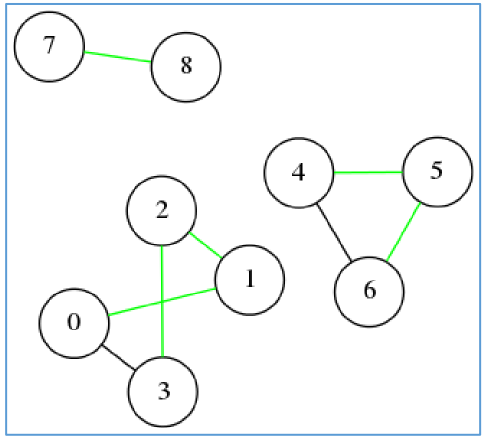


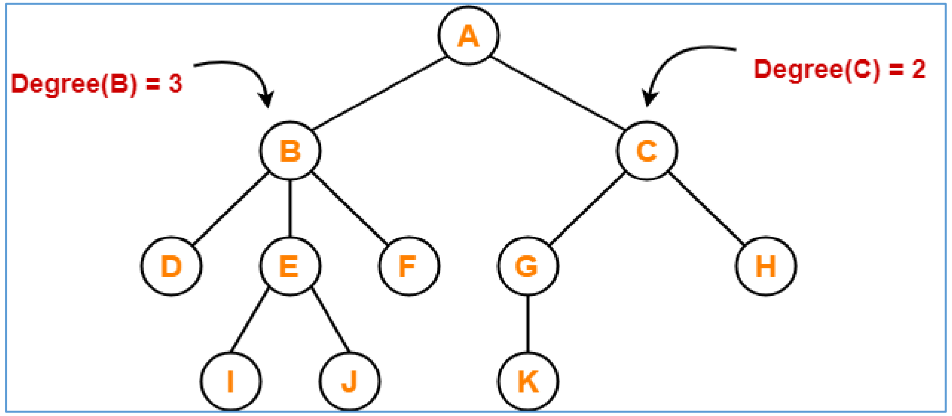
Justification: Suppose that G is undirected. Since no two edges can have the same endpoints and there are no self-loops, the maximum degree of a vertex in G is $n-1$ in this case. Thus, by Proposition 14.8, $2m \leq n(n-1)$. Now suppose that G is directed. Since no two edges can have the same origin and destination, and there are no self-loops, the maximum in-degree of a vertex in G is $n-1$ in this case. Thus, by Proposition 14.9, $m \leq n(n-1)$. ■

There are a number of simple properties of trees, forests, and connected graphs.

Proposition 14.11: *Let G be an undirected graph with n vertices and m edges.*

- *If G is connected, then $m \geq n-1$.*
- *If G is a tree, then $m = n-1$.*
- *If G is a forest, then $m \leq n-1$.*





Graph Traversals

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

Graph Traversals

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

Graph Traversals

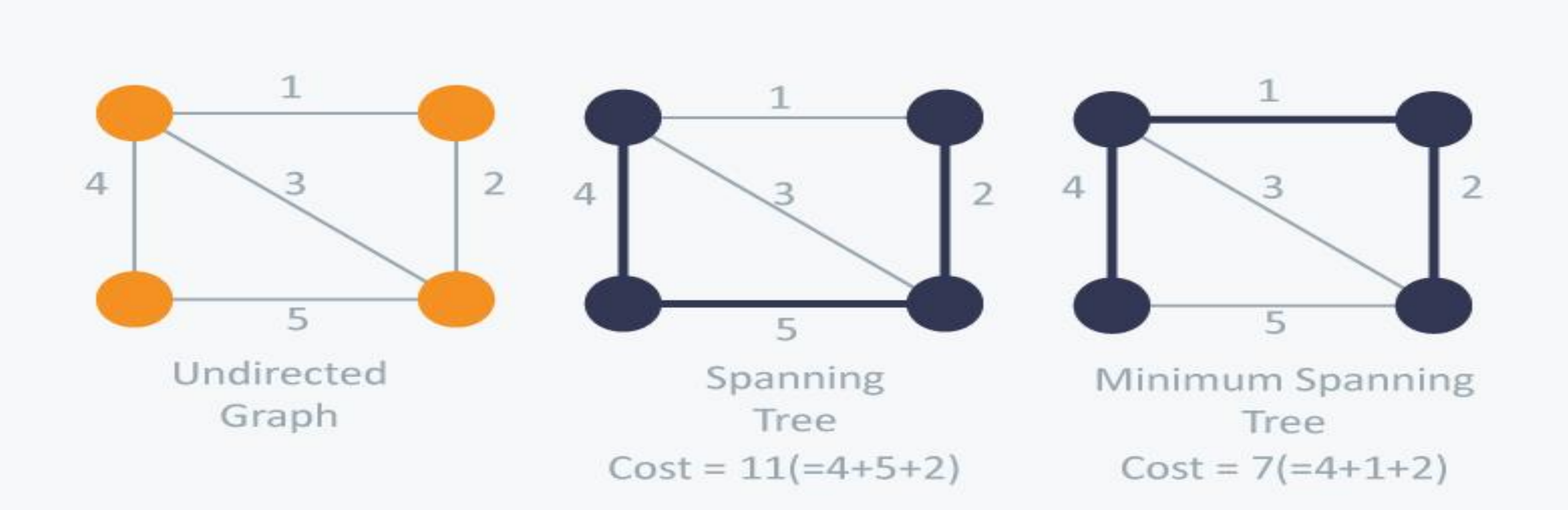
- Interesting problems that deal with reachability in a directed graph G include the following:
  - Computing a directed path from vertex u to vertex v, or reporting that no such path exists
  - Finding all the vertices of G that are reachable from a given vertex s
  - Determine whether G is acyclic
  - Determine whether G is strongly connected

Minimum Spanning Trees

- Given an undirected, weighted graph ***G***, we are interested in finding a tree ***T*** that contains all the vertices in ***G*** and minimizes the sum

$$w(T) = \sum_{(u,v) \text{ in } T} w(u,v).$$

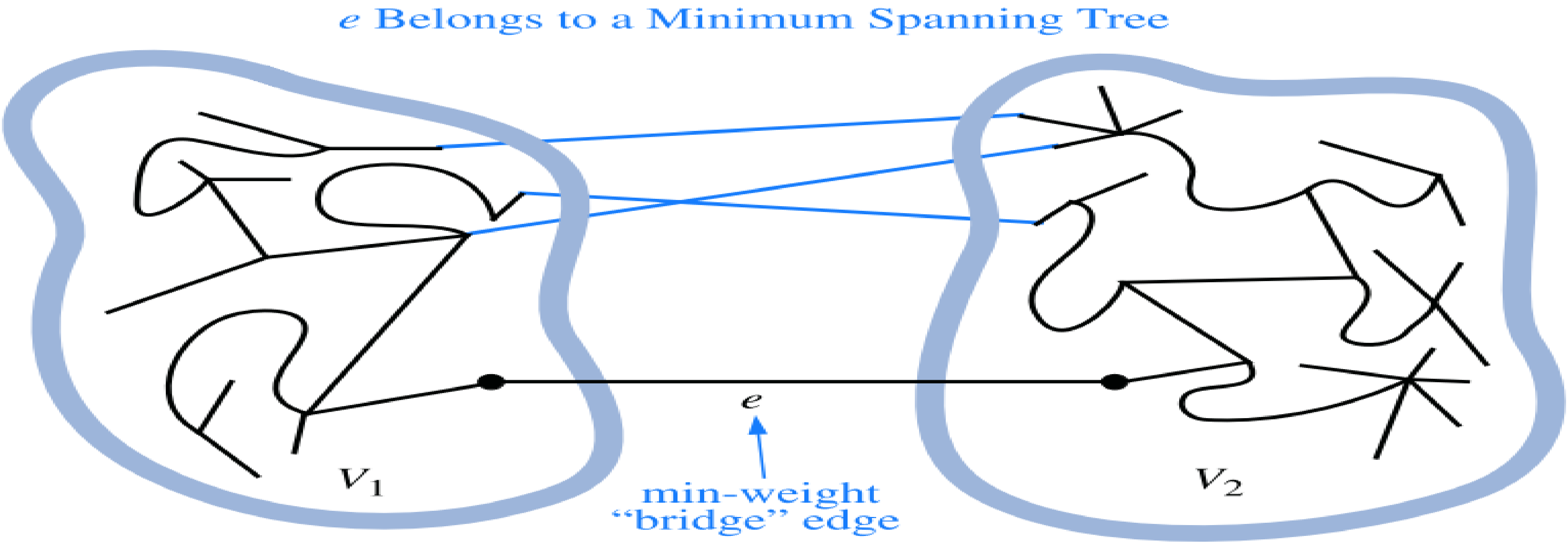
- A tree, such as this, that contains every vertex of a connected graph ***G*** is said to be a spanning tree, and the problem of computing a spanning tree ***T*** with smallest total weight is known as the minimum spanning tree (or MST) problem



A Crucial Fact about Minimum Spanning Trees

- The two MST algorithms we discuss are based on the greedy method, which in this case depends crucially on the following fact

Proposition 14.25: *Let G be a weighted connected graph, and let V_1 and V_2 be a partition of the vertices of G into two disjoint nonempty sets. Furthermore, let e be an edge in G with minimum weight from among those with one endpoint in V_1 and the other in V_2. There is a minimum spanning tree T that has e as one of its edges.*



Design and Analysis of Algorithm

Mahesh Shirole

VJTI, Mumbai-19

Shortest Paths

- In a **shortest-paths** problem, we are given a weighted, directed graph ***G =(V, E)***, with weight function ***w : E → R*** mapping edges to real-valued weights
- The **weight** ***w(p)*** of path ***p =<v₀, v₁, . . ., vₖ>*** is the sum of the weights of its constituent edges:

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

We define the ***shortest-path weight*** $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there is a path from } u \text{ to } v , \\ \infty & \text{otherwise .} \end{cases}$$

Single-Source Shortest Paths

- A shortest path from vertex ***u*** to vertex ***v*** is then defined as any path ***p*** with weight ***w(p)= δ(u, v)***
- 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 ϵ V*** to each vertex ***v ϵ V***
- *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, we can reduce this problem to a single-source problem
- *All-pairs shortest-paths* problem: Find a shortest path from ***u*** to ***v*** for every pair of vertices ***u*** and ***v***.

Optimal substructure of a shortest path

- Shortest-paths algorithms typically rely on the property that *a shortest path between two vertices contains other shortest paths within it*

Lemma 24.1 (Subpaths of shortest paths are shortest paths)
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then, p_{ij} is a shortest path from v_i to v_j.

Proof If we decompose path p into $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ■

Relaxation

RELAX(u, v, w)

```
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

INITIALIZE-SINGLE-SOURCE(G, s)

```
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

- Given a graph ***G =(V, E)***, for each vertex ***v ϵ V***, we maintain an attribute ***v.d***, which is an upper bound on the weight of a shortest path from source ***s*** to ***v***
- We call ***v.d*** a *shortest-path estimate*
- We also maintain for each vertex ***v ϵ V*** a *predecessor* ***v.π*** that is either another vertex or ***NIL***
- We **initialize the shortest-path** estimates and predecessors in ***θ(V)***-time procedure
- After initialization, we have ***v.π = NIL*** for all ***v ϵ V***, ***s.d = 0***,and ***v.d =∞*** for ***v ϵ V - {s}***
- The process of **relaxing** an edge ***(u, v)*** consists of testing whether we can improve the shortest path to ***v*** found so far by going through ***u*** and, if so, updating ***v.d*** and ***v.π***
- A relaxation step may decrease the value of the shortest-path estimate ***v.d*** and update **v's** predecessor attribute ***v.π***
- A relaxation step executes on edge ***(u, v)*** in ***O(1)*** time

Dijkstra’s algorithm

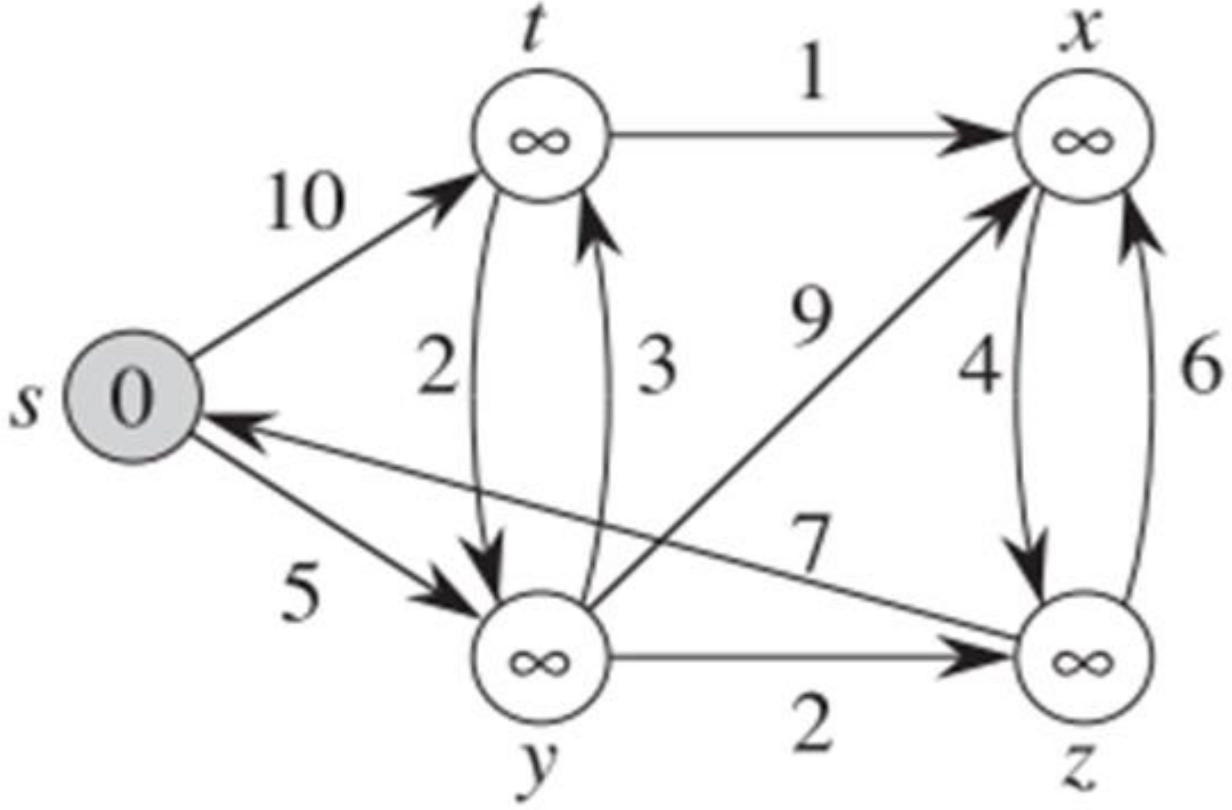
- Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph ***G =(V, E)*** for the case in which *all edge weights are nonnegative*
- Dijkstra’s algorithm maintains a set ***S*** of vertices whose *final shortest-path weights from the source **s** have already been determined*
- The algorithm repeatedly selects the vertex ***u ϵ V - S*** with the *minimum shortest-path estimate*, adds ***u*** to ***S***, and relaxes all edges leaving ***u***
- In the following implementation, we use a *min-priority queue* ***Q*** of vertices, keyed by their ***d*** values

Dijkstra's algorithm

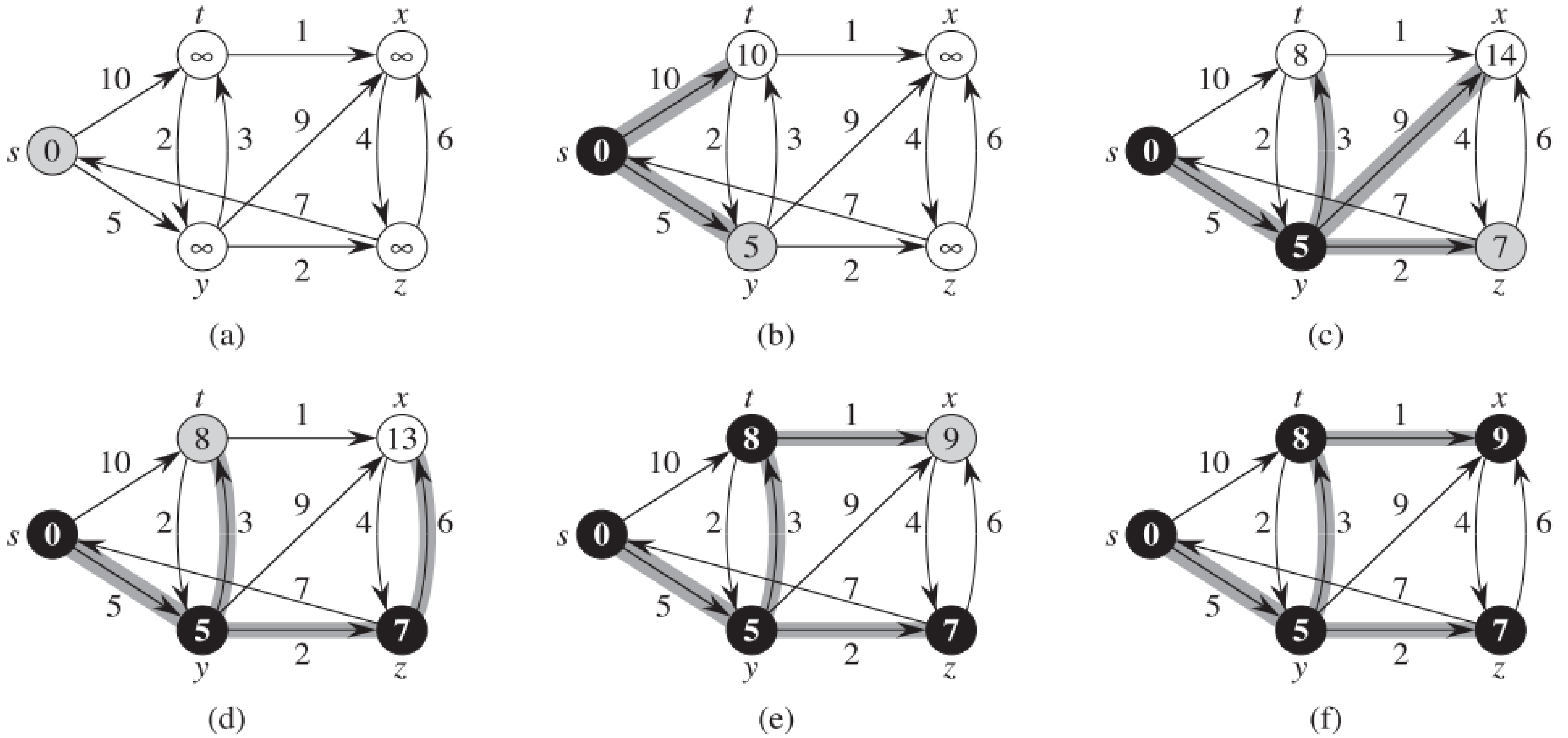
- Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph ***G =(V, E)*** for the case in which *all edge weights are nonnegative*
- Dijkstra's algorithm maintains a set ***S*** of vertices whose *final shortest-path weights from the source* ***s*** *have already been determined*
- The algorithm repeatedly selects the vertex ***u ϵ V - S*** with the *minimum shortest-path estimate*, adds ***u*** to ***S***, and relaxes all edges leaving ***u***
- In the following implementation, we use a *min-priority queue* ***Q*** of vertices, keyed by their ***d*** values

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

- Line 1 initializes the ***d*** and ***v*** values
- Line 2 initializes the set ***S*** to the empty set
- Line 3 initializes the min-priority queue ***Q*** to contain all the vertices in ***V***
- The while loop of lines 4–8, line 5 extracts a vertex ***u*** from ***Q = V-S*** and line 6 adds it to set ***S***
- lines 7–8 relax each edge ***(u, v)*** leaving ***u***, thus updating the estimate ***v.d*** and the predecessor ***v.π***


t
x
1
10
9
2
3
4
6
s
0
∞
∞
∞
∞
7
5
2
y
z

Dijkstra’s algorithm



Is Dijkstra's algorithm Greedy?

- Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in ***V* - *S*** to add to set ***S***
- Greedy strategies do not always yield optimal results in general
- The key is to show that each time it adds a vertex ***u*** to set ***S***, we have $u.d = \delta(s, u)$
- It uses a greedy strategy

Correctness of Dijkstra's algorithm

Theorem 24.6 (Correctness of Dijkstra's algorithm)
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with non-negative weight function w and source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

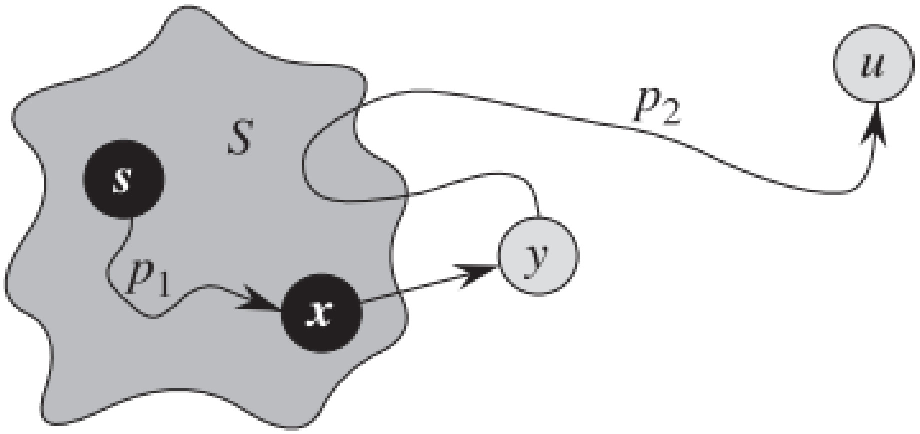


Figure 24.7 The proof of Theorem 24.6. Set S is nonempty just before vertex u is added to it. We decompose a shortest path p from source s to vertex u into $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, where y is the first vertex on the path that is not in S and $x \in S$ immediately precedes y. Vertices x and y are distinct, but we may have $s = x$ or $y = u$. Path p_2 may or may not reenter set S.

Correctness of Dijkstra's algorithm

- It suffices to show for each vertex ***u* ϵ *V***, we have ***u.d* = *δ*(*s*, *u*)** at the time when ***u*** is added to set ***S***
- Once we show that ***u.d* = *δ*(*s*, *u*)**, we rely on the upper-bound property to show that the equality holds at all times thereafter
- Initially, ***S* =*Φ***, and so the invariant is trivially true
- Let ***u*** be the first vertex for which ***u.d* ≠ *δ*(*s*, *u*)** when it is added to set ***S***
- We must have ***u* ≠ *s*** because ***s*** is the first vertex added to set ***S*** and ***s.d* = *δ*(*s*, *s*)=0** at that time
- There must be some path from ***s*** to ***u***, for otherwise ***u.d* = *δ*(*s*, *u*)=∞** by the no-path property
- Because there is at least one path, there is a shortest path ***p*** from ***s*** to ***u***
- Prior to adding ***u*** to ***S***, path ***p*** connects a vertex in ***S***, namely ***s***, to a vertex in ***V* - *S***, namely ***u***
- Let us consider the first vertex ***y*** along ***p*** such that ***y* ϵ *V* - *S***,and let ***x* ϵ *S*** be ***y***'s predecessor along ***p***
- Thus, we can decompose path ***p*** into $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u.$

Correctness of Dijkstra's algorithm

- We claim that ***y.d = δ(s, y)*** when ***u*** is added to ***S***
- To prove this claim, observe that ***x ϵ S***
- Then, because we chose ***u*** as the first vertex for which ***u.d ≠ δ(s, u)*** when it is added to ***S***, we had ***x.d = δ(s, x)*** when ***x*** was added to ***S***
- Edge ***(x, y)*** was relaxed at that time, and the claim follows from the convergence property
- We can now obtain a contradiction to prove that ***u.d ≠ δ(s, u)***
- Because ***y*** appears before ***u*** on a shortest path from ***s*** to ***u*** and all edge weights are nonnegative ***δ(s, y) <= δ(s, u)***
- Hence,

$$
\begin{aligned}
y.d &= \delta(s, y) \\
&\leq \delta(s, u) \\
&\leq u.d \qquad \text{(by the upper-bound property)}
\end{aligned}
$$

- Thus $y.d = \delta(s, y) = \delta(s, u) = u.d$

Analysis of Dijkstra’s algorithm

- The running time of Dijkstra’s algorithm depends on the min-priority queue
- Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered **1** to **|V|**
- We simply store ***v.d*** in the **v^th^** entry of an array
- Each **INSERT** and **DECREASE-KEY** operation takes ***O(1)*** time, and each **EXTRACT-MIN** operation takes ***O(V)*** time(since we have to search through the entire array), for a total time of $O(V^2 + E) = O(V^2)$
- Dijkstra's algorithm with $O(V^2)$ when implemented using an unsorted array and no priority queue
- Time Complexity of Dijkstra's Algorithm is $O(V^2)$ but with min-priority queue it drops down to ***O(V+E logV)***

Minimum-Spanning-Tree Algorithms

- Kruskal’s algorithm
  - The set ***A*** is a forest whose vertices are all those of the given graph
  - The safe edge added to ***A*** is always *a least-weight edge* in the graph that connects two distinct components
- Prim’s algorithm
  - The set ***A*** forms a single tree
  - The safe edge added to ***A*** is always *a least-weight edge* connecting the tree to a vertex not in the tree

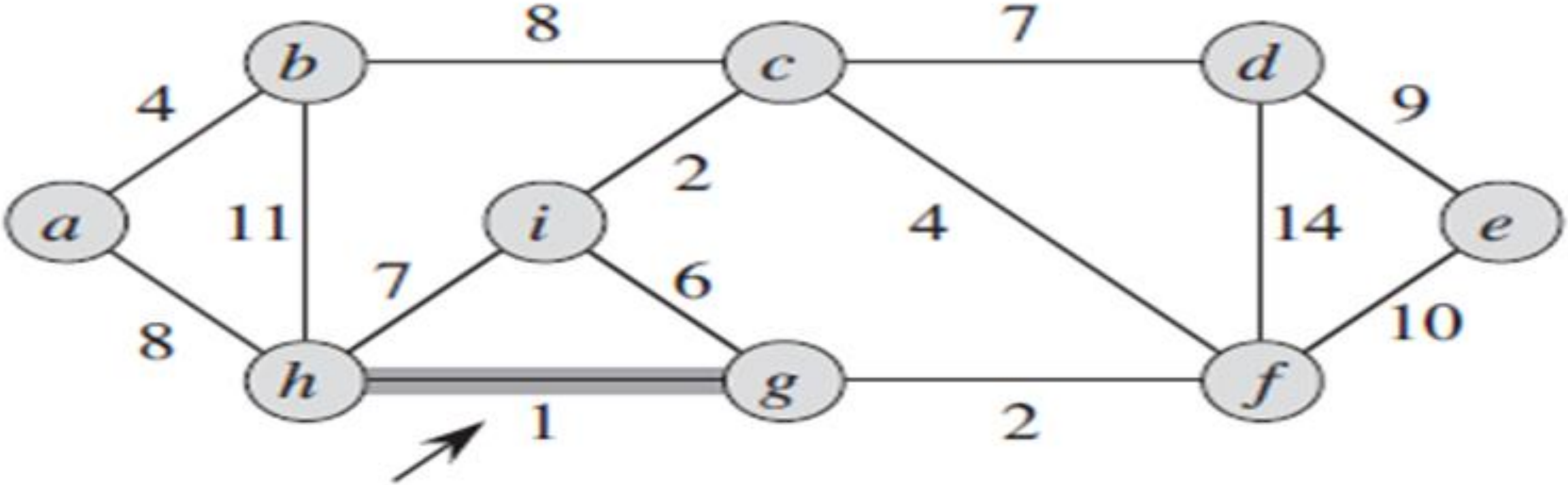
Kruskal's algorithm

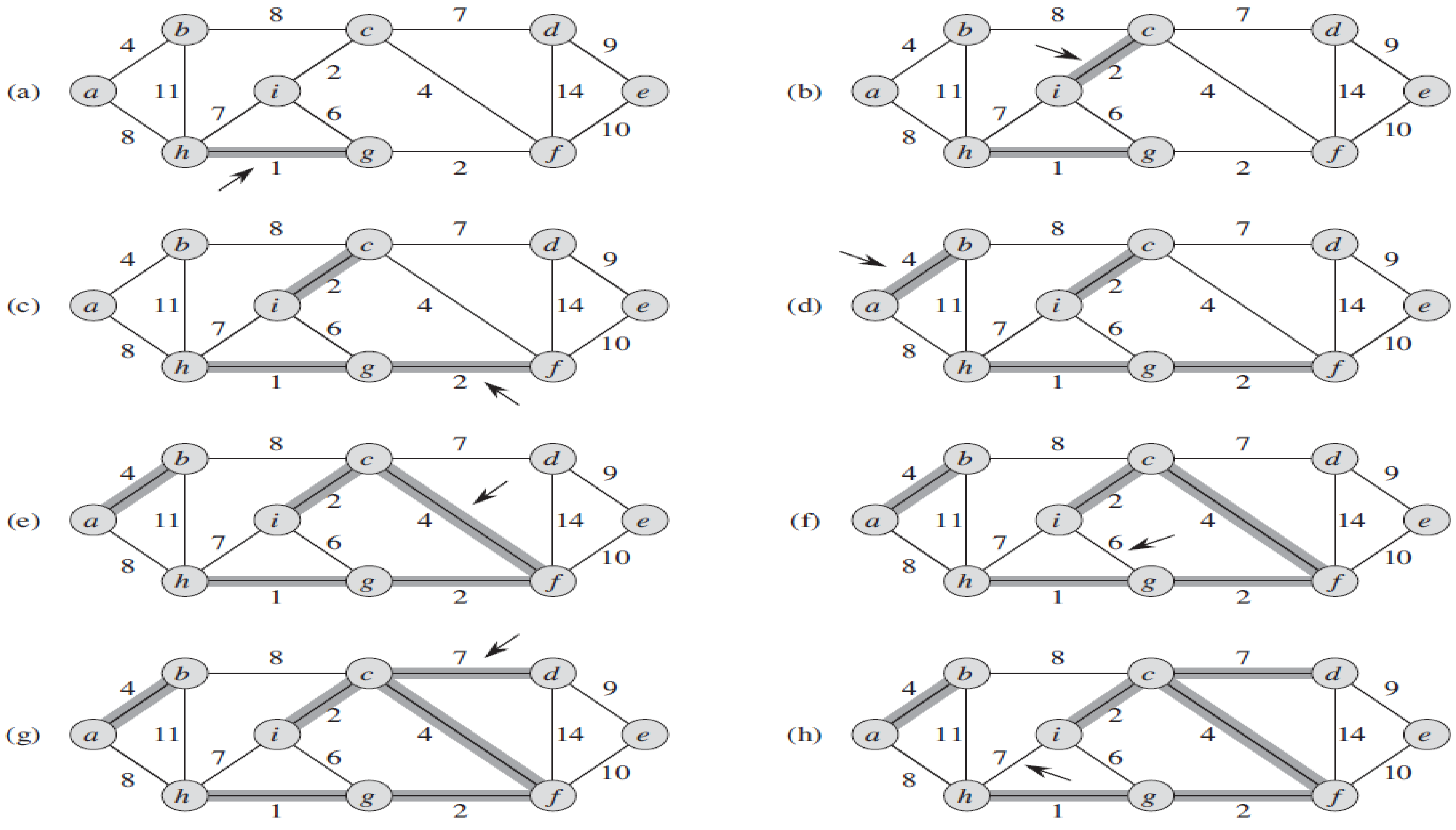
- Kruskal's algorithm *finds a safe edge* to add to the *growing forest* by finding, of all the edges that connect any two trees in the forest, an edge ***(u, v)*** of least weight
- Let ***C1*** and ***C2*** denote the two trees that are connected by ***(u, v)***
- Since ***(u, v)*** must be a light edge connecting ***C1*** to some other tree, implies that ***(u, v)*** is a safe edge for ***C1***
- Kruskal's algorithm qualifies as a **greedy algorithm** because at each step it adds to the forest an edge of least possible weight
- Kruskal's algorithm uses a **disjoint-set** data structure to maintain several disjoint sets of elements
- Each set contains the vertices in one tree of the current forest
- The operation **FIND-SET(*u*)** returns a representative element from the set that contains ***u***
- Thus, we can determine whether two vertices ***u*** and ***v*** belong to the same tree by testing whether **FIND-SET(*u*)** equals **FIND-SET(*v*)**
- To combine trees, Kruskal's algorithm calls the **UNION** procedure

Kruskal's algorithm

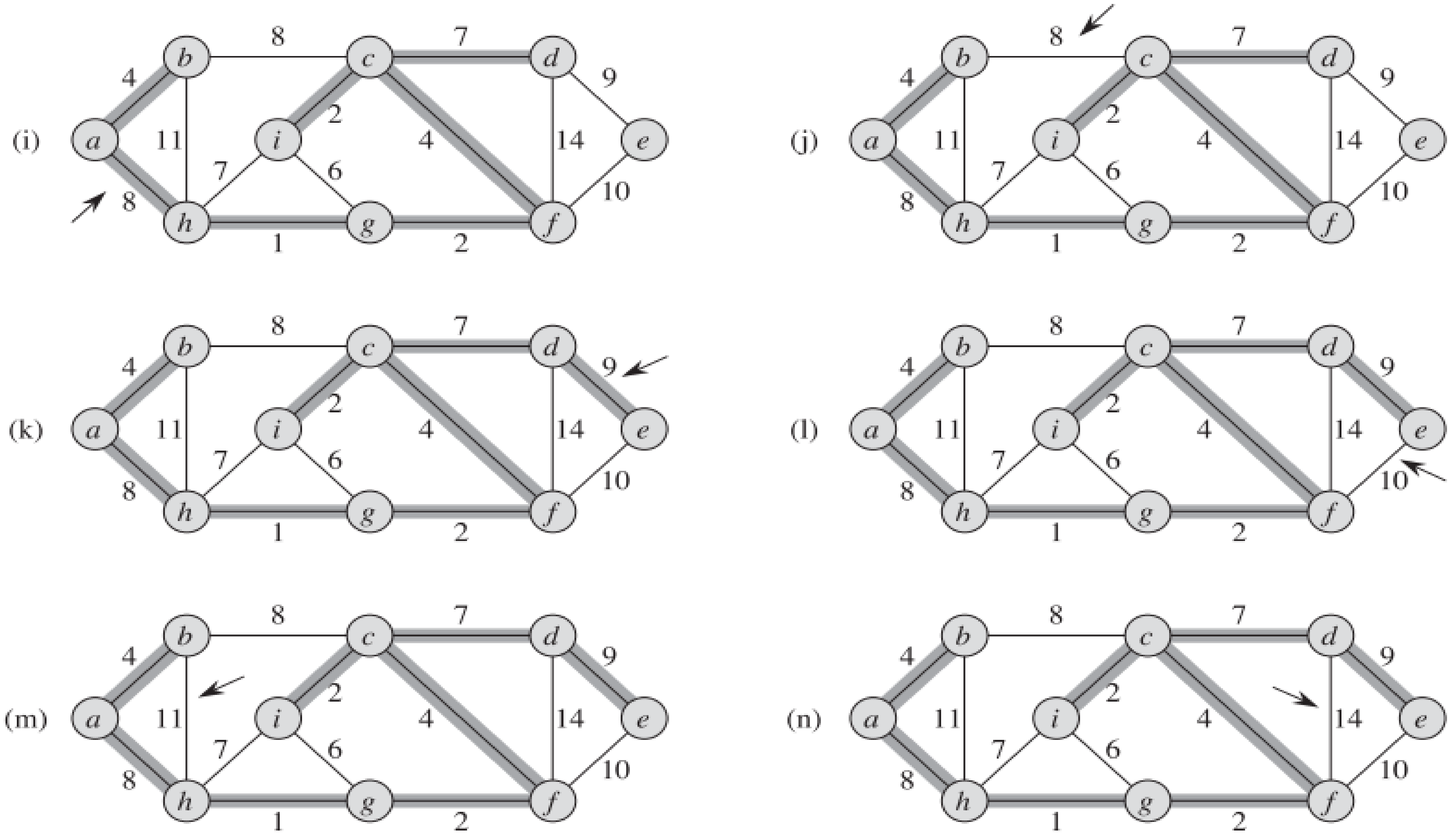
```
MST-KRUSKAL(G, w)
1  A = Ø
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return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

- Lines 1–3 initialize the set ***A*** to the empty set and create ***|V|*** trees, one containing each vertex
- The for loop in lines 5–8 examines edges in order of weight, from lowest to highest
- The loop checks, for each edge ***(u, v)***, whether the endpoints ***u*** and ***v*** belong to the same tree
    - If they do, then the edge ***(u, v)*** cannot be added to the forest without creating a cycle, and the edge is discarded
    - Otherwise, the two vertices belong to different trees
- In this case, line 7 adds the edge ***(u, v)*** to ***A***, and line 8 merges the vertices in the two trees


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



(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)



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

Analysis of Kruskal's algorithm

- The running time of Kruskal's algorithm for a graph ***G =(V, E)*** depends on how we implement the disjoint-set data structure
- We assume that we use the disjoint-set-forest implementation with the **union-by-rank** and **path-compression heuristics**, since it is the asymptotically fastest implementation known
- Initializing the set ***A*** in line 1 takes ***O(1)*** time, and the time to sort the edges in line 4 is ***O(E lg E)***
- The for loop of lines 5–8 performs ***O(E)*** **FIND-SET** and **UNION** operations on the disjoint-set forest
- Along with the ***|V|*** **MAKE-SET** operations, these take a total of ***O((V + E) α(V))*** time, where ***α*** is the very slowly growing function
- Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as ***O(E lg V)***

Design and Analysis of Algorithm

Mahesh Shirole

VJTI, Mumbai-19

Minimum-Spanning-Tree Algorithms

- Kruskal's algorithm
  - The set A is a forest whose vertices are all those of the given graph
  - The safe edge added to A is always a least-weight edge in the graph that connects two distinct components
- Prim's algorithm
  - The set A forms a single tree
  - The safe edge added to A is always a least-weight edge connecting the tree to a vertex not in the tree

Prim's algorithm

- Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph
- 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*** a light edge that connects ***A*** to an isolated vertex—one on which no edge of ***A*** is incident
- This strategy qualifies as greedy since at each step it adds to the tree *an edge that contributes the minimum amount possible to the tree's weight*

Prim's algorithm

- In order to implement Prim's algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in ***A***
- All vertices that are not in the tree reside in a min-priority queue ***Q*** based on a key attribute
- For each vertex ***v***, the attribute ***v.key*** is the minimum weight of any edge connecting ***v*** to a vertex in the tree; by convention, ***v.key = ∞*** if there is no such edge
- The attribute ***v.π*** names the parent of ***v*** in the tree
- The algorithm implicitly maintains the set ***A*** from GENERIC-MST as

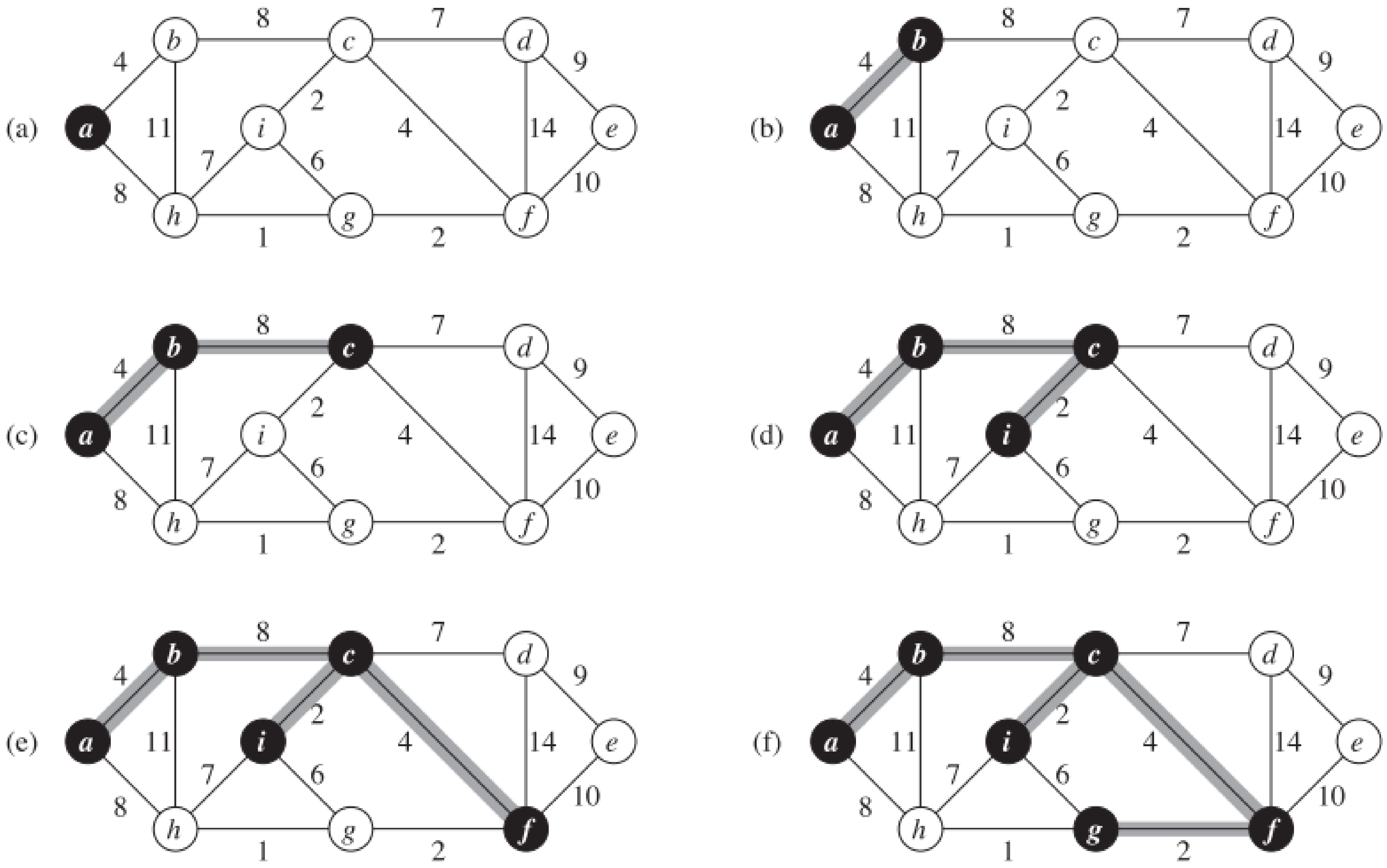
MST-PRIM(G, w, r)

```
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

$$A = \{(v, v.\pi) : v \in V - \{r\} - Q\} .$$

When the algorithm terminates, the min-priority queue Q is empty; the minimum spanning tree A for G is thus

$$A = \{(v, v.\pi) : v \in V - \{r\}\} .$$


(a)
(b)
(c)
(d)
(e)
(f)

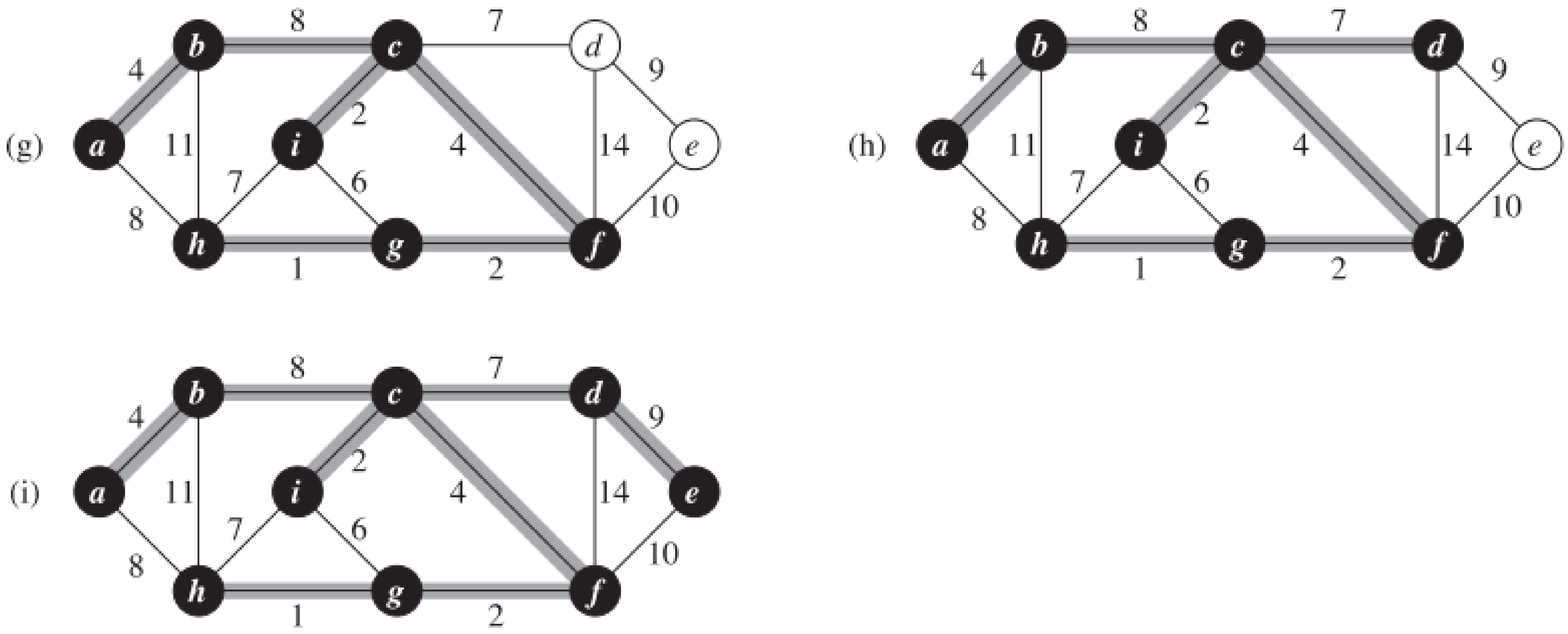


Figure 23.5 The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Analysis of Prim's algorithm

- The running time of Prim's algorithm depends on how we implement the minpriority queue ***Q***
- If we implement ***Q*** as a binary min-heap, we can use the **BUILD-MIN-HEAP** procedure to perform lines 1–5 in ***O(V)*** time
- The body of the while loop executes ***|V|*** times, and since each **EXTRACT-MIN** operation takes ***O(lg V)*** time, the total time for all calls to **EXTRACT-MIN** is ***O(V lg V)***
- The for loop in lines 8–11 executes ***O(E)*** times altogether, since the sum of the lengths of all adjacency lists is ***2 |E|***
- The assignment in line 11 involves an implicit **DECREASE-KEY** operation on the min-heap, which a binary min-heap supports in ***O(lg V)*** time
- Thus, the total time for Prim's algorithm is

O(V lg V + E lg V) = O(E lg V), which is asymptotically the same as for our implementation of Kruskal's algorithm

MST-PRIM(G, w, r)

```
1   for each u ∈ G.V
2       u.key = ∞
3       u.π = NIL
4   r.key = 0
5   Q = G.V
6   while Q ≠ ∅
7       u = EXTRACT-MIN(Q)
8       for each v ∈ G.Adj[u]
9           if v ∈ Q and w(u, v) < v.key
10              v.π = u
11              v.key = w(u, v)
```

All-Pairs Shortest Paths

- We consider the problem of finding shortest paths between all pairs of vertices in a graph
- This problem might arise in making a table of distances between all pairs of cities for a road atlas
- we are given a weighted, directed graph $G = (V, E)$ with a weight function $w : E \rightarrow R$ that maps edges to real-valued weights
- We wish to find, for every pair of vertices $u, v \in V$, a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges
- We typically want the output in tabular form: the entry in u's row and v's column should be the weight of a shortest path from u to v
- Unlike the single-source algorithms, which assume an *adjacency-list representation* of the graph, we use an *adjacency-matrix representation* for all pair shortest path algorithm

All-Pairs Shortest Paths

- For convenience, we assume that the vertices are numbered ***1, 2, ..., |V|***, so that the input is an ***n X n*** matrix ***W*** representing the edge weights of an ***n-vertex*** directed graph ***G = (V, E)***
- That is, ***W = (w_{ij})***, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \text{ ,} \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \text{ ,} \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \text{ .} \end{cases}$$

- The tabular output of the all-pairs shortest-paths algorithms presented an ***n X n*** matrix ***D = (d_{ij})*** , where entry ***d_{ij}*** contains the weight of a shortest path from vertex ***i*** to vertex ***j***

The structure of a shortest path

- For the all-pairs shortest-paths problem on a graph ***G = (V, E)***, we have proven that all subpaths of a shortest path are shortest paths
- Suppose that we represent the graph by an adjacency matrix $W = (w_{ij})$
- Consider a shortest path ***p*** from vertex ***i*** to vertex ***j*** , and suppose that ***p*** contains at most ***m*** edges
- Assuming that there are no negative-weight cycles, ***m*** is finite
- If ***i = j*** ,then ***p*** has weight ***0*** and no edges
- If vertices ***i*** and ***j*** are distinct, then we decompose path ***p*** into

  $i \overset{p'}{\leadsto} k \rightarrow j$, where path p' now contains at most $m - 1$ edges.
- ***p'*** is a shortest path from ***i*** to ***k***, and so $\delta(i, j) = \delta(i, k) + w_{kj}$

A recursive solution to the all-pairs shortest-paths problem

- Now, let ***$l_{ij}^{(m)}$*** be the minimum weight of any path from vertex ***i*** to vertex ***j*** that contains at most ***m*** edges
- When ***m = 0***, there is a shortest path from ***i*** to ***j*** with no edges if and only if ***i = j***. Thus

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \,, \\ \infty & \text{if } i \neq j \,. \end{cases}$$

For $m \geq 1$, we compute $l_{ij}^{(m)}$ as the minimum of $l_{ij}^{(m-1)}$ (the weight of a shortest path from i to j consisting of at most $m-1$ edges) and the minimum weight of any path from i to j consisting of at most m edges, obtained by looking at all possible predecessors k of j. Thus, we recursively define

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1\leq k\leq n} \{l_{ik}^{(m-1)} + w_{kj}\}\right) \\ &= \min_{1\leq k\leq n} \{l_{ik}^{(m-1)} + w_{kj}\} \,. \end{aligned} \tag{25.2}$$

The Floyd-Warshall algorithm

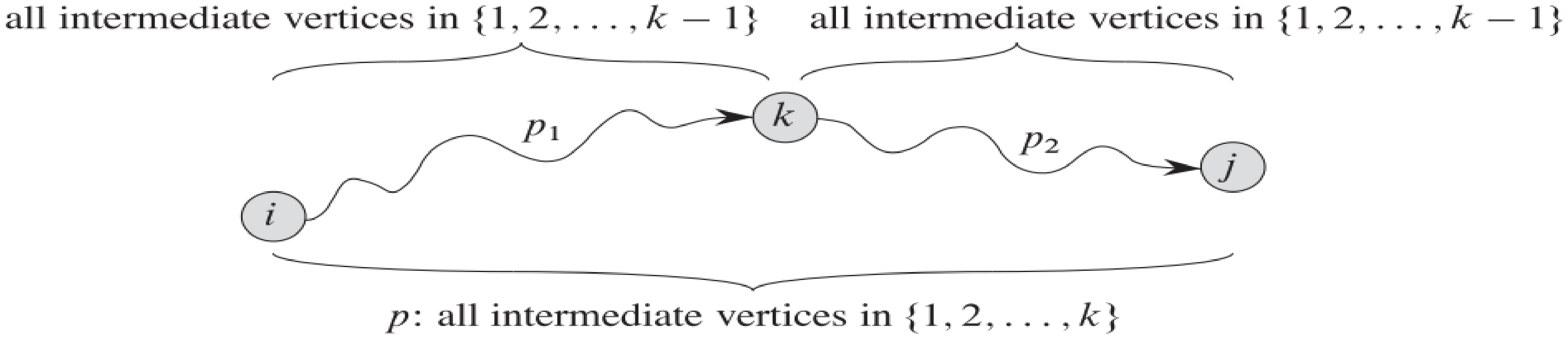
- A dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$
- The resulting algorithm, known as the **Floyd-Warshall** algorithm, runs in $O(V^3)$ time.

The structure of a shortest path

- The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path ***$p = \langle v_1, v_2, \ldots, v_l \rangle$*** is any vertex of ***$p$*** other than ***$v_1$*** or ***$v_l$***, that is, any vertex in the set ***$\{v_2, v_3, \ldots, v_{l-1}\}$***
- The Floyd-Warshall algorithm exploits a relationship between path ***p*** and shortest paths from ***i*** to ***j*** with all intermediate vertices in the set ***$\{1, 2, \ldots, k-1\}$***. The relationship depends on whether or not ***k*** is an intermediate vertex of path ***p***
- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $\{1, 2, \ldots, k-1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from i to j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

The structure of a shortest path

- If k is an intermediate vertex of path p, then we decompose p into $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$, as Figure 25.3 illustrates. By Lemma 24.1, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. In fact, we can make a slightly stronger statement. Because vertex k is not an intermediate vertex of path p_1, all intermediate vertices of p_1 are in the set $\{1, 2, \ldots, k-1\}$. Therefore, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Similarly, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.



A recursive solution to the all-pairs shortest-paths problem

- Let $d^{(k)}_{ij}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$
- When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all
- Such a path has at most one edge, and hence $d^{(0)}_{ij} = w_{ij}$

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases} \tag{25.5}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

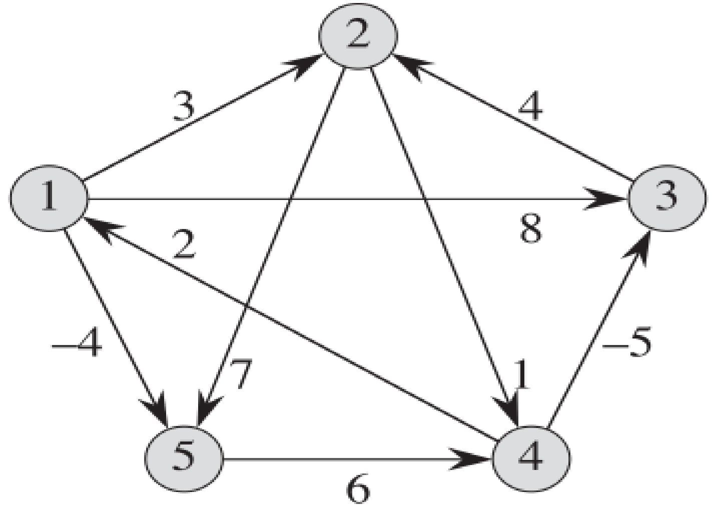
Computing the shortest-path weights bottom up

- Based on recurrence (25.5), we can use the following bottom-up procedure to compute the values $d^{(k)}_{ij}$ in order of increasing values of k
- Its input is an $n \times n$ matrix W
- The procedure returns the matrix $D^{(n)}$ of shortest path weights

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d^{(k)}_{ij}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d^{(k)}_{ij} = \min\left(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}\right)$
return $D^{(n)}$

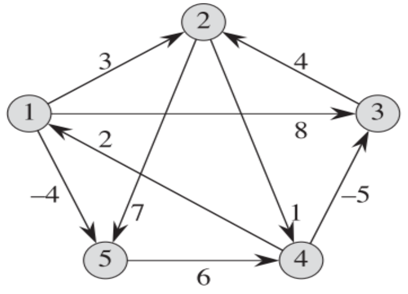
Example



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Example



$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$