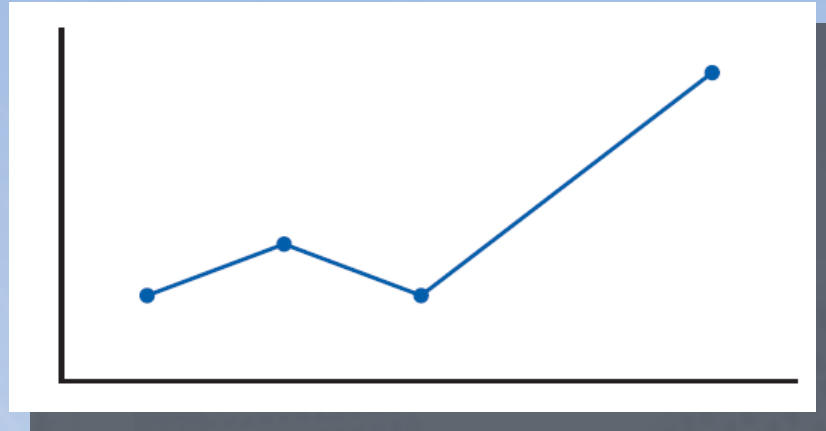


Graphs

Chapter 20

Terminology

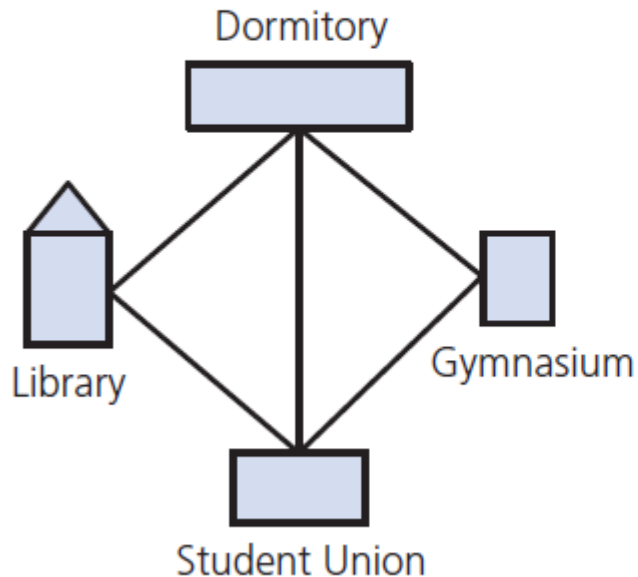
FIGURE 20-1 An ordinary line graph



- In the context of this book, graphs represent relations among data items
- $G = \{ V, E \}$
 - A graph is a set of vertices (nodes) and
 - A set of edges that connect the vertices

Terminology

(a) A campus map as a graph



(b) A subgraph of the graph in part a

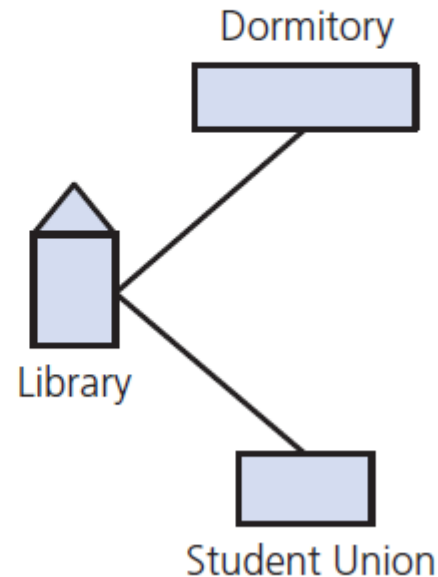


FIGURE 20-2 A graph and one of its subgraphs

Terminology

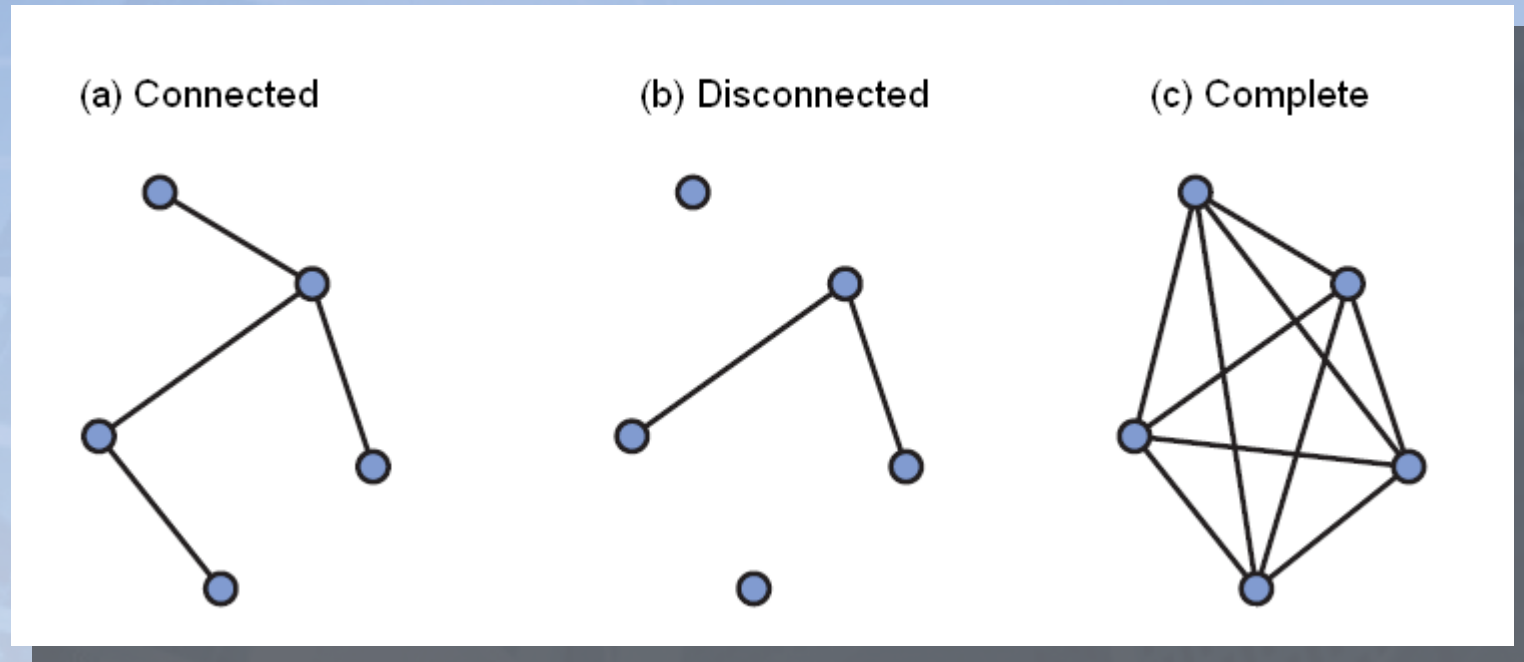


FIGURE 20-3 Examples of graphs that are either connected, disconnected, or complete

Terminology

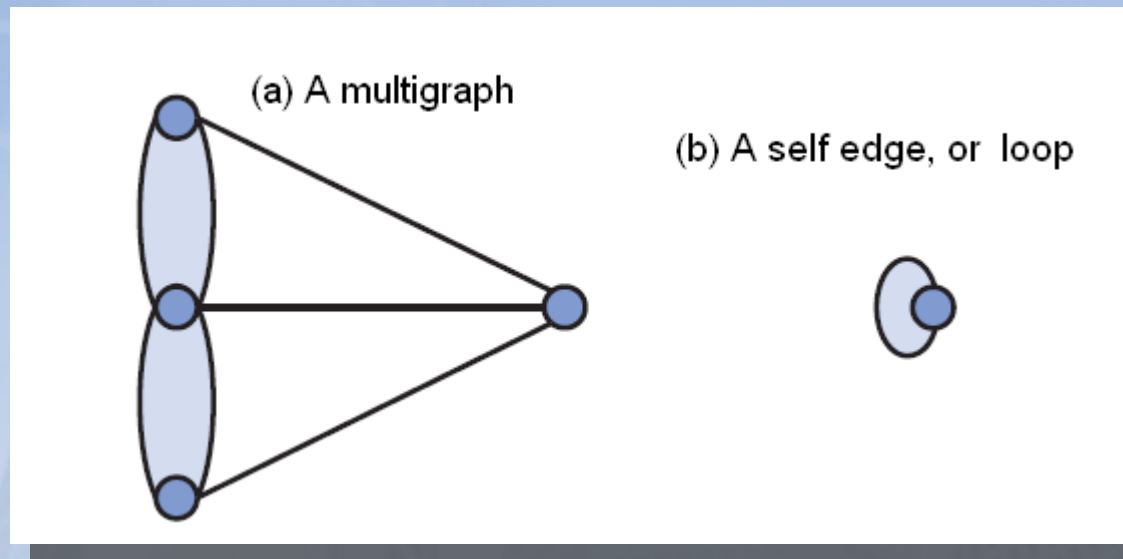


FIGURE 20-4 Graph-like structures that are not graphs

Terminology

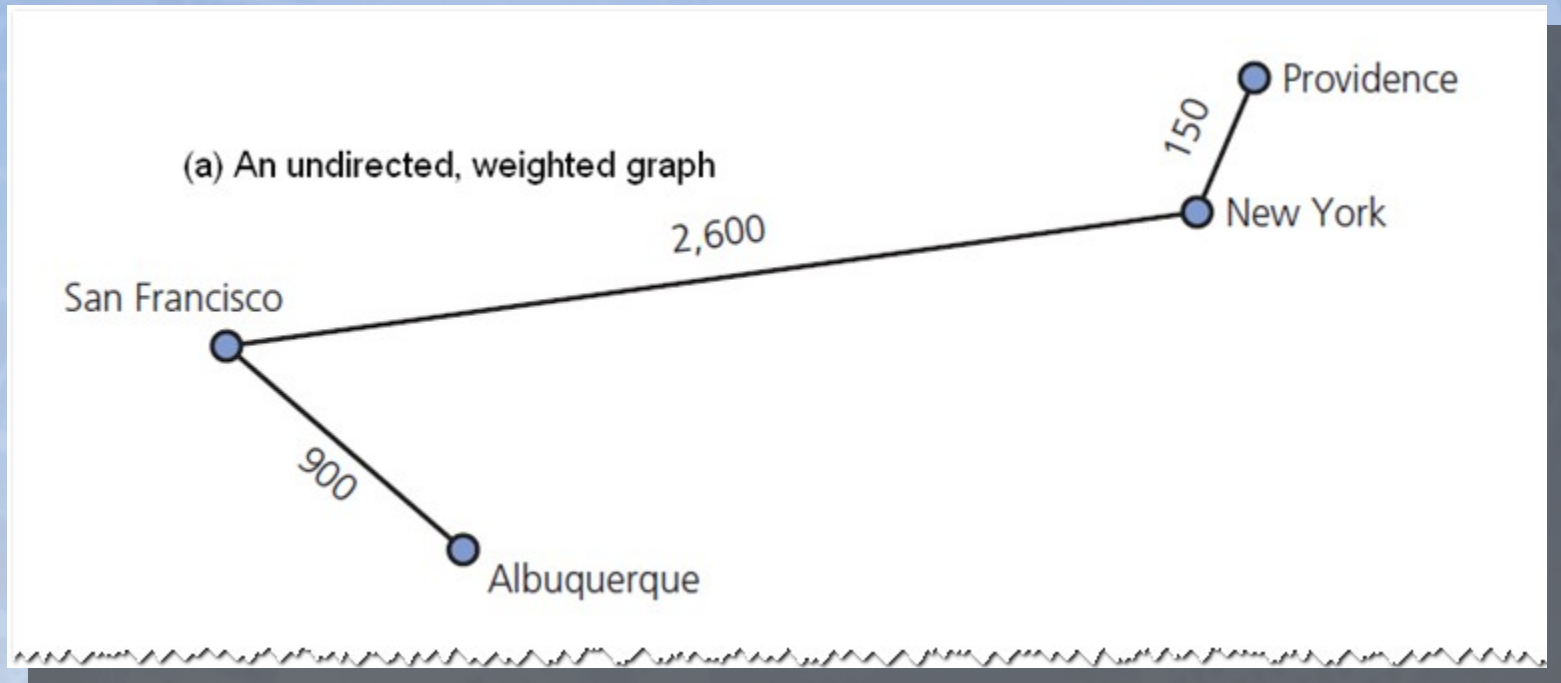


FIGURE 20-5 Examples of two kinds of graphs

Terminology

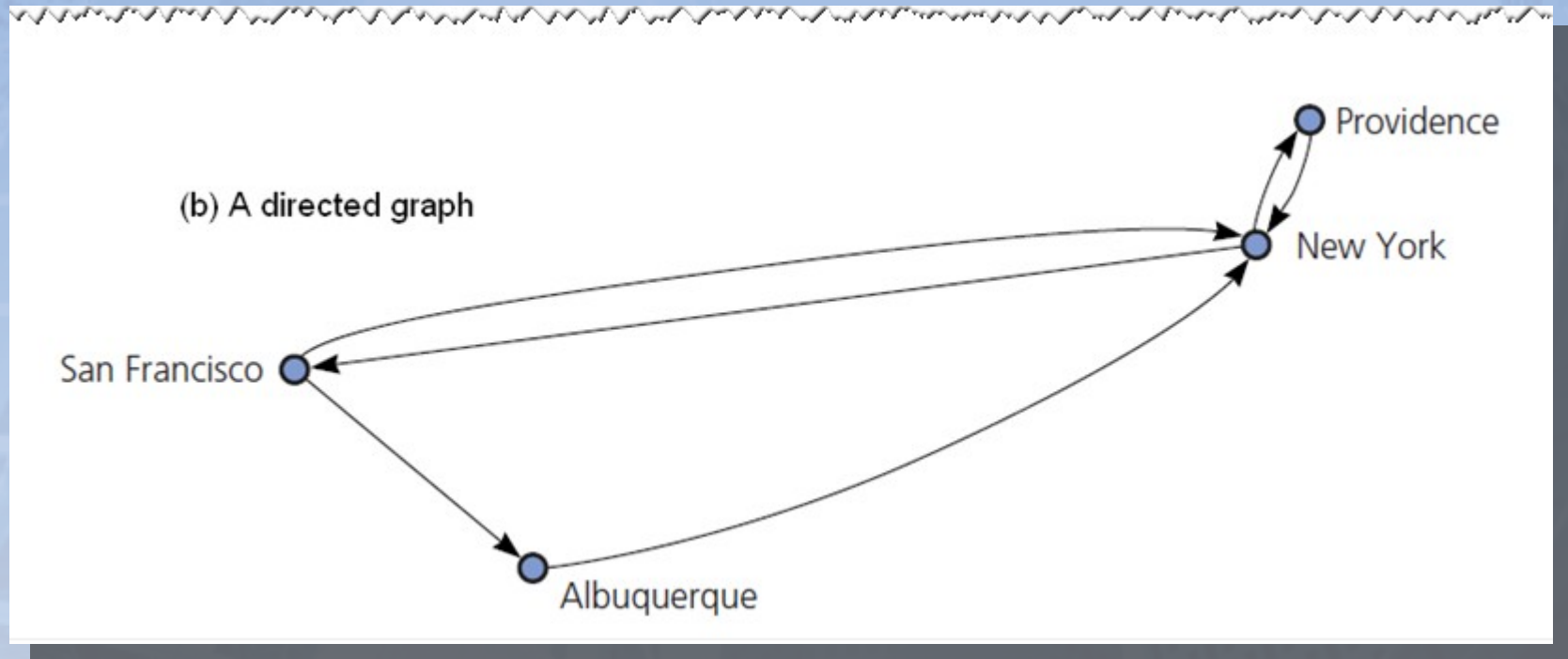


FIGURE 20-5 Examples of two kinds of graphs

Terminology

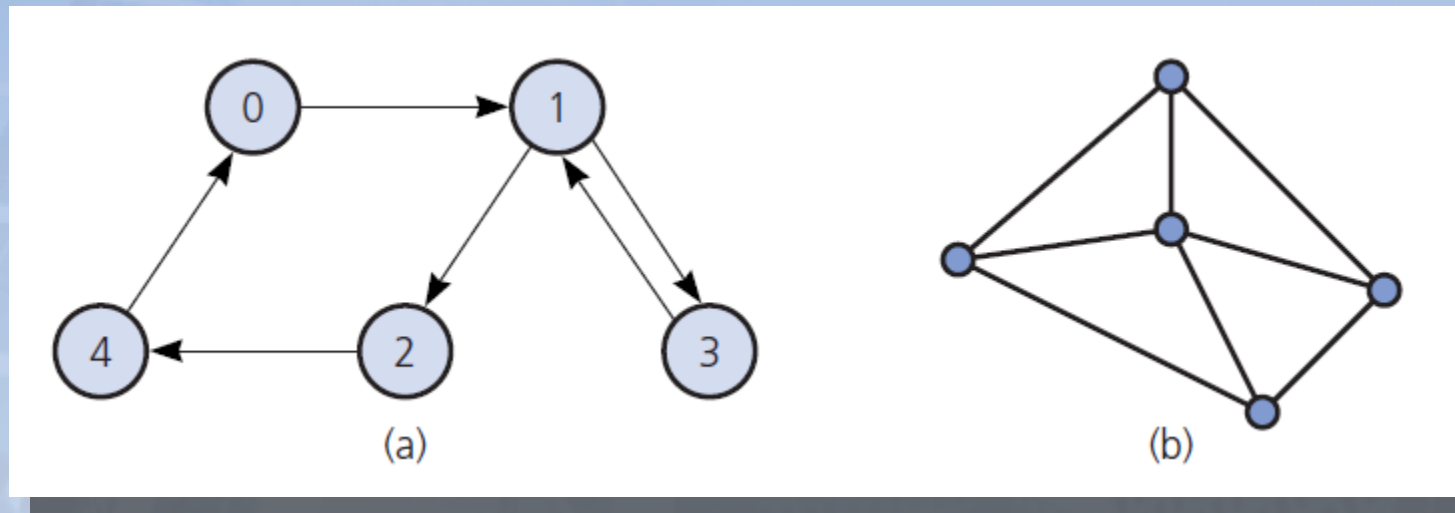


FIGURE 20-6 Graphs for Checkpoint Questions 1, 3, 4, 5

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

Graphs as ADTs

```
1  /** An interface for the ADT undirected, connected graph.
2   @file GraphInterface.h */
3  #ifndef GRAPH_INTERFACE_
4  #define GRAPH_INTERFACE_
5
6  template<class LabelType>
7  class GraphInterface
8  {
9  public:
10     /** Gets the number of vertices in this graph.
11     @return The number of vertices in the graph. */
12     virtual int getNumVertices() const = 0;
13
14     /** Gets the number of edges in this graph.
15     @return The number of edges in the graph. */
16     virtual int getNumEdges() const = 0;
```

LISTING 20-1 A C++ interface for undirected, connected graphs

Graphs as ADTs

```
17
18  /** Creates an undirected edge in this graph between two vertices
19      that have the given labels. If such vertices do not exist, creates
20      them and adds them to the graph before creating the edge.
21      @param start  A label for the first vertex.
22      @param end    A label for the second vertex.
23      @param edgeWeight  The integer weight of the edge.
24      @return  True if the edge is created, or false otherwise. */
25  virtual bool add(LabelType start, LabelType end, int edgeWeight) = 0;
26
27  /** Removes an edge from this graph. If a vertex is left with no other edges,
28      it is removed from the graph since this is a connected graph.
29      @param start  A label for the vertex at the beginning of the edge.
30      @param end    A label for the vertex at the end of the edge.
31      @return  True if the edge is removed, or false otherwise. */
32  virtual bool remove(LabelType start, LabelType end) = 0;
33
34  /** Gets the weight of an edge in this graph.
35      @return  The weight of the specified edge.
36      If no such edge exists, returns a negative integer. */
37  virtual int getEdgeWeight(LabelType start, LabelType end) const = 0;
38
```

LISTING 20-1 A C++ interface for undirected, connected graphs

Graphs as ADTs

```
38
39  /** Performs a depth-first search of this graph beginning at the given
40      vertex and calls a given function once for each vertex visited.
41      @param start  A label for the beginning vertex.
42      @param visit  A client-defined function that performs an operation on
43                    or with each visited vertex. */
44  virtual void depthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;
45
46  /** Performs a breadth-first search of this graph beginning at the given
47      vertex and calls a given function once for each vertex visited.
48      @param start  A label for the beginning vertex.
49      @param visit  A client-defined function that performs an operation on
50                    or with each visited vertex. */
51  virtual void breadthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;
52
53  /** Destroys this graph and frees its assigned memory. */
54  virtual ~GraphInterface() { }
55 }; // end GraphInterface
56 #endif
```

LISTING 20-1 A C++ interface for undirected, connected graphs

Implementing Graphs

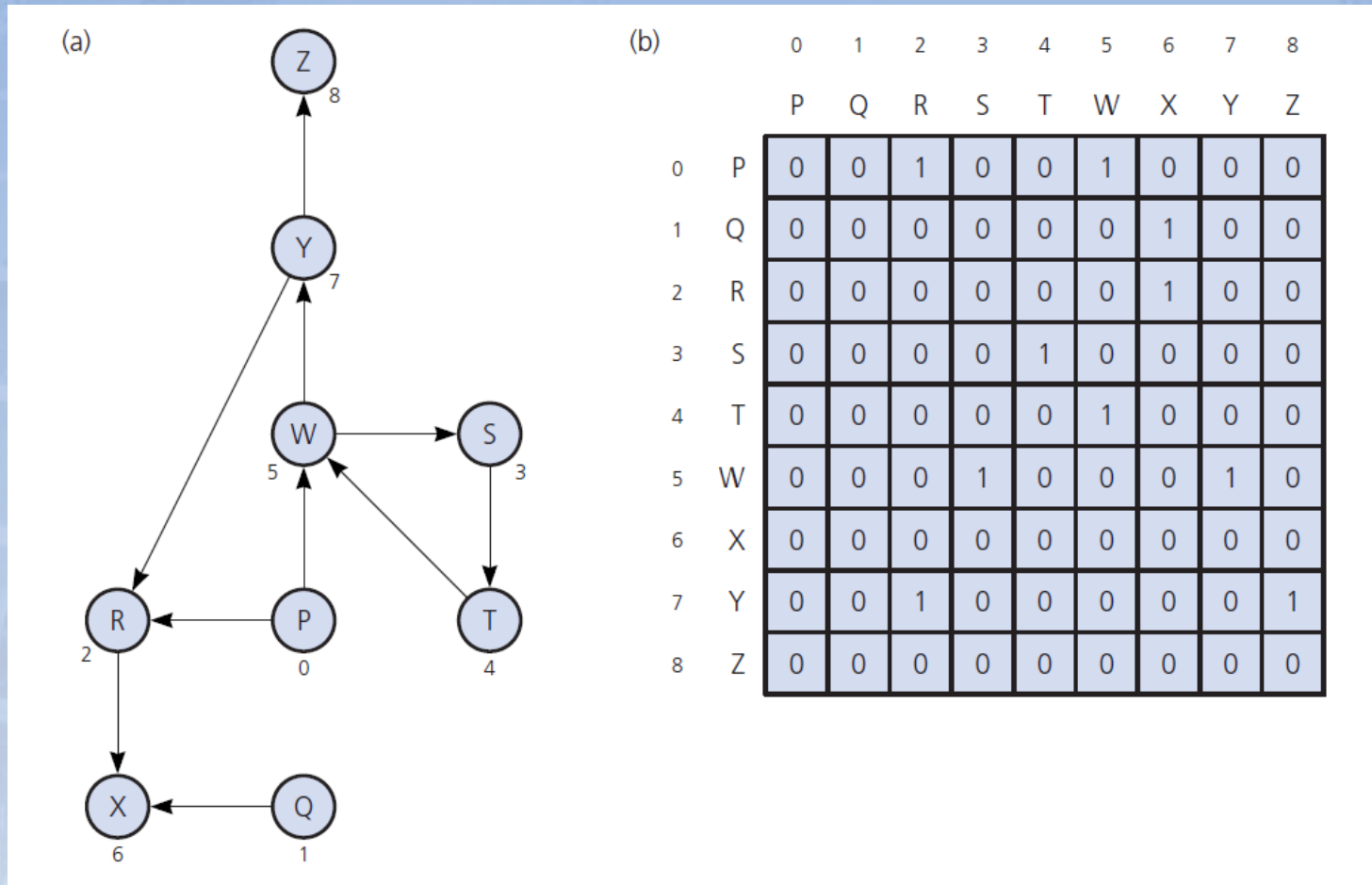


FIGURE 20-7 (a) A directed graph and (b) its adjacency matrix

Implementing Graphs

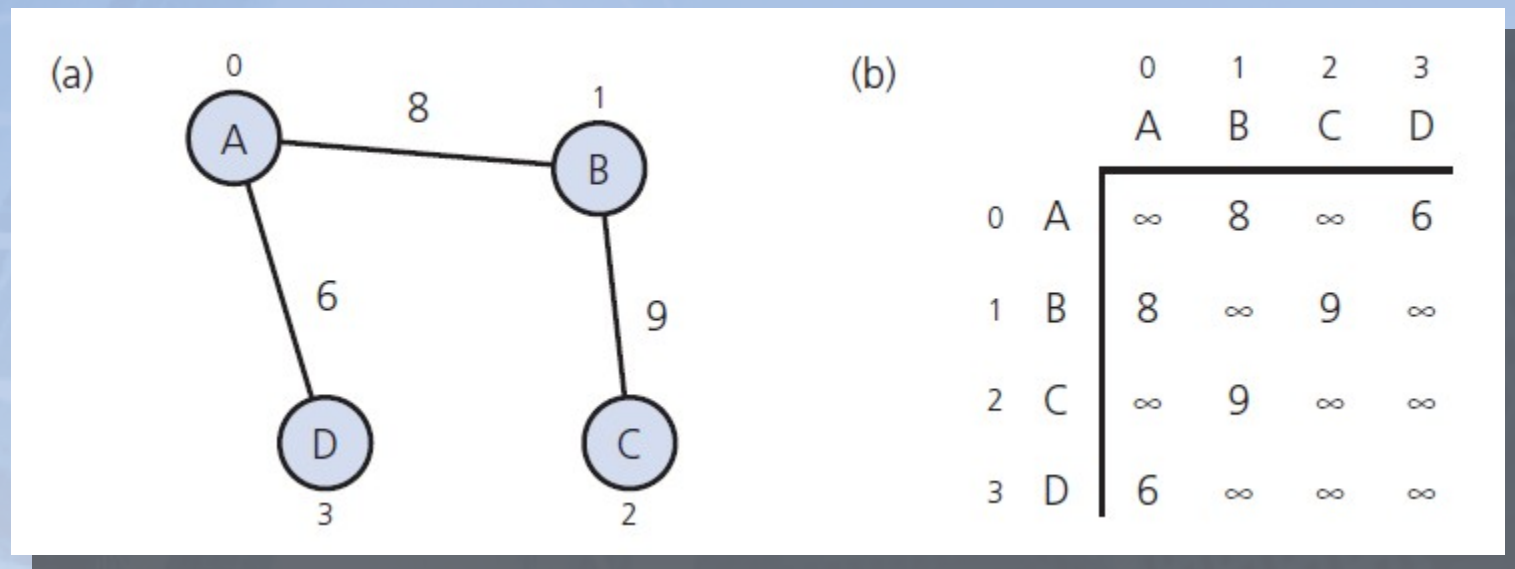


FIGURE 20-8 (a) A weighted undirected graph and (b) its adjacency matrix

Implementing Graphs

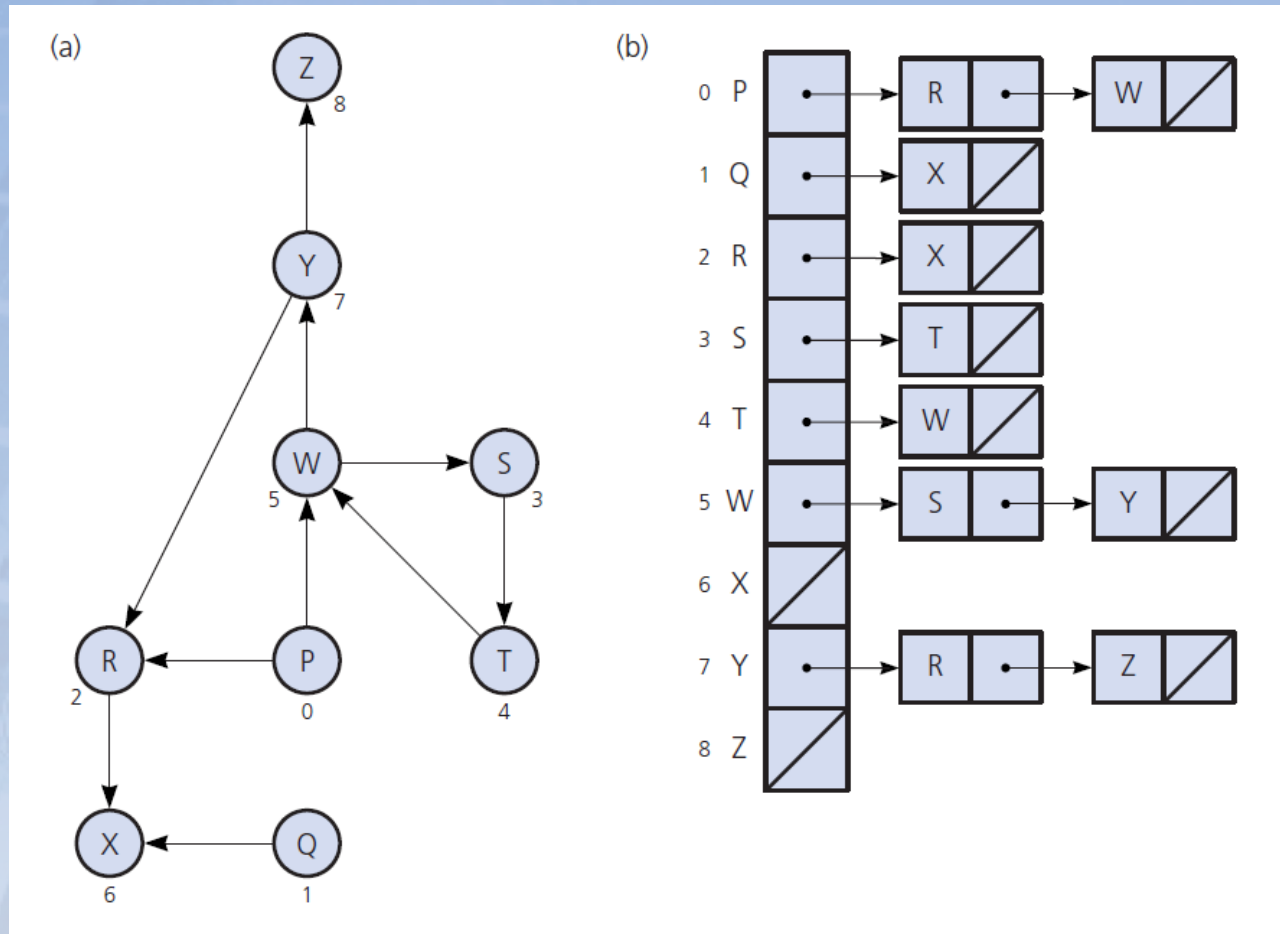


FIGURE 20-9 (a) A directed graph and (b) its adjacency list

Implementing Graphs

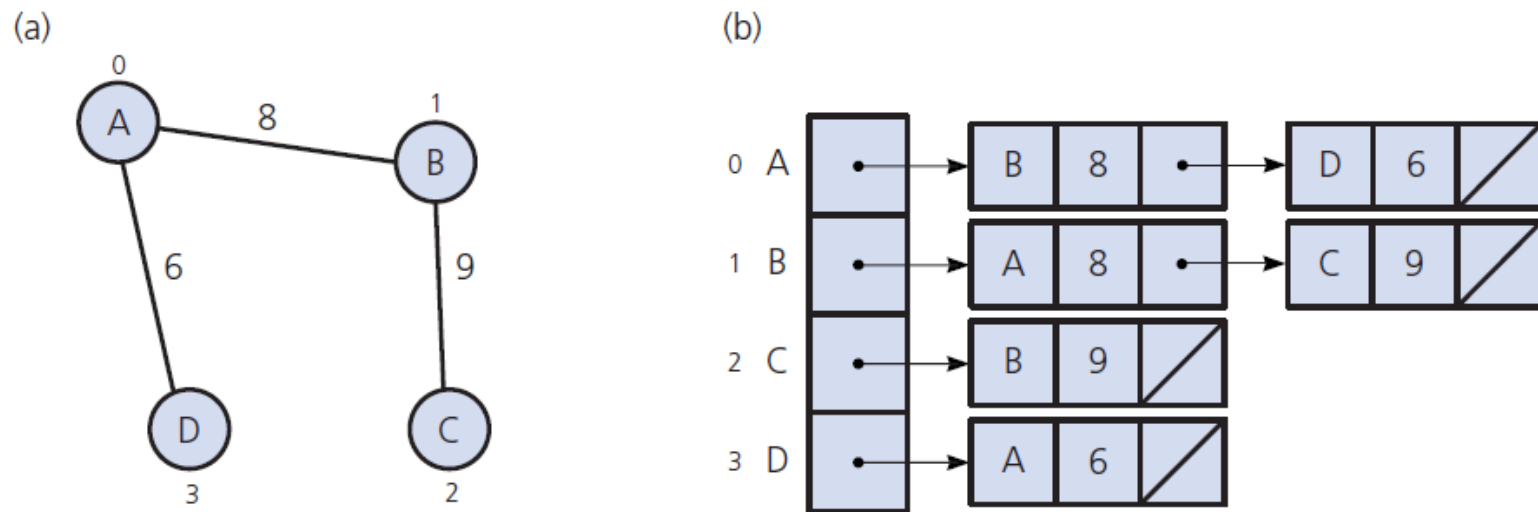


FIGURE 20-10 (a) A weighted undirected graph and (b) its adjacency list

Implementing Graphs

- Adjacency list
 - Often requires less space than adjacency matrix
 - Supports finding vertices adjacent to given vertex
- Adjacency matrix
 - Supports process of seeing if there exists an edge from vertex i to vertex j

Graph Traversals

- Visits all vertices it can reach
- Visits all vertices if and only if the graph is connected
- Connected component
 - Subset of vertices visited during a traversal that begins at a given vertex

Depth-First Search

- DFS traversal
 - Goes as far as possible from a vertex before backing up
- Recursive transversal algorithm

```
// Traverses a graph beginning at vertex v by using a  
// depth-first search: Recursive version.  
dfs(v: Vertex)  
{ Mark v as visited  
  for (each unvisited vertex u adjacent to v)  
    dfs(u)  
}
```


Breadth-First Search

- BFS traversal
 - Visits all vertices adjacent to a vertex before going forward
- BFS is a first visited, first explored strategy
 - Contrast DFS as last visited, first explored

Breadth-First Search

FIGURE 20-13 The results of a depth-first traversal, beginning at vertex a, of the graph in Figure 20-12

<u>Node visited</u>	<u>Stack (bottom to top)</u>
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)

Breadth-First Search

FIGURE 20-14 The results of a breadth-first traversal, beginning at vertex a, of the graph in Figure 20-12

<u>Node visited</u>	<u>Queue (front to back)</u>
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)

Applications of Graphs

- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits
- Some Difficult Problems

Depth-First Search

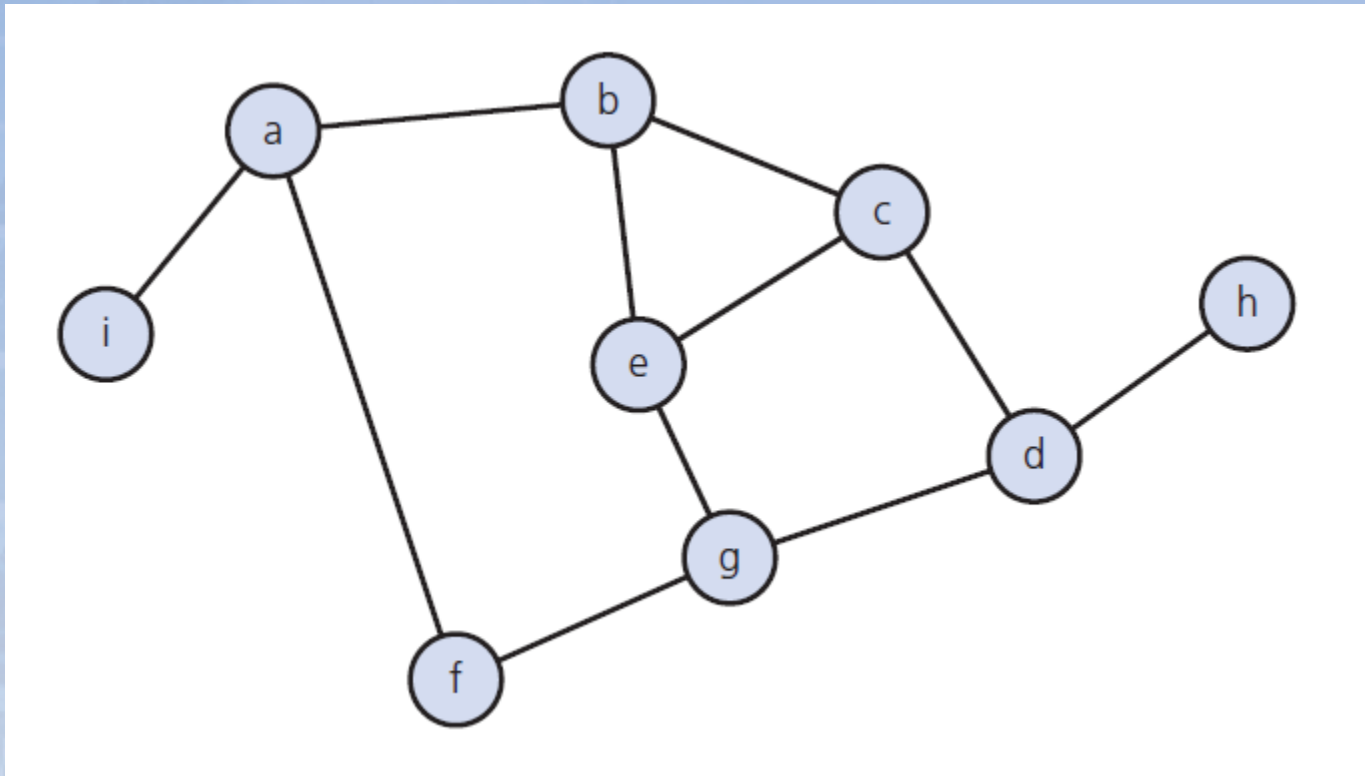


FIGURE 20-12 A connected graph with cycles

Topological Sorting

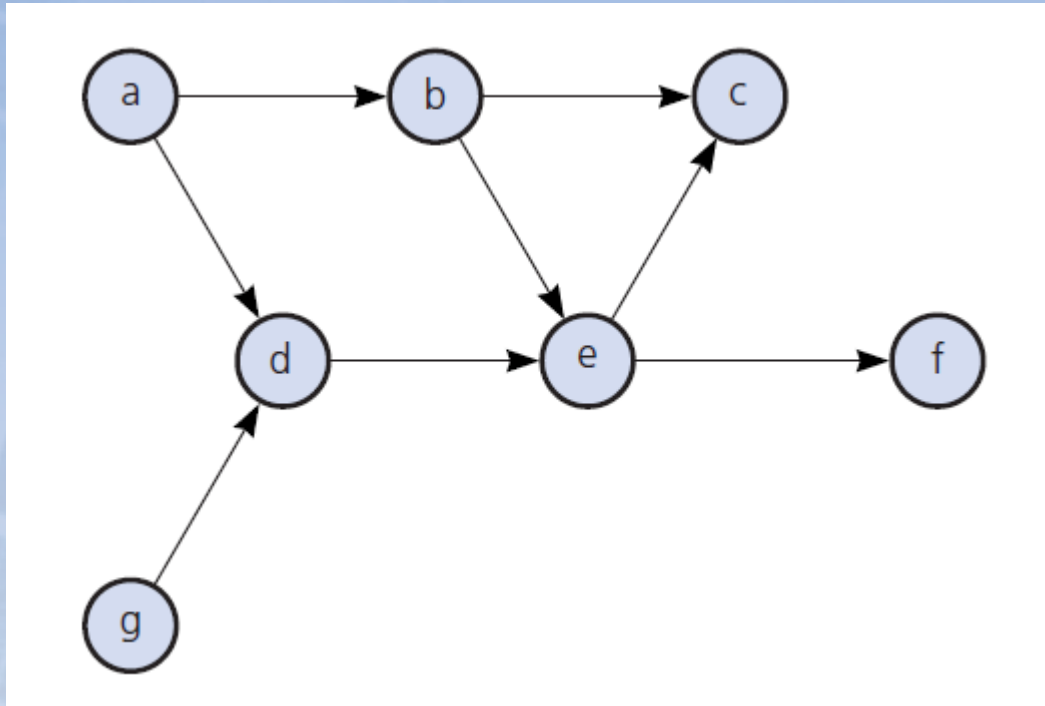


FIGURE 20-15 A directed graph without cycles

Topological Sorting

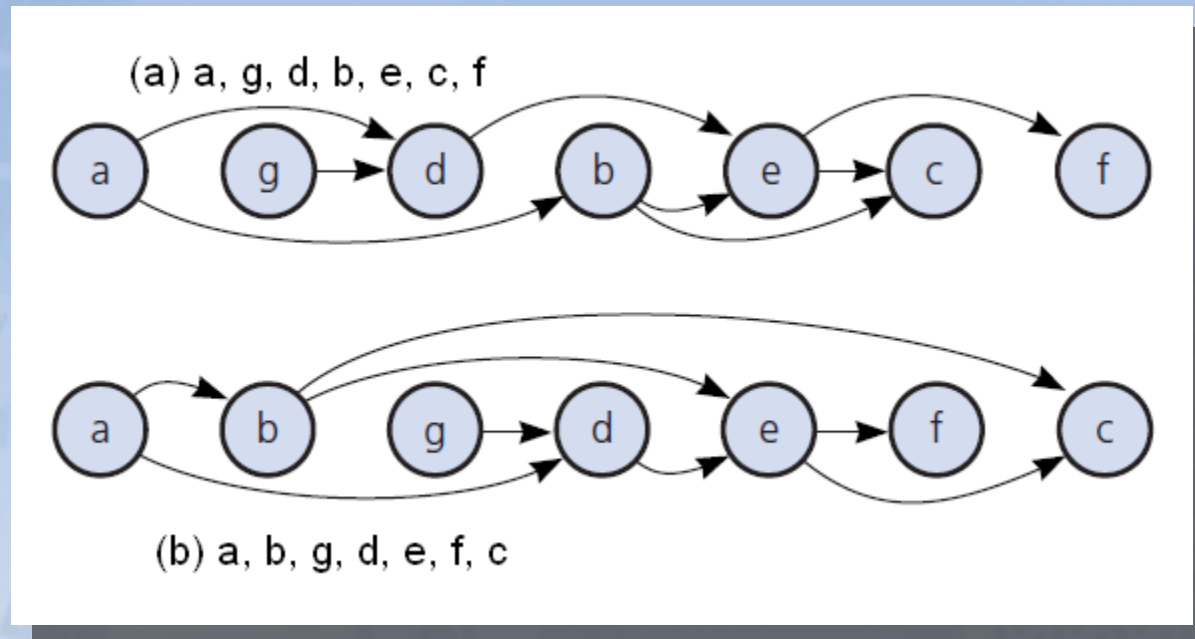


FIGURE 20-16 The graph in Figure 20-15 arranged according to two topological orders

Topological Sorting

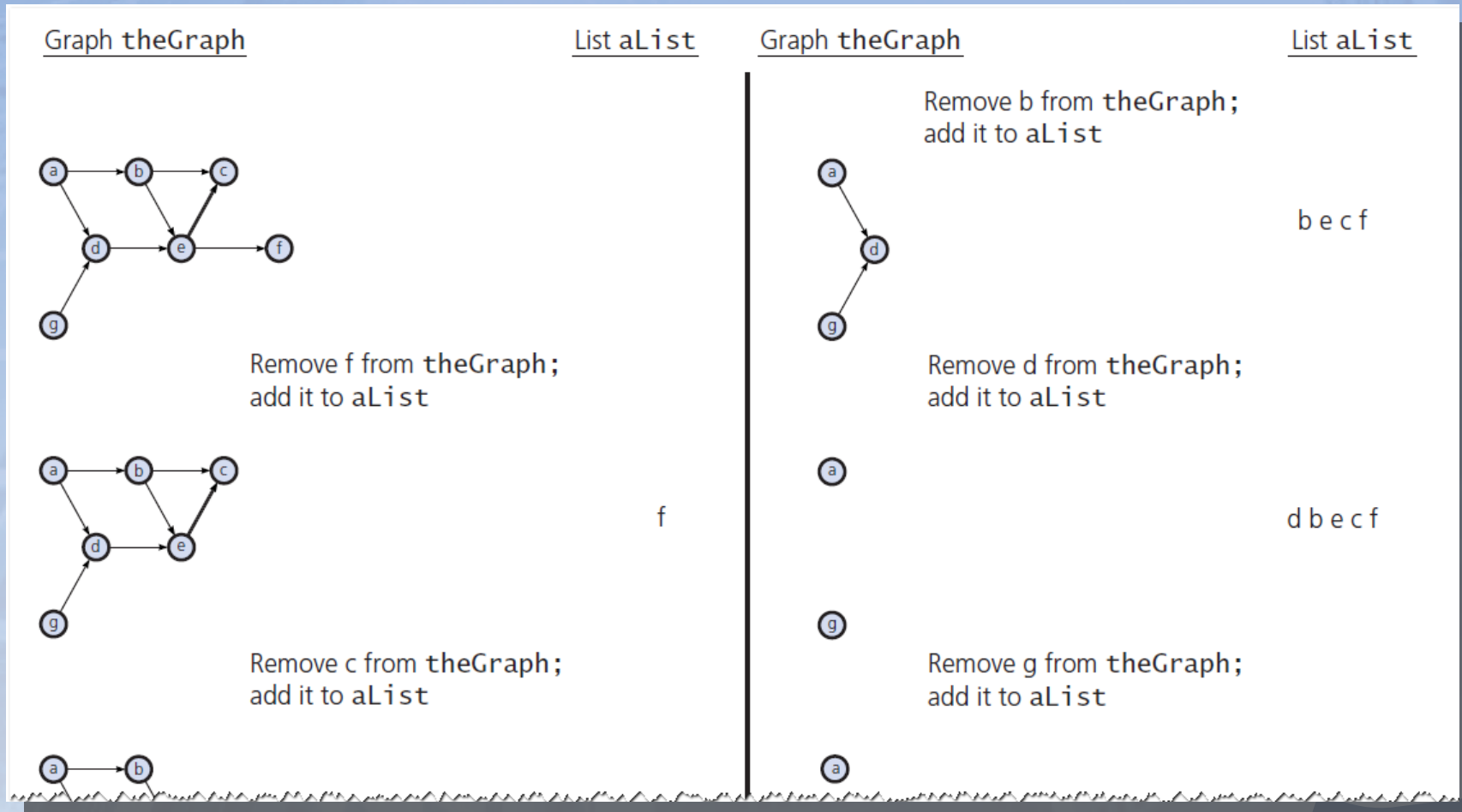


FIGURE 20-17 A trace of `topSort1` for the graph in Figure 20-15

Topological Sorting

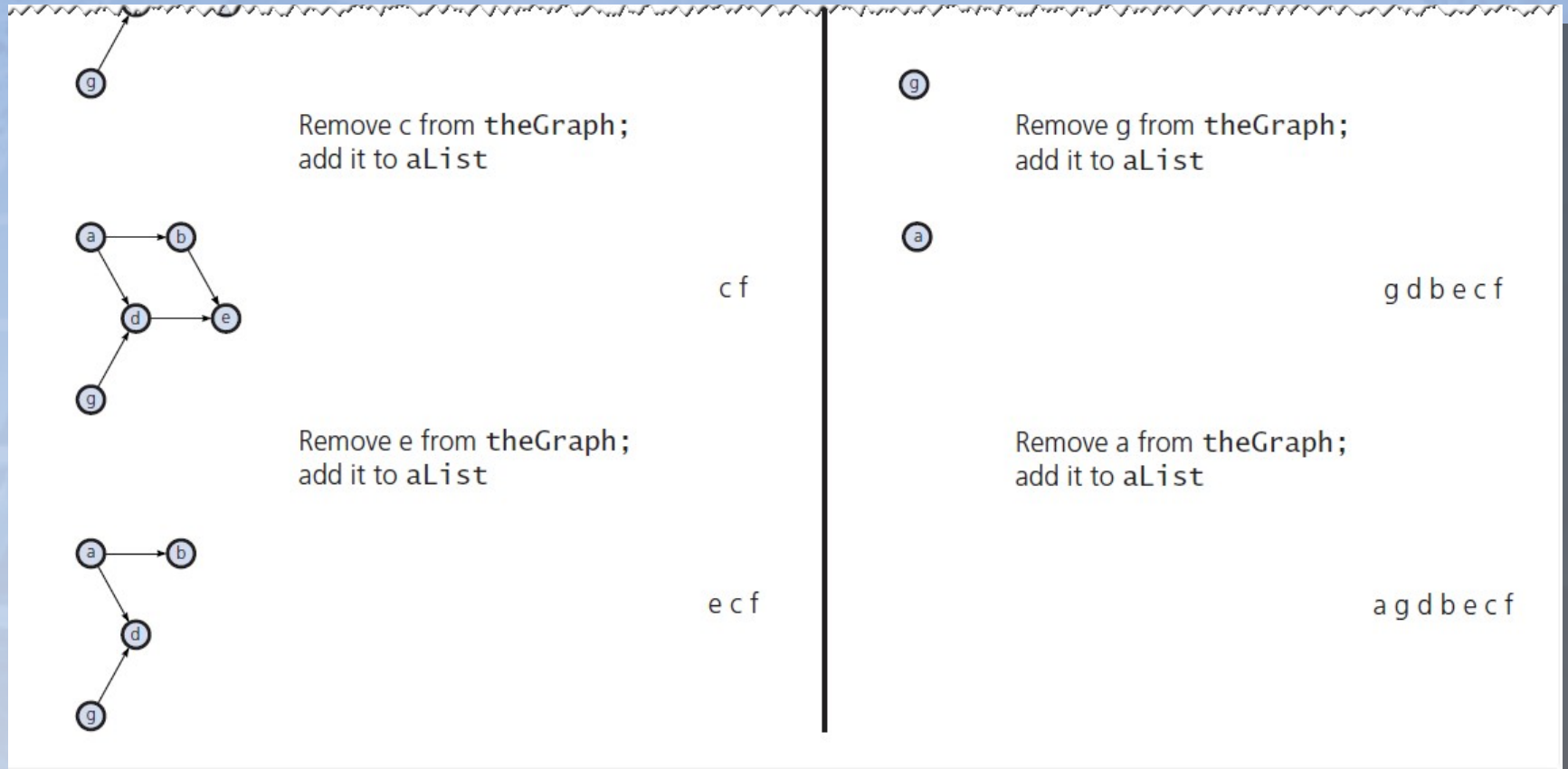


FIGURE 20-17 A trace of [topSort1](#) for the graph in Figure 20-15

Topological Sorting

<u>Action</u>	<u>Stack s (bottom to top)</u>	<u>List aList (beginning to end)</u>
Push a	a	
Push g	a g	
Push d	a g d	
Push e	a g d e	c
Push c	a g d e c	c
Pop c, add c to aList	a g d e	f c
Push f	a g d e f	e f c
Pop f, add f to aList	a g d e	d e f c
Pop e, add e to aList	a g d	g d e f c
Pop d, add d to aList	a g	g d e f c
Pop g, add g to aList	a	b g d e f c
Push b	a b	a b g d e f c
Pop b, add b to aList	a	
Pop a, add a to aList	(empty)	

FIGURE 20-18 A trace of [topSort2](#) for the graph in Figure 20-15

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

Spanning Trees

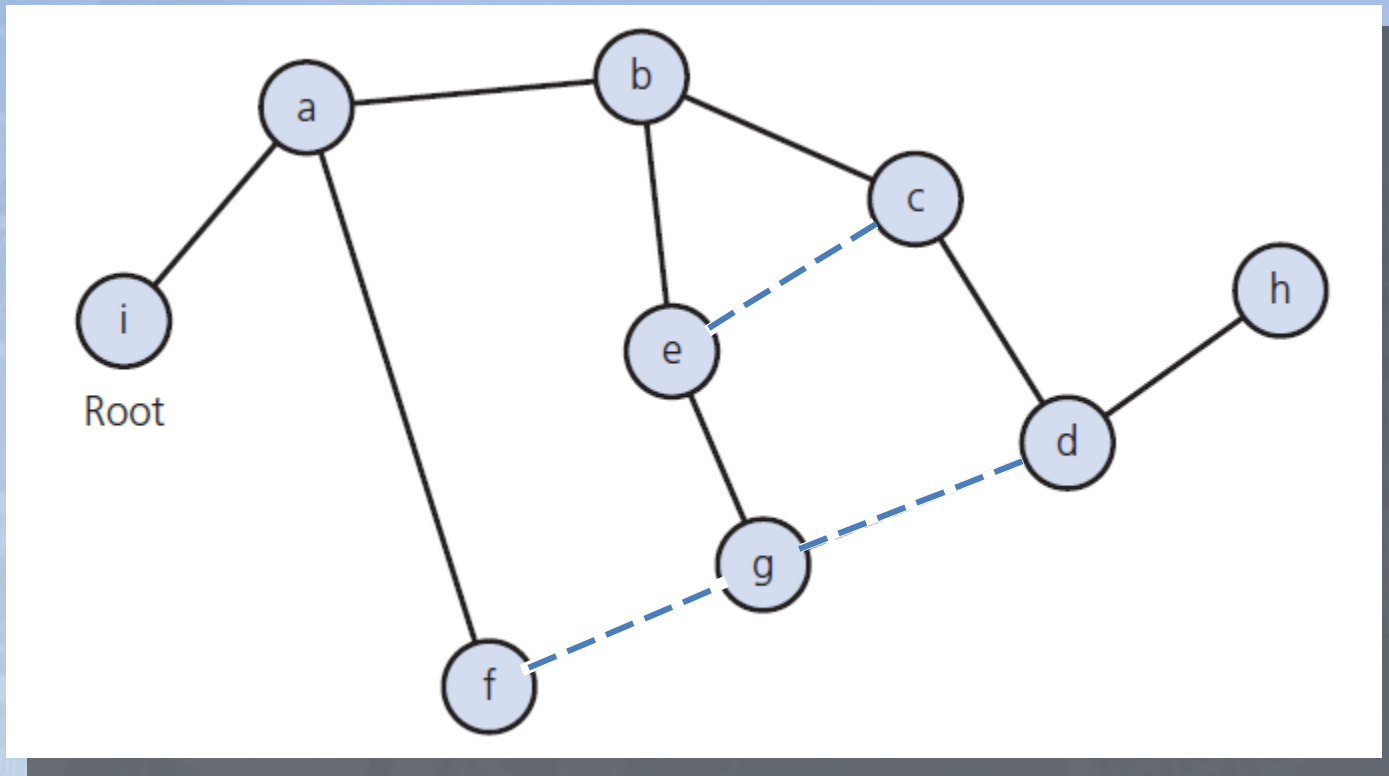


FIGURE 20-19 A spanning tree for the graph in Figure 20-12

Spanning Trees

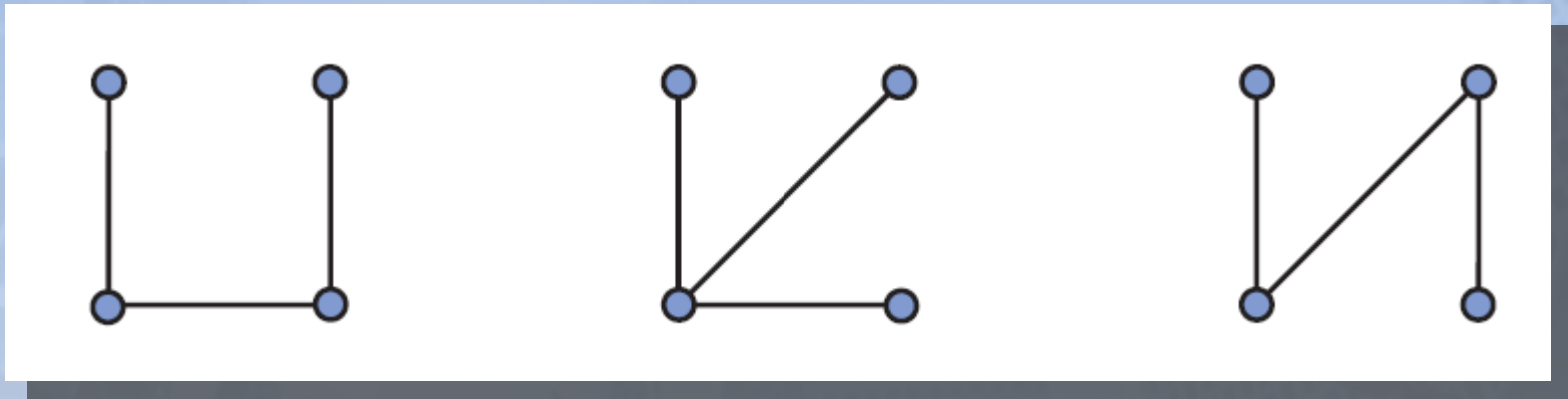
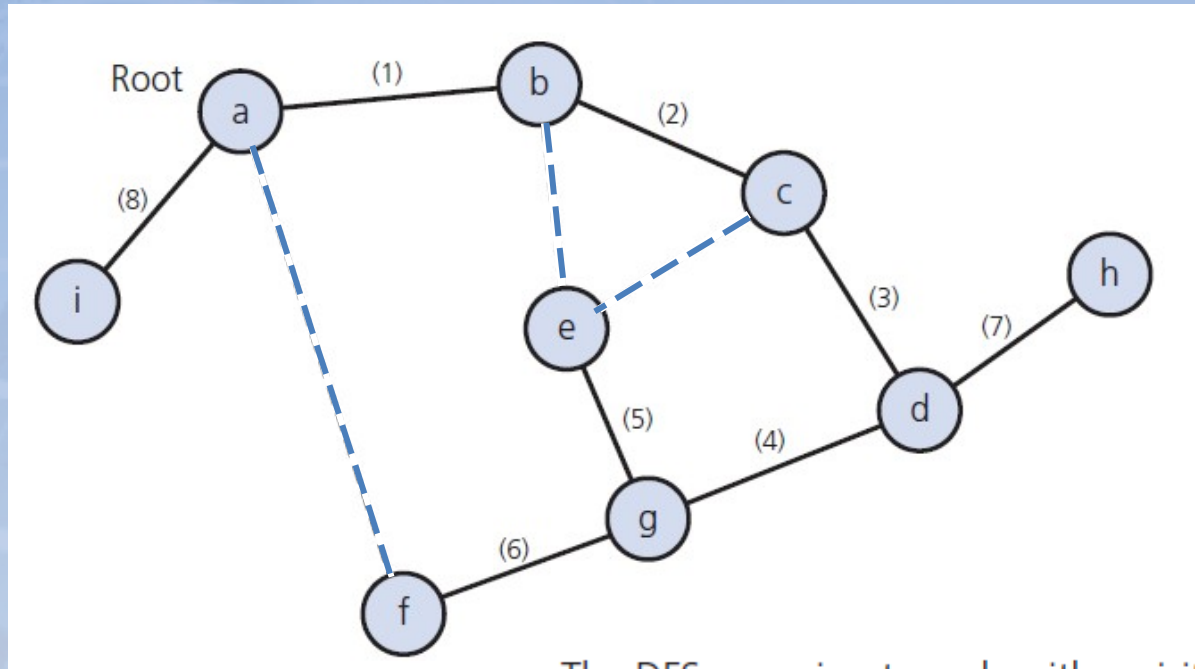


FIGURE 20-20 Connected graphs that each have four vertices and three edges

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.

FIGURE 20-21 Connected graphs that each have four vertices and three edges

????????????

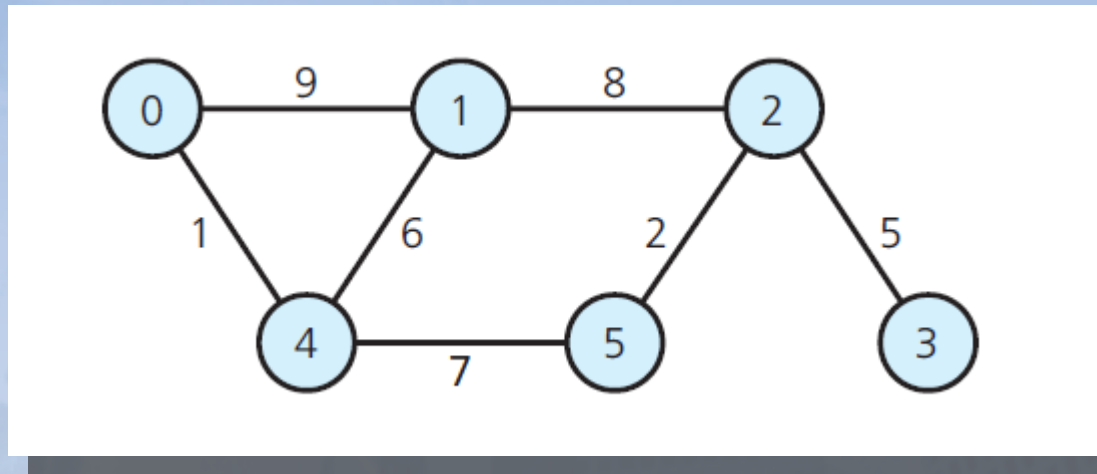


Figure 20-22 A graph for Checkpoint Questions 8, 9, and 10 and for Exercises 1 and 4

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.

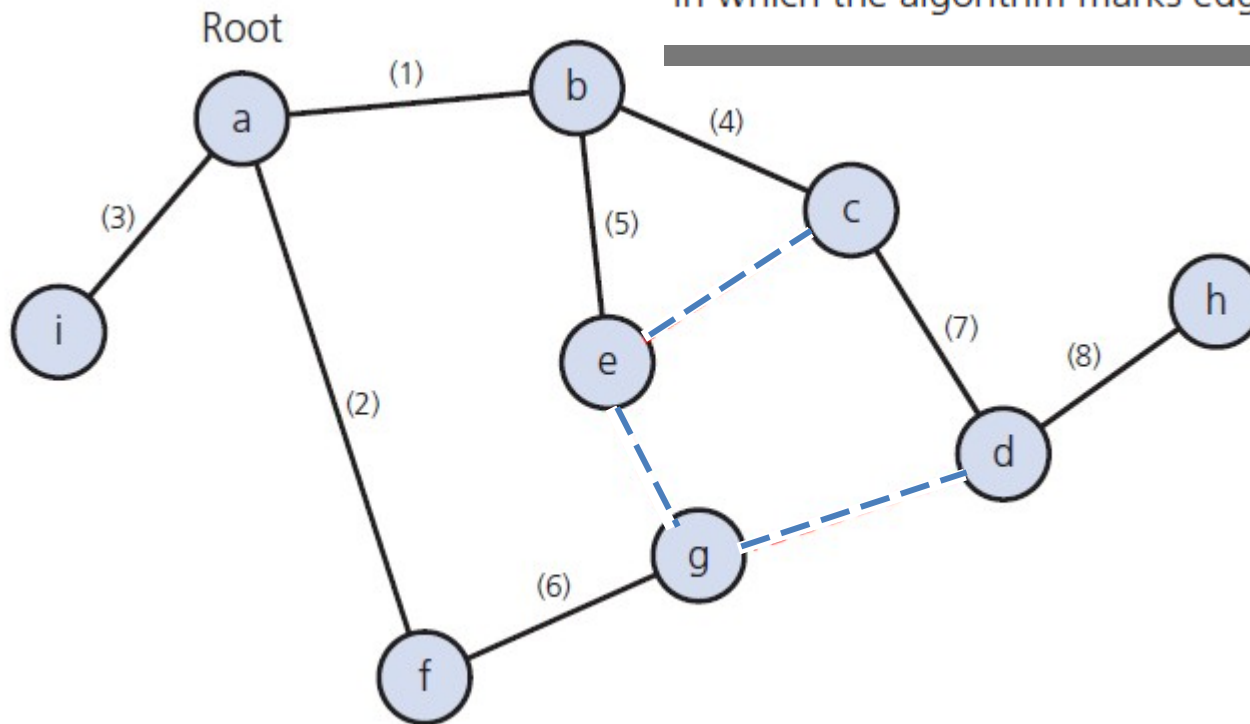


FIGURE 20-23 The BFS spanning tree rooted at vertex *a* for the graph in Figure 20-12

Minimum Spanning Trees

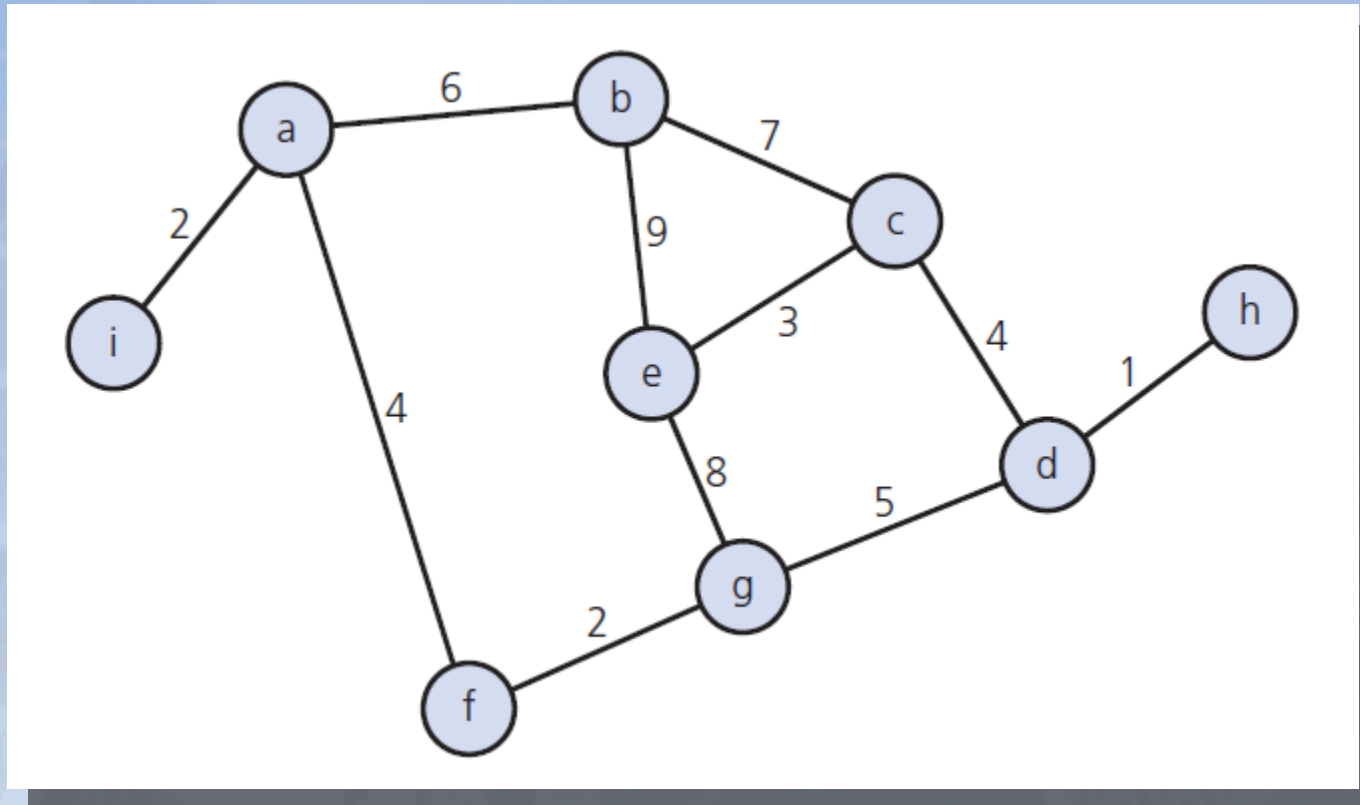


FIGURE 20-24 A weighted, connected, undirected graph

Minimum Spanning Trees

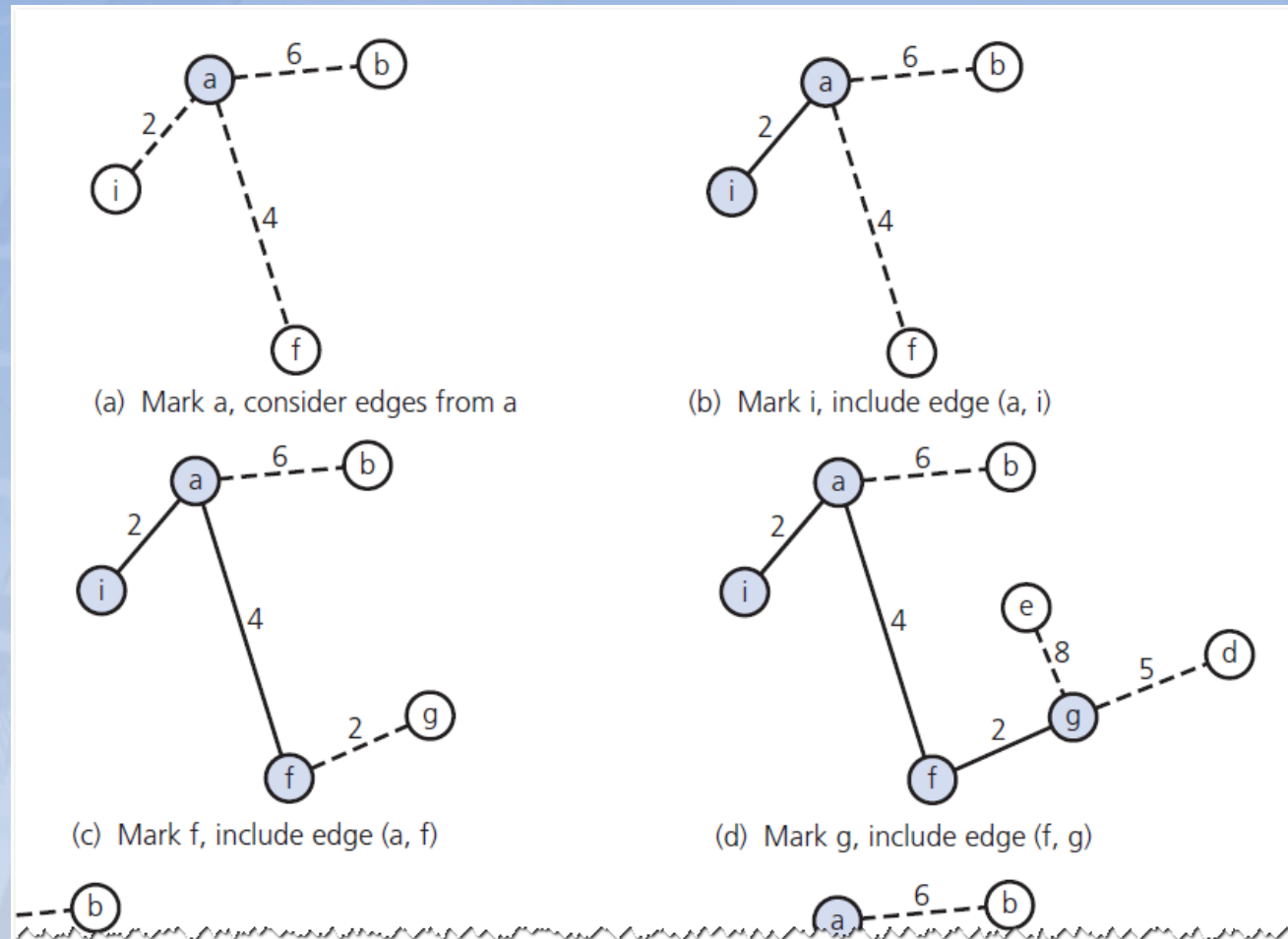


FIGURE 20-25 A trace of [prim'sAlgorithm](#) for the graph in Figure 20-23, beginning at vertex *a*

Minimum Spanning Trees

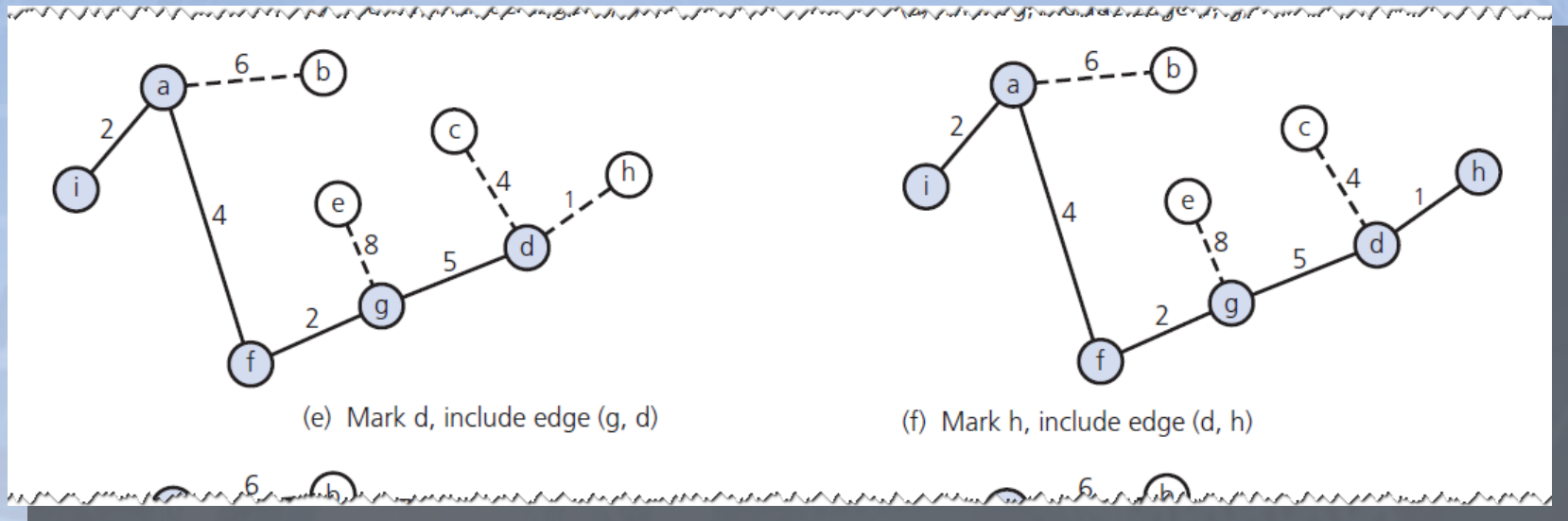


FIGURE 20-25 A trace of [primsAlgorithm](#) for the graph in Figure 20-23, beginning at vertex a

Minimum Spanning Trees

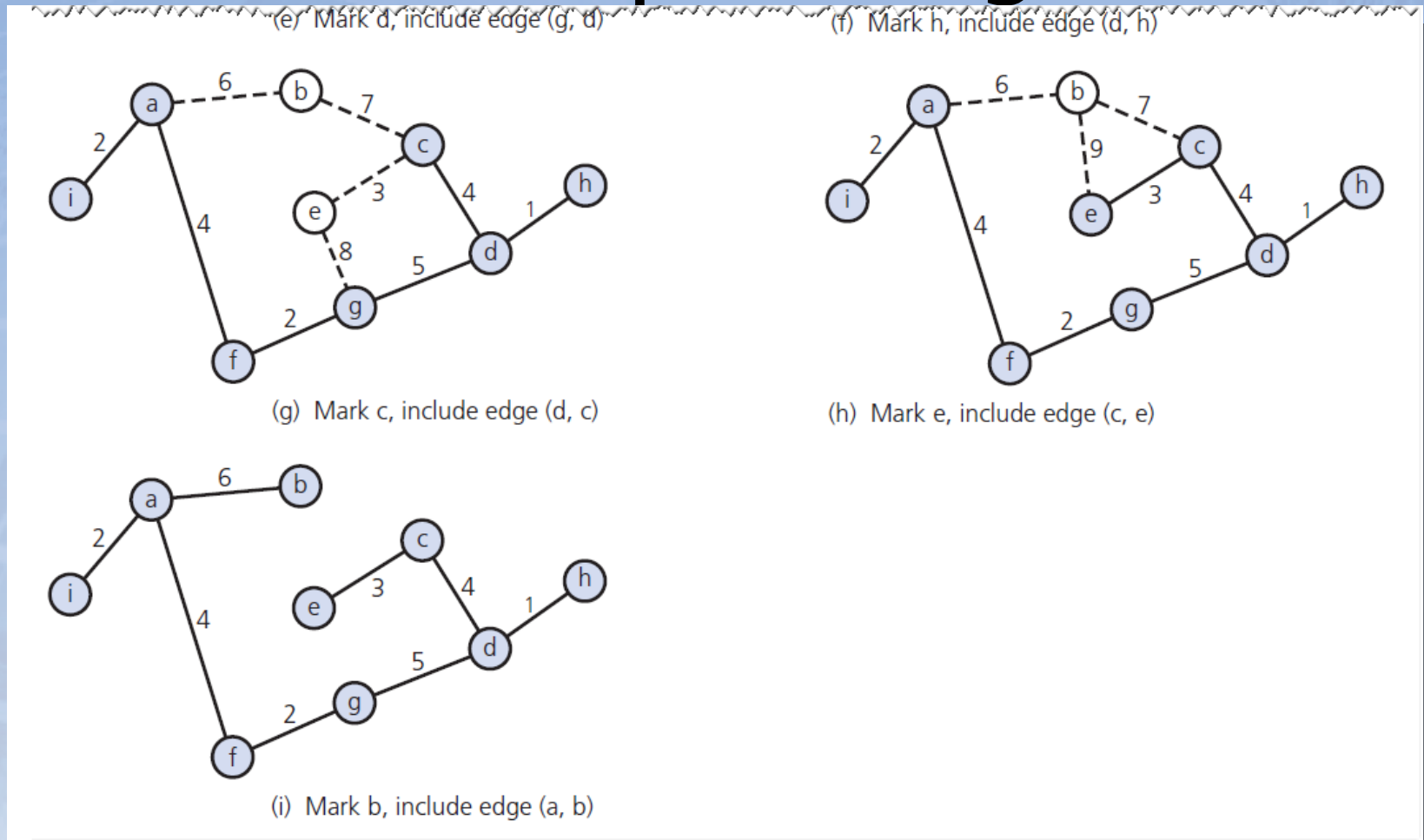


FIGURE 20-25 A trace of [primsAlgorithm](#) for the graph in Figure 20-23, beginning at vertex a

Shortest Paths

- The shortest path between two vertices in a weighted graph
 - Has the smallest edge-weight sum

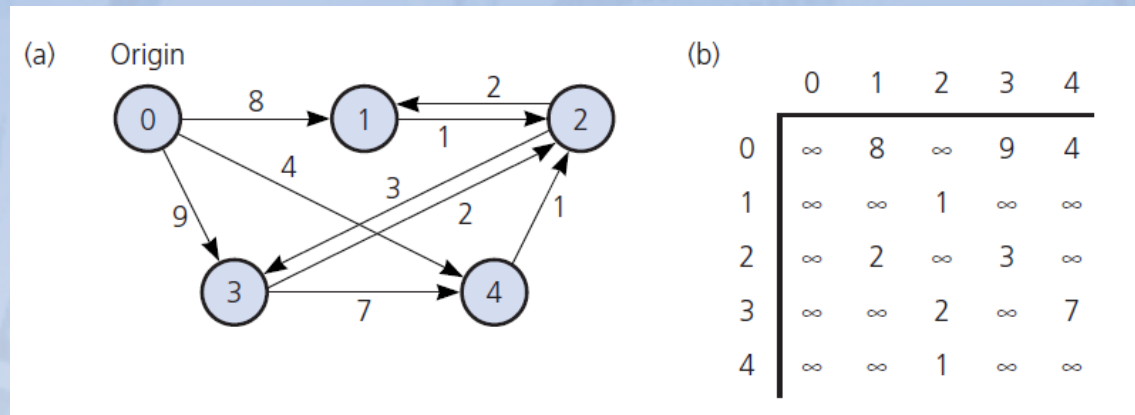


FIGURE 20-26 (a) A weighted directed graph and (b) its adjacency matrix

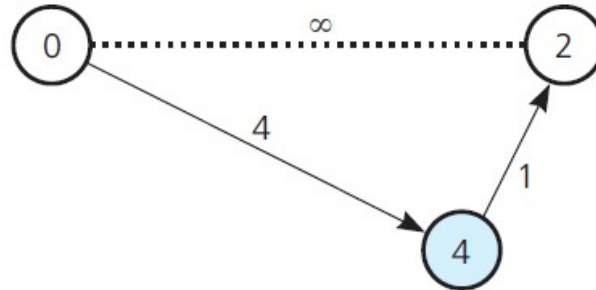
Shortest Paths

<u>Step</u>	<u>v</u>	<u>vertexSet</u>	<u>weight</u>				
			<u>[0]</u>	<u>[1]</u>	<u>[2]</u>	<u>[3]</u>	<u>[4]</u>
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

FIGURE 20-27 A trace of the shortest-path algorithm applied to the graph in Figure 20-25a

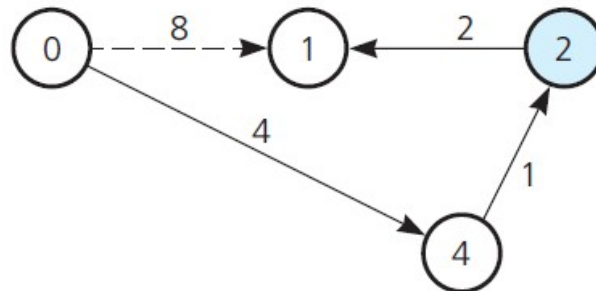
Shortest Paths

(a) $weight[2]$ in step 2



Step 2. The path 0-4-2 is shorter than 0-2

(b) $weight[1]$ in step 3

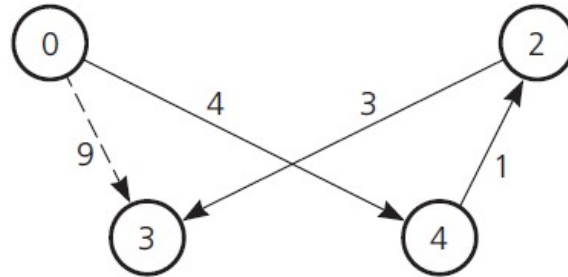


Step 3. The path 0-4-2-1 is shorter than 0-1

FIGURE 20-28 Checking $weight[u]$ by examining the graph:
(a) $weight[2]$ in step 2; (b) $weight[1]$ in step 3;

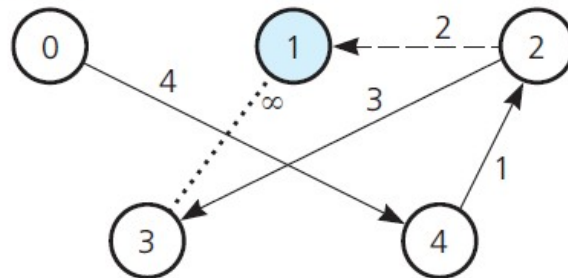
Shortest Paths

(c) $\text{weight}[3]$ in step 3



Step 3 continued. The path 0-4-2-3 is shorter than 0-3

(d) $\text{weight}[3]$ in step 4



Step 4. The path 0-4-2-3 is shorter than 0-4-2-1-3

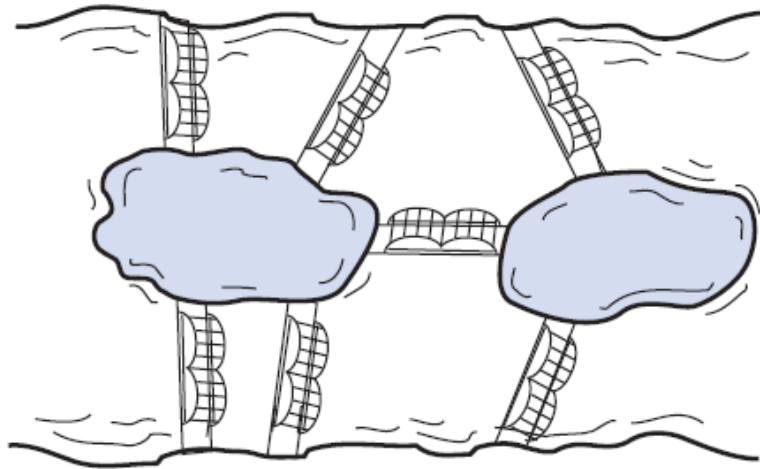
FIGURE 20-28 Checking $\text{weight}[u]$ by examining the graph:
(c) $\text{weight}[3]$ in step 3; (d) $\text{weight}[3]$ in step 4

Circuits

- Circuit
 - Another name for type of cycle common in statement of certain types of problems
 - Typical circuits either visit every vertex once or every edge once
- Euler Circuit
 - Begins at vertex v
 - Passes through every edge exactly once
 - Terminates at v

Circuits

(a)



(b)

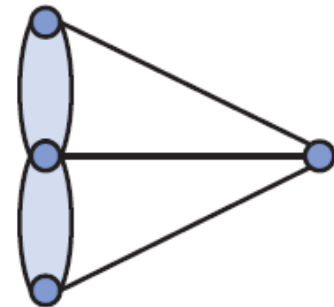
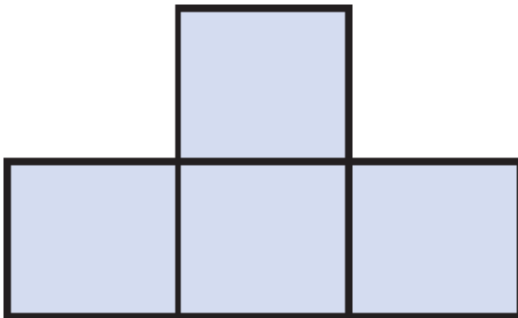


FIGURE 20-28 (a) Euler's bridge problem and
(b) its multigraph representation

Circuits

(a)



(b)

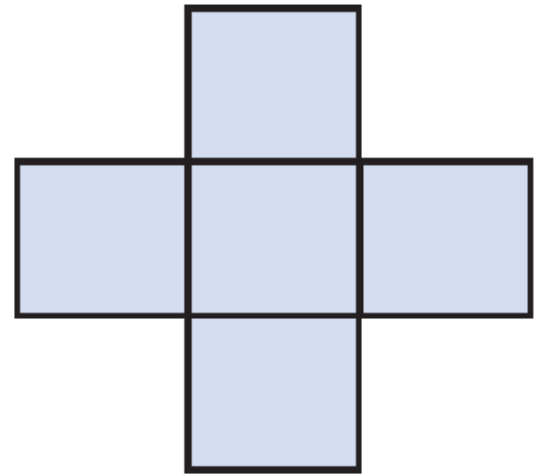
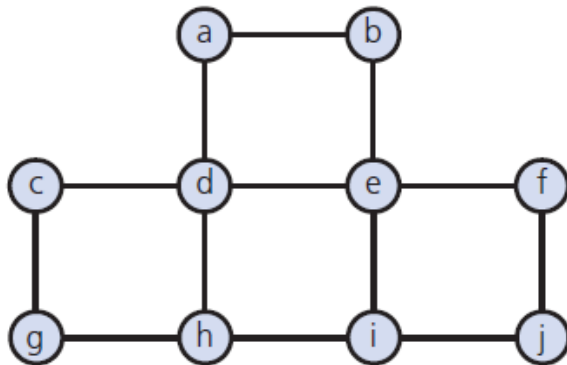


FIGURE 20-29 Pencil and paper drawings

Circuits

(a)



(b)

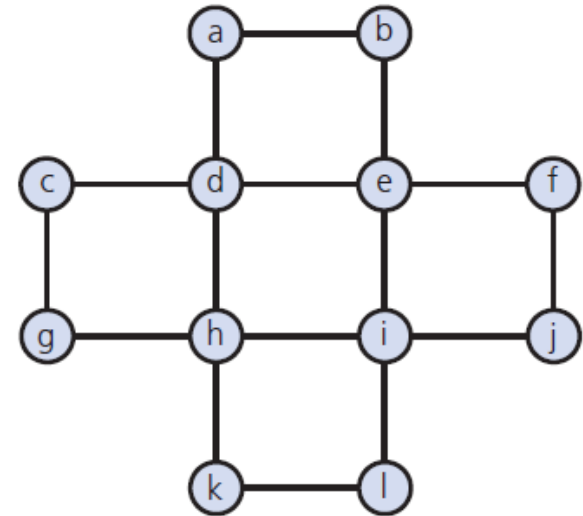


FIGURE 20-30 Connected undirected graphs based on the drawings in Figure 20-29

Circuits

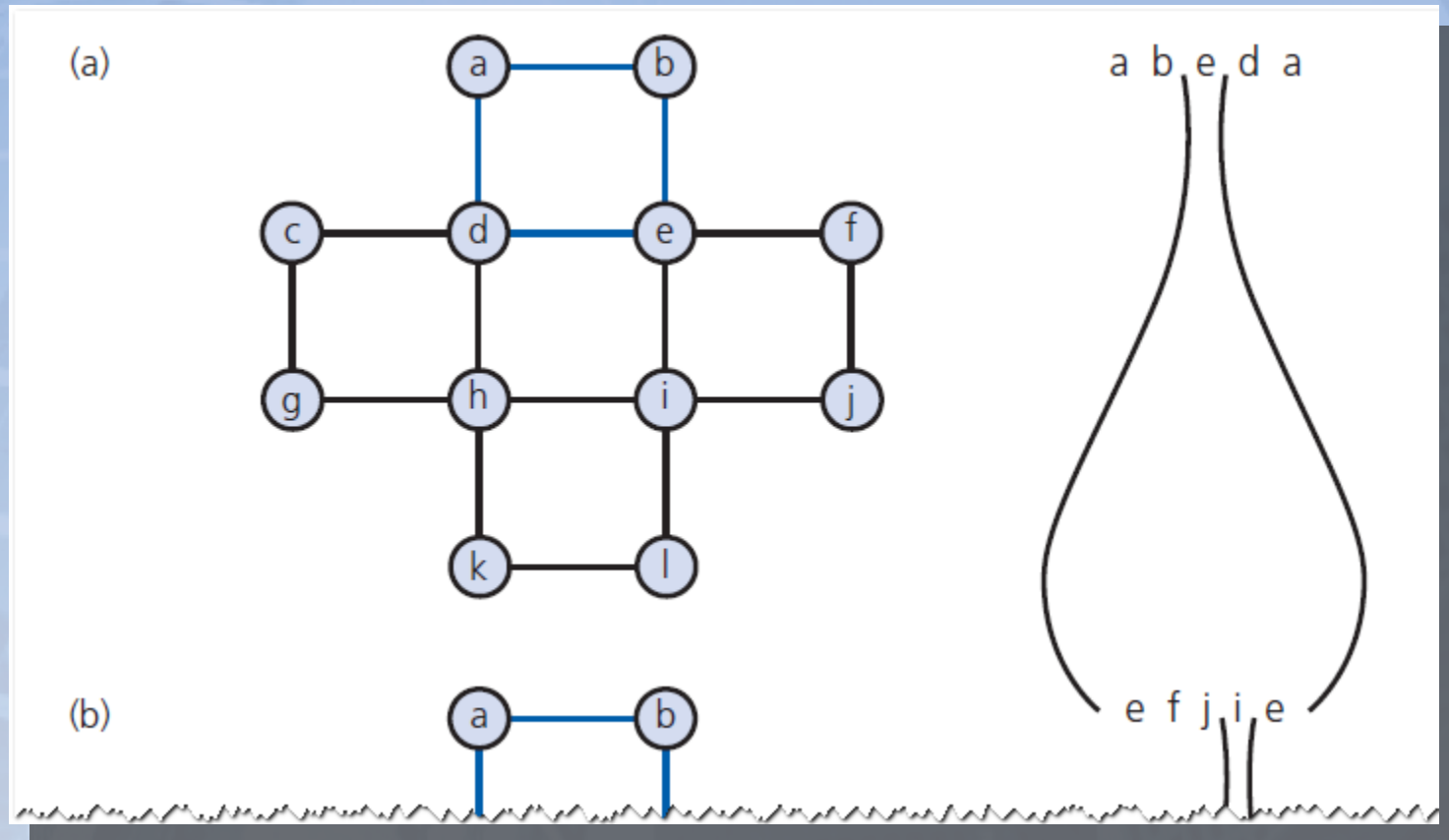


FIGURE 20-31 The steps to find an Euler circuit for the graph in Figure 20-30b

Circuits

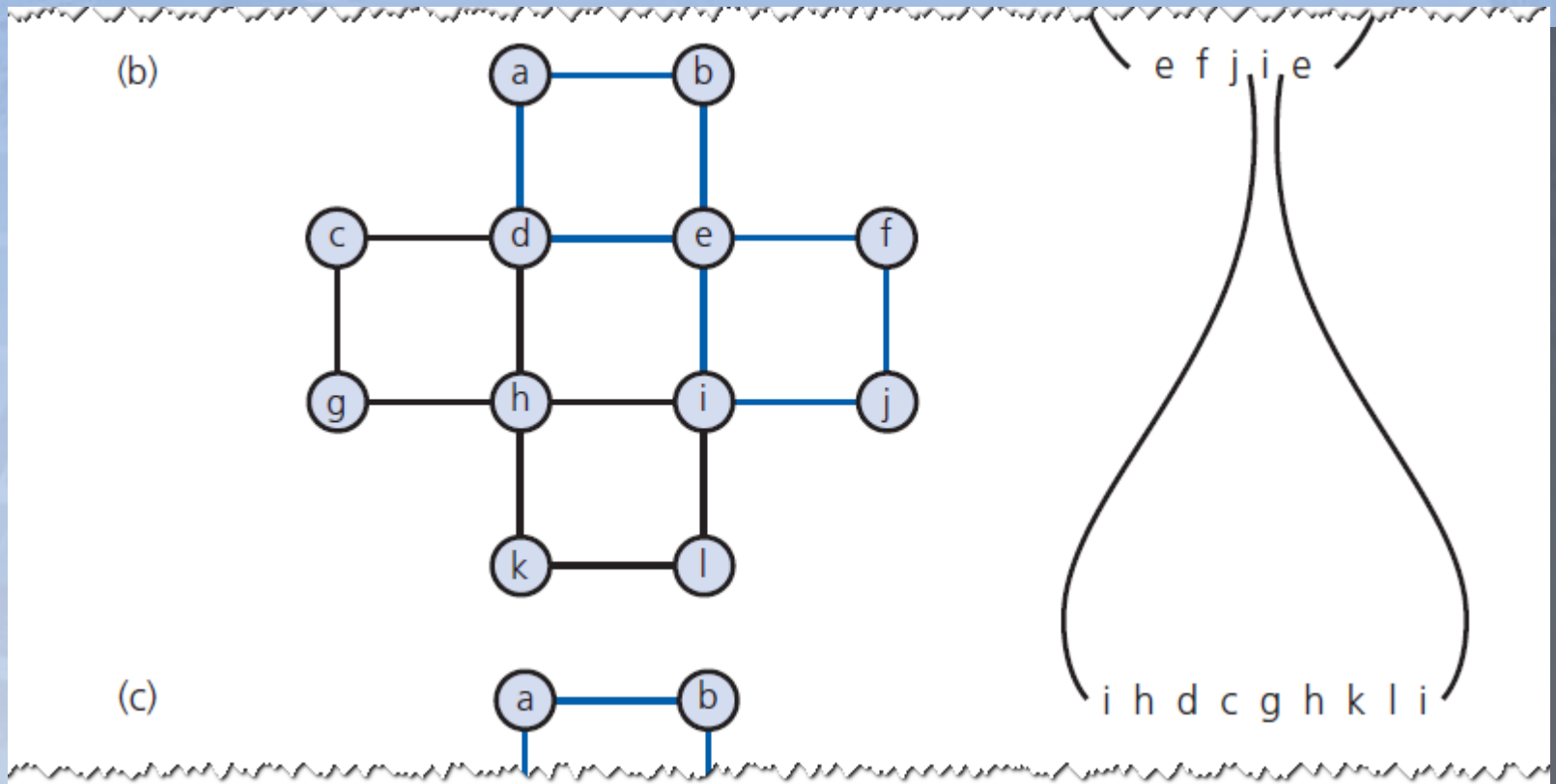


FIGURE 20-31 The steps to find an Euler circuit for the graph in Figure 20-30b

Circuits

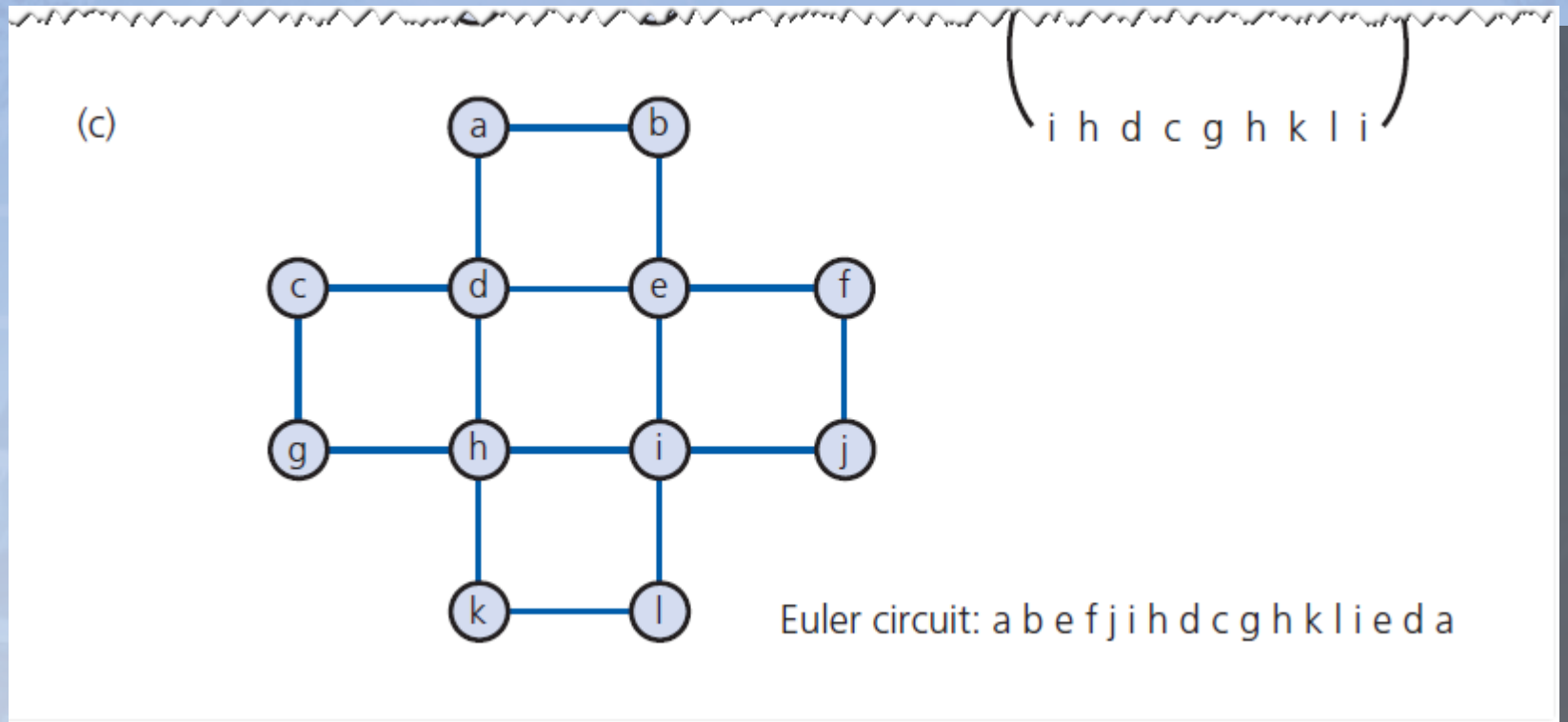


FIGURE 20-31 The steps to find an Euler circuit for the graph in Figure 20-30b

Some Difficult Problems

- 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 route exactly once
 - Edge (road) has associated cost (mileage)
 - Goal is determine least expensive circuit

Some Difficult Problems

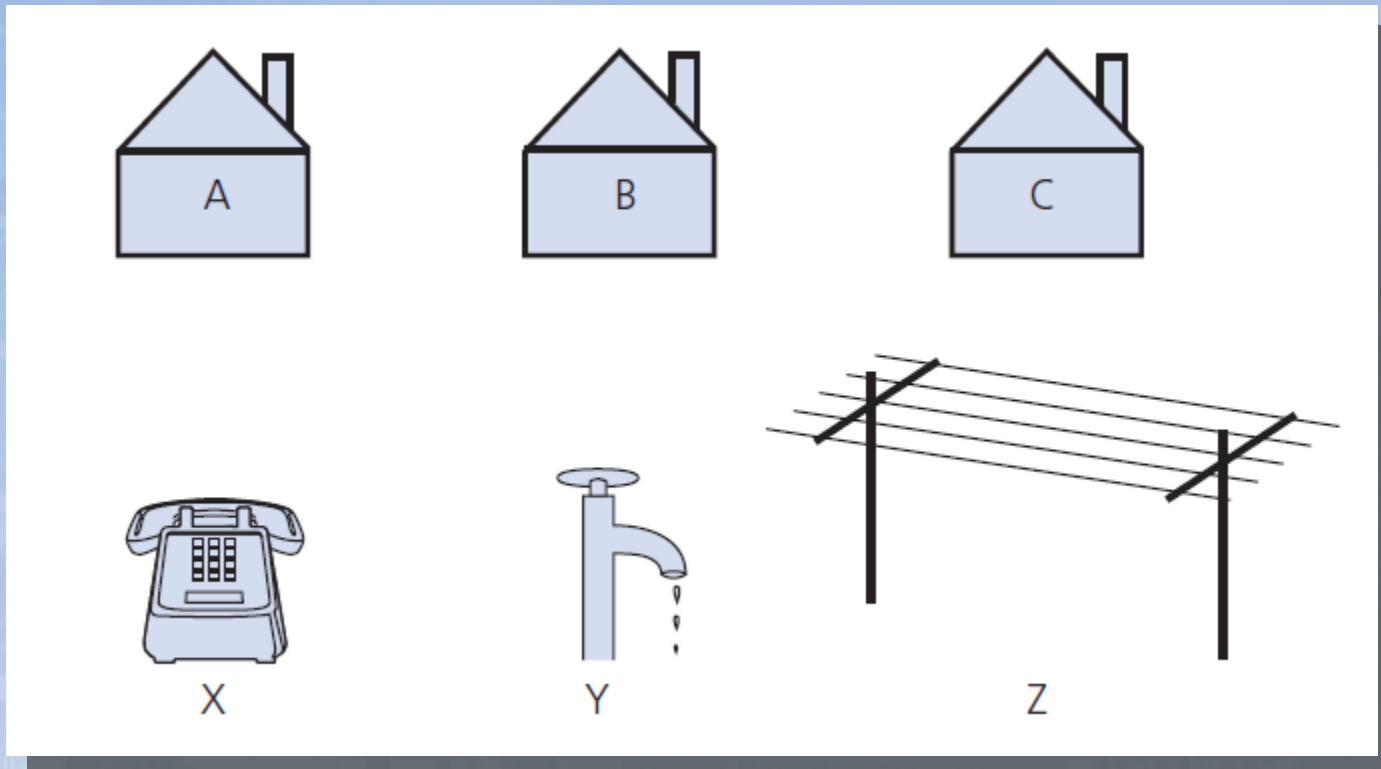
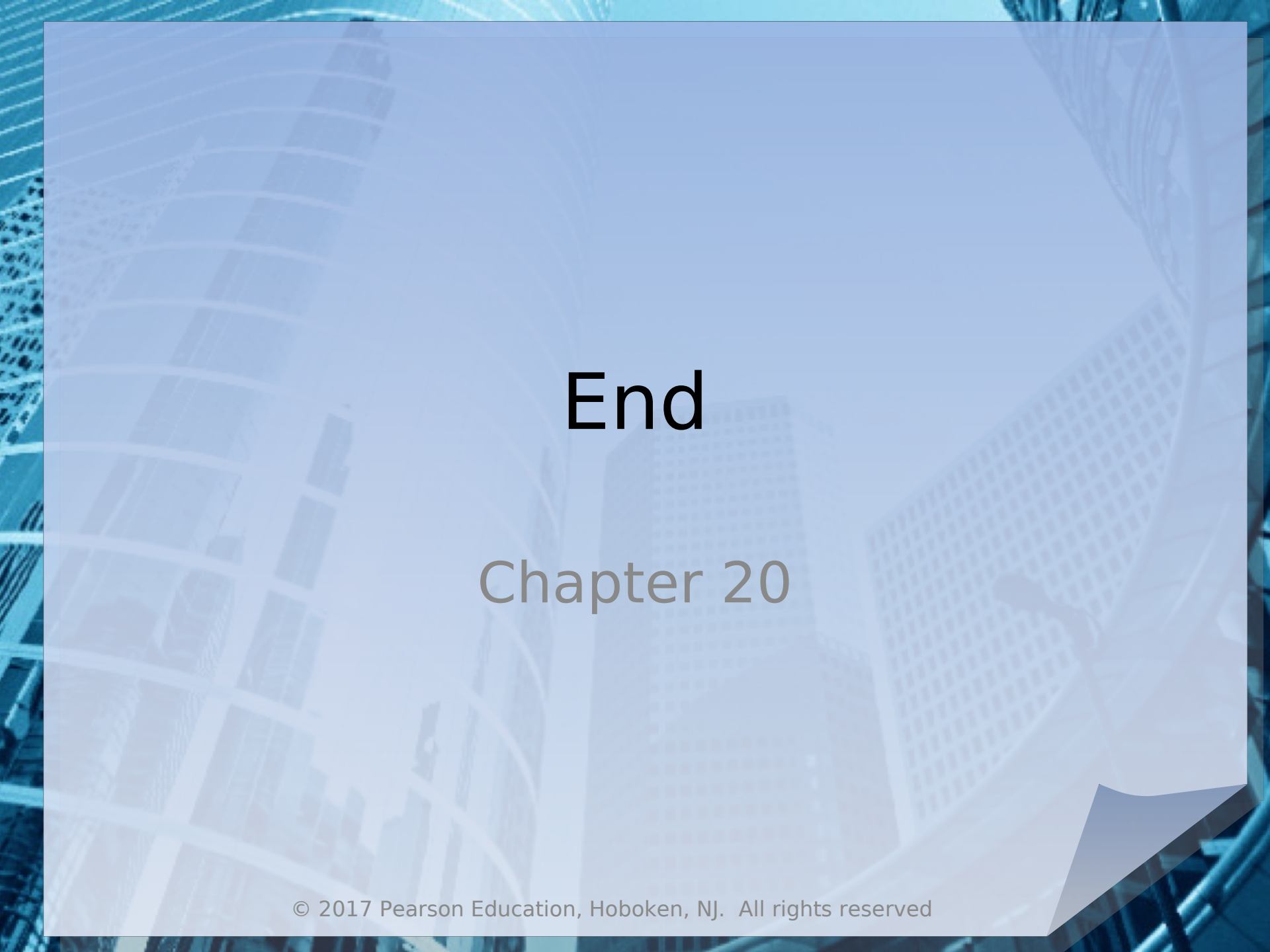


FIGURE 20-32 The three utilities problem



End

Chapter 20