Graphs

Contents

- Terminology
- Graphs as ADTs
- Graphs as ADTs
- Applications of Graphs

Terminology

- Definition:
  - A set of points that are joined by lines
- Graphs also represent the relationships among data items
- G = { V , E }; that is, a graph is a set of vertices and edges
- A subgraph consists of a subset of a graph's vertices and a subset of its edges

Terminology

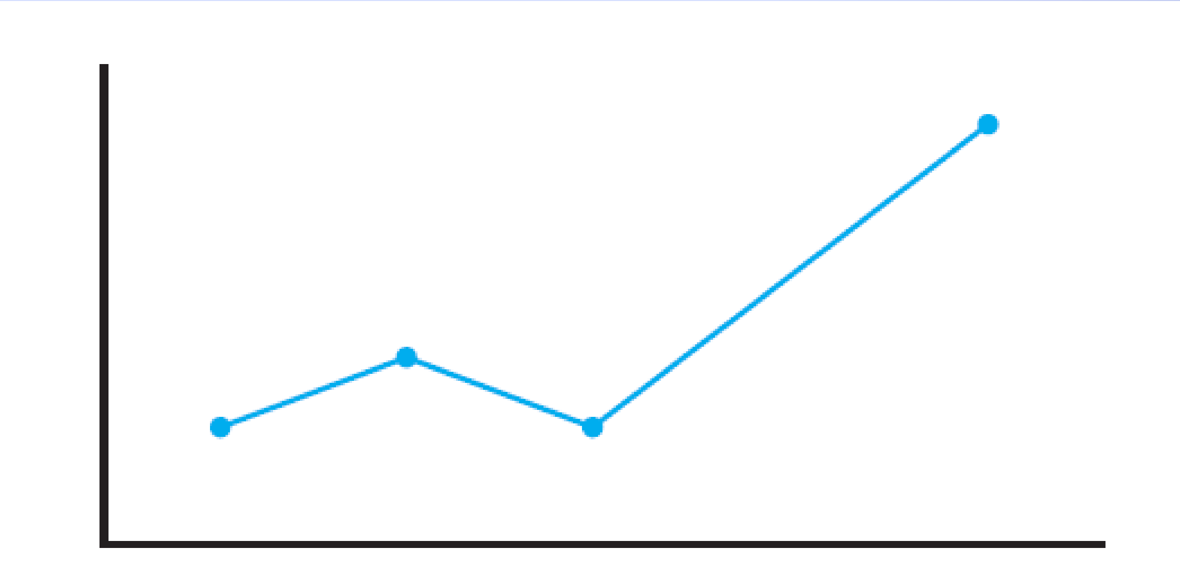


FIGURE 20-1 An ordinary line graph

Terminology

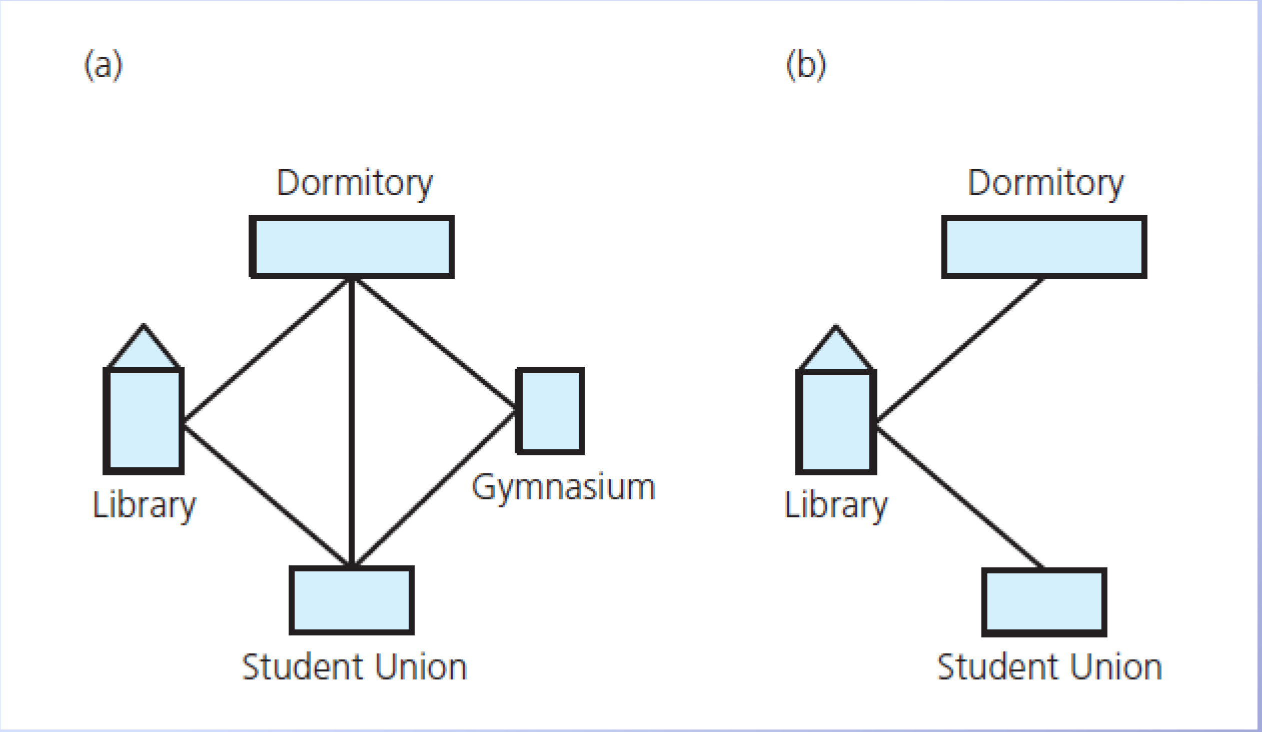


FIGURE 20-2 (a) A campus map as a graph; (b) a subgraph

Terminology

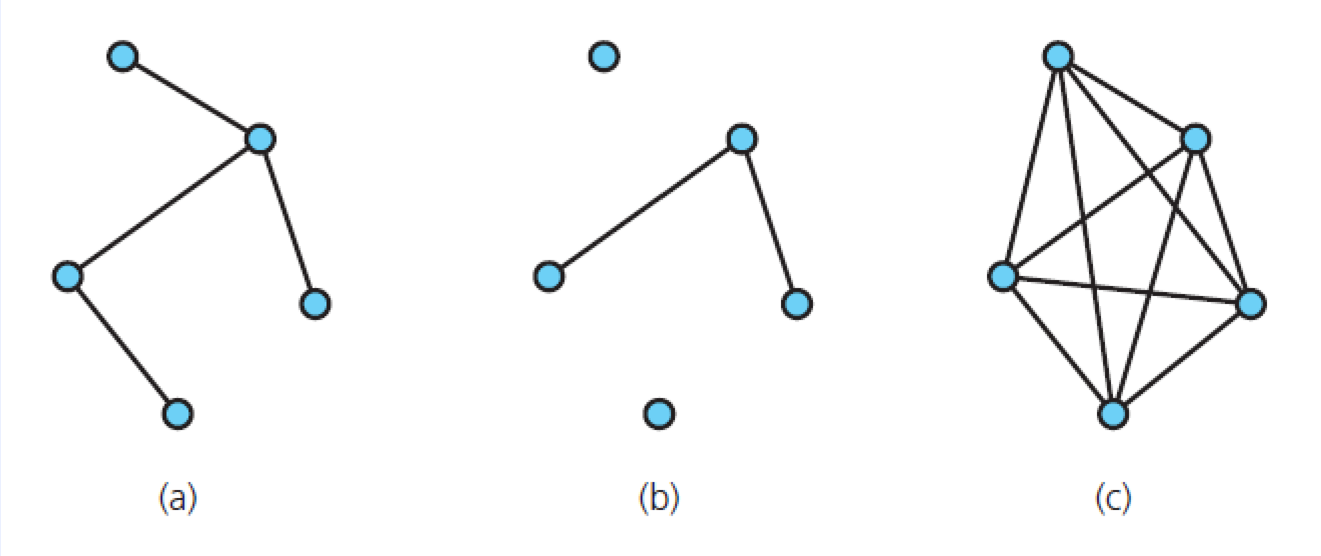


FIGURE 20-3 Graphs that are (a) connected; (b) disconnected; and (c) complete

Terminology

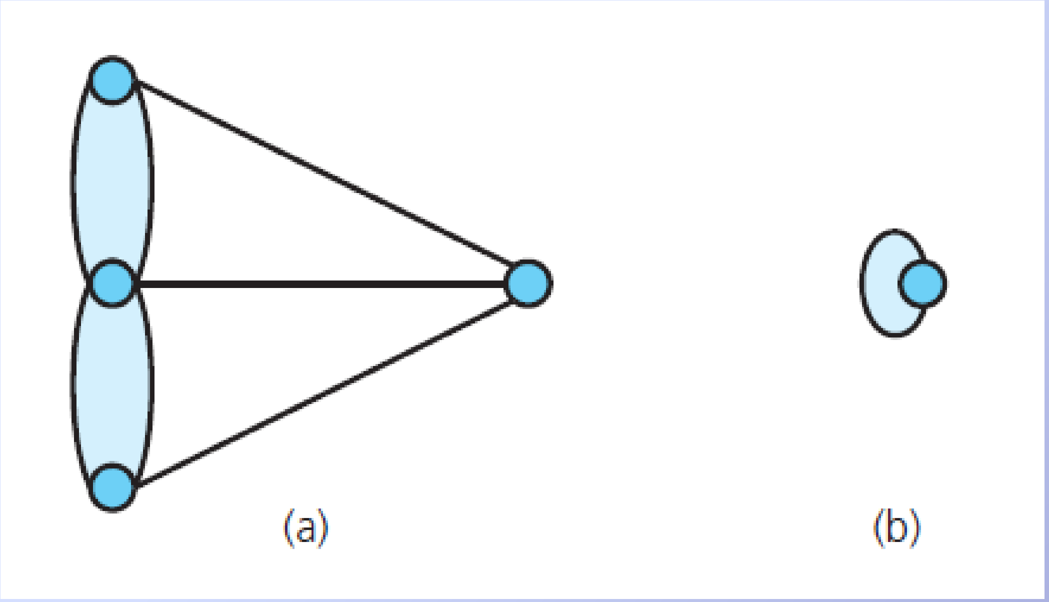


FIGURE 20-4 (a) A multigraph is not a graph;
(b) a self edge is not allowed in a graph

Terminology

- Simple path: passes through vertex only once
- Cycle: a path that begins and ends at same vertex
- Simple cycle: cycle that does not pass through other vertices more than once
- Connected graph: each pair of distinct vertices has a path between them

Terminology

- Complete graph: each pair of distinct vertices has an edge between them
- Graph cannot have duplicate edges between vertices
  - Multigraph: does allow multiple edges
- When labels represent numeric values, graph is called a weighted graph

Terminology

- Undirected graphs: edges do not indicate a direction
- Directed graph, or digraph: each edge has a direction

Terminology

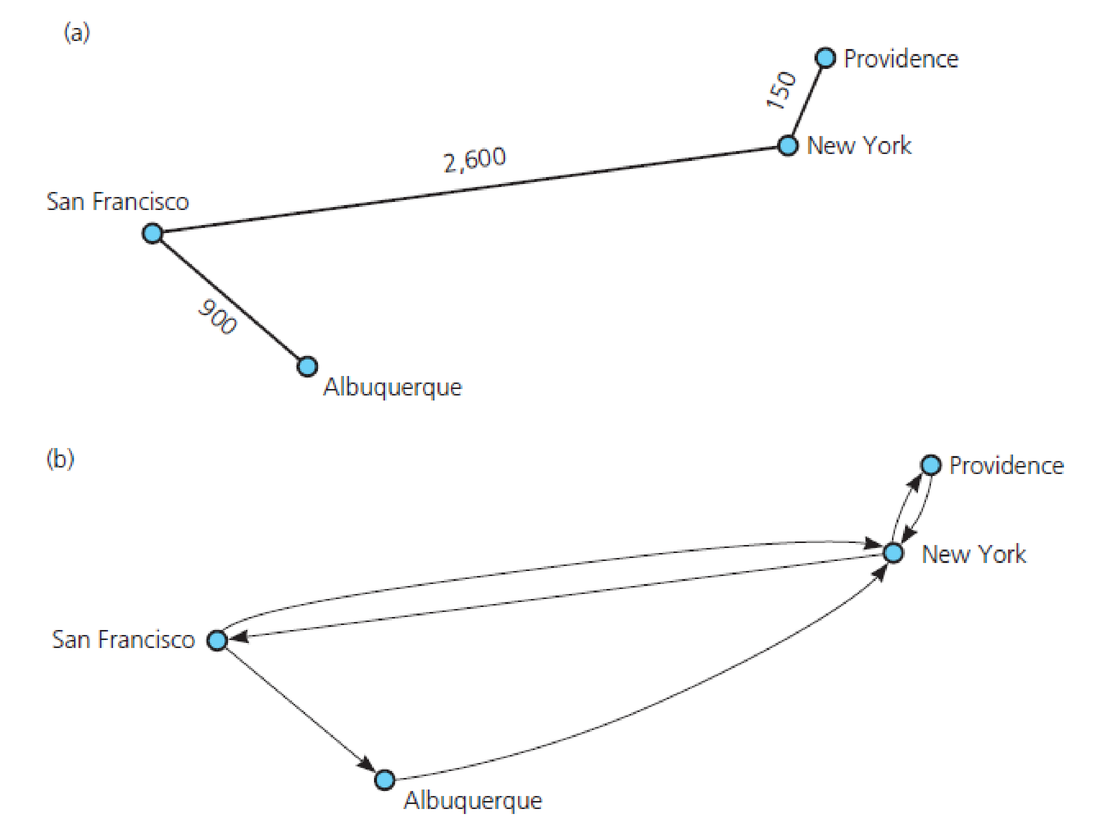


FIGURE 20-5 (a) A weighted graph; (b) a directed graph

Graphs as ADTs

ADT graph operations

- Test whether graph is empty.
- Get number of vertices in a graph.
- Get number of edges in a graph.
- See whether edge exists between two given vertices.
- Insert vertex in graph whose vertices have distinct values that differ from new vertex's value.

Graphs as ADTs

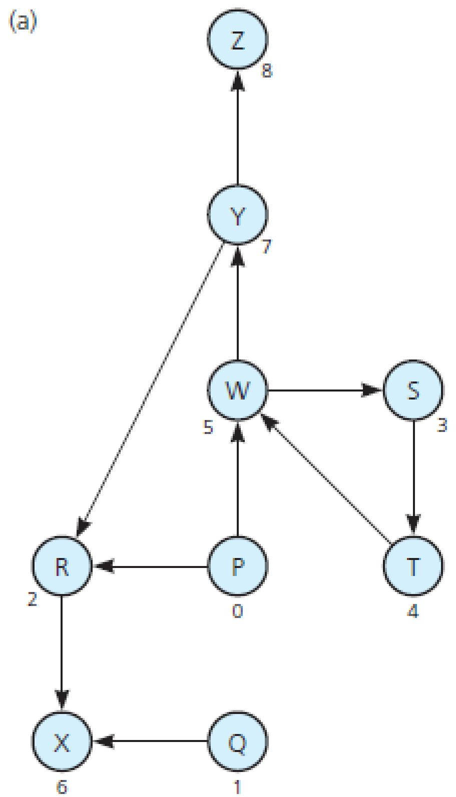
ADT graph operations, ctd.

- Insert edge between two given vertices in graph.
- Remove specified vertex from graph and any edges between the vertex and other vertices.
- Remove edge between two vertices in graph.
- Retrieve from graph vertex that contains given value.

- View interface for undirected, connected graphs, Listing 20-1

.htm code listing files must be in the same folder as the .ppt files for these links to work

Implementing Graphs

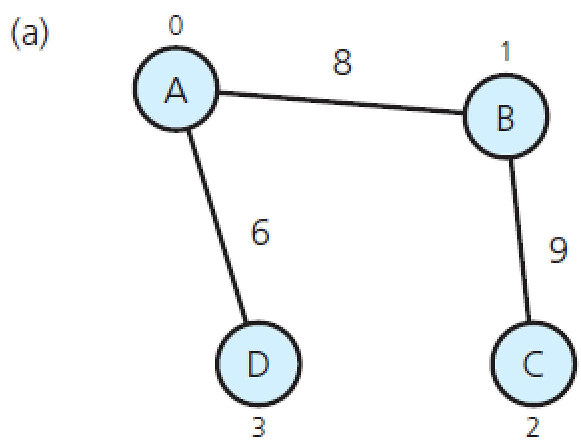


(b)

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

FIGURE 20-6 (a) A directed graph and (b) its adjacency matrix

Implementing Graphs



(b)

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

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

Implementing Graphs

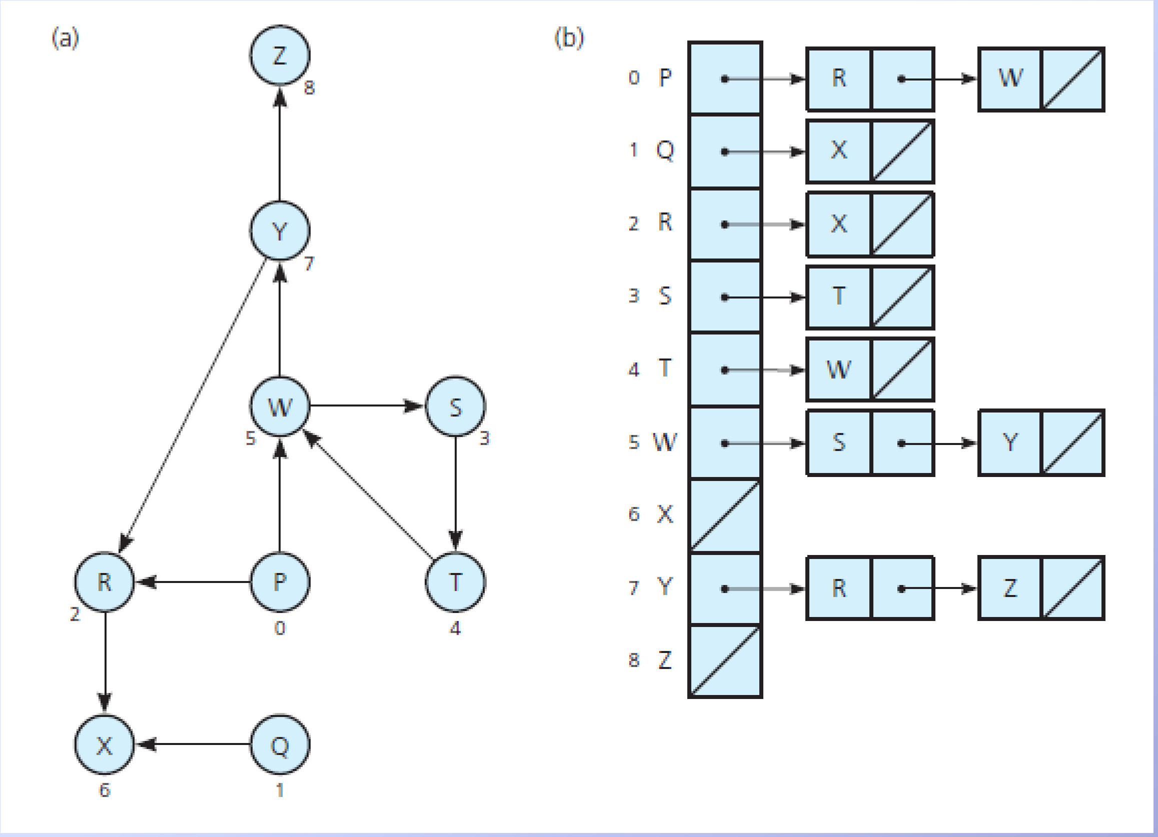


FIGURE 20-8 (a) A directed graph and (b) its adjacency list

Implementing Graphs

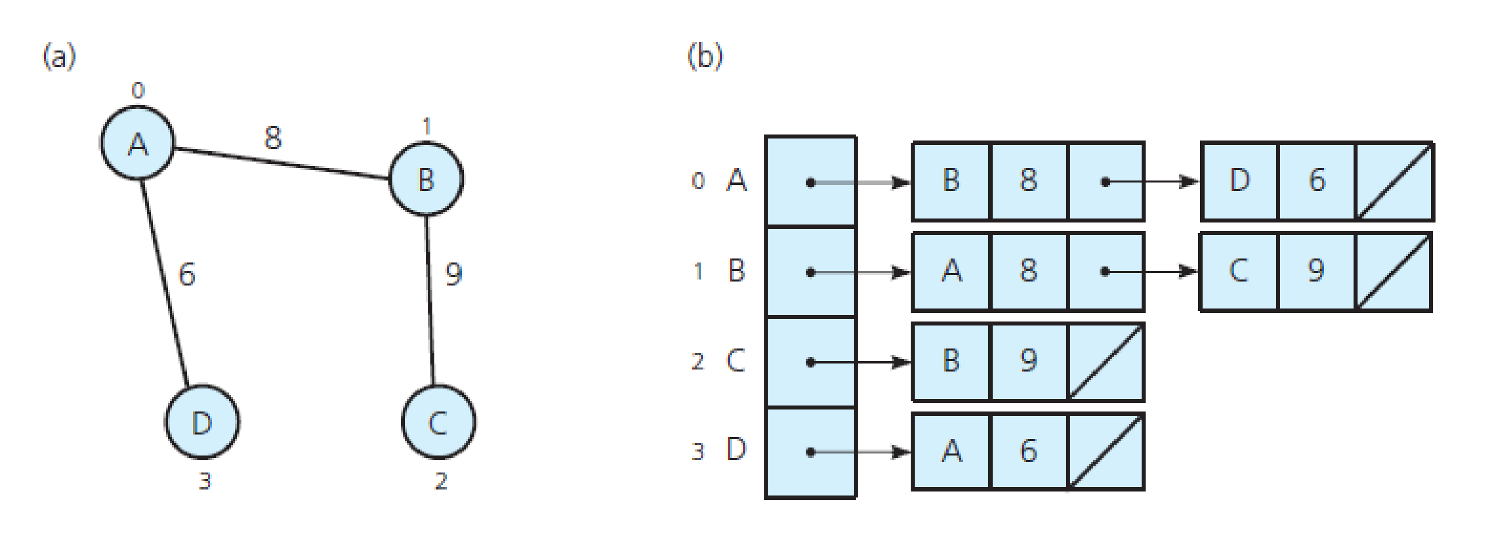


FIGURE 20-9 (a) A weighted undirected graph and (b) its adjacency list

Graph Traversals

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

Depth-First Search

- Goes as far as possible from a vertex before backing up
- Recursive 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)
```

Depth-First Search

- Iterative algorithm, using a stack

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Iterative version.
dfs(v: Vertex)

   s= a new empty stack

   // Push v onto the stack and mark it
   s.push(v)
   Mark v as visited

   // Loop invariant: there is a path from vertex v at the
   // bottom of the stack s to the vertex at the top of s
   while (!s.isEmpty())
```

Depth-First Search

- Iterative algorithm, using a stack, ctd.

```
{
   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

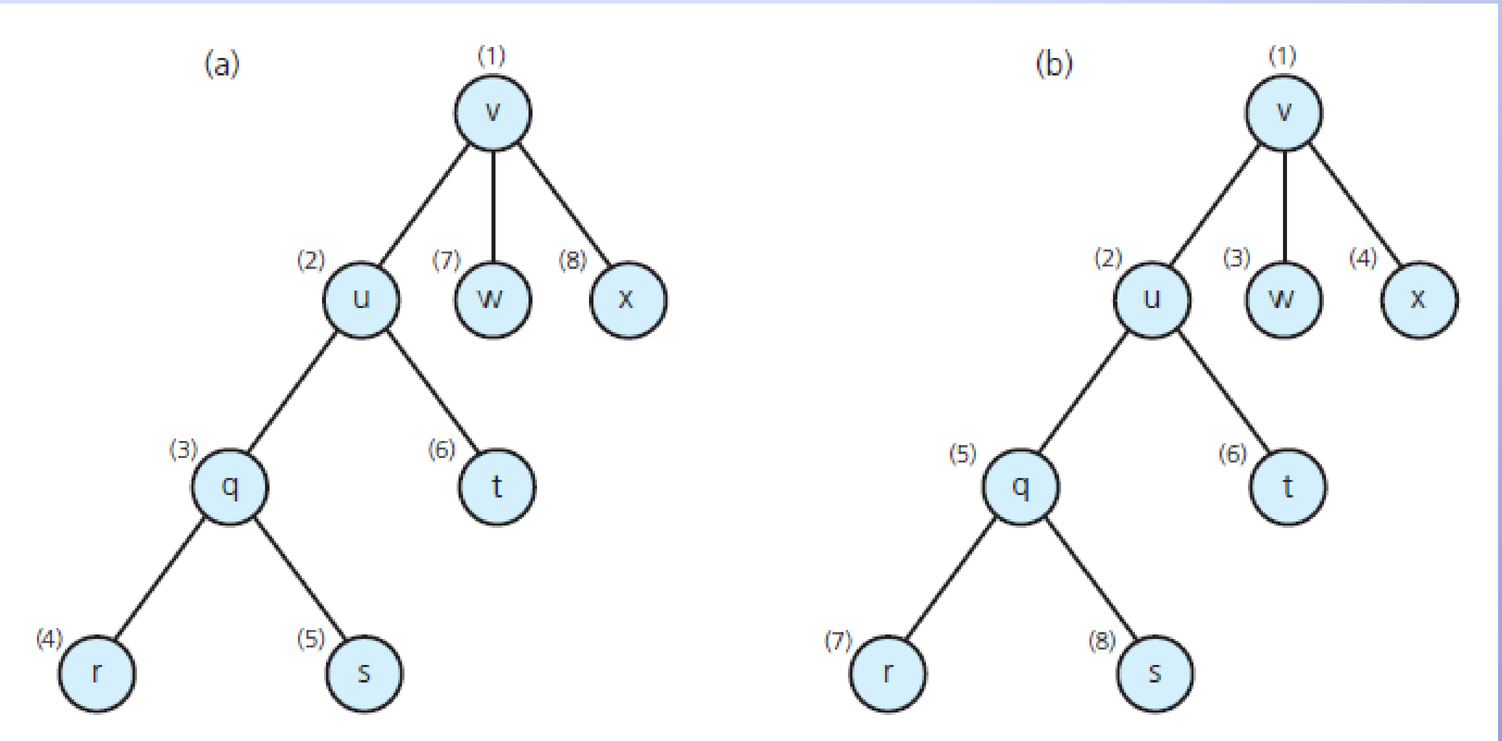


FIGURE 20-10 Visitation order for (a) a depth-first search; (b) a breadth-first search

Depth-First Search

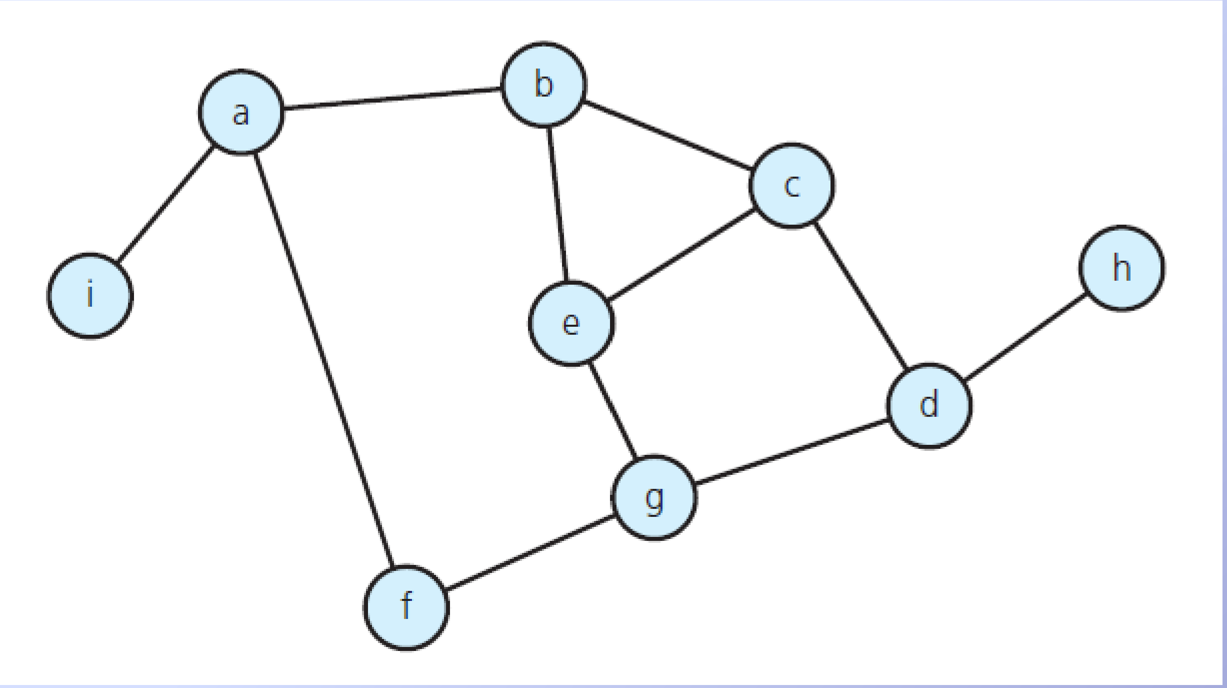


FIGURE 20-11 A connected graph with cycles

Depth-First Search

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)*

FIGURE 20-12 The results of a depth-first traversal, beginning at vertex a , of the graph in Figure 20-11

Breadth-First Search

- Visits all vertices adjacent to vertex before going forward
  - See Figure 20-10b
- Breadth-first search uses a queue

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Iterative version.
bfs(v: Vertex)

   q = a new empty queue

   // Add v to queue and mark it
   q.enqueue(v)
   Mark v as visited

   while (!q.isEmpty())
   {
```

```
while (!q.isEmpty())
{
   q.dequeue(w)

   // Loop invariant: there is a path from vertex w to every vertex in th
   for (each unvisited vertex u adjacent to w)
   {
      Mark u as visited
      q.enqueue(u)
   }
}
```

Breadth-First Search

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)

FIGURE 20-13 The results of a breadth-fi rst traversal, beginning at vertex a, of the graph in Figure 20-11

Applications of Graphs

- Topological Sorting

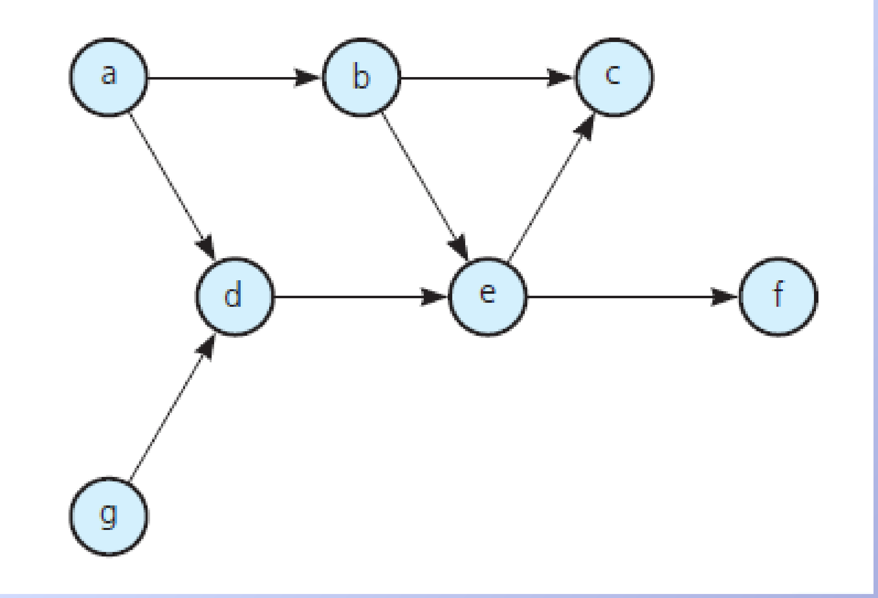


FIGURE 20-14 A directed graph without cycles

Applications of Graphs

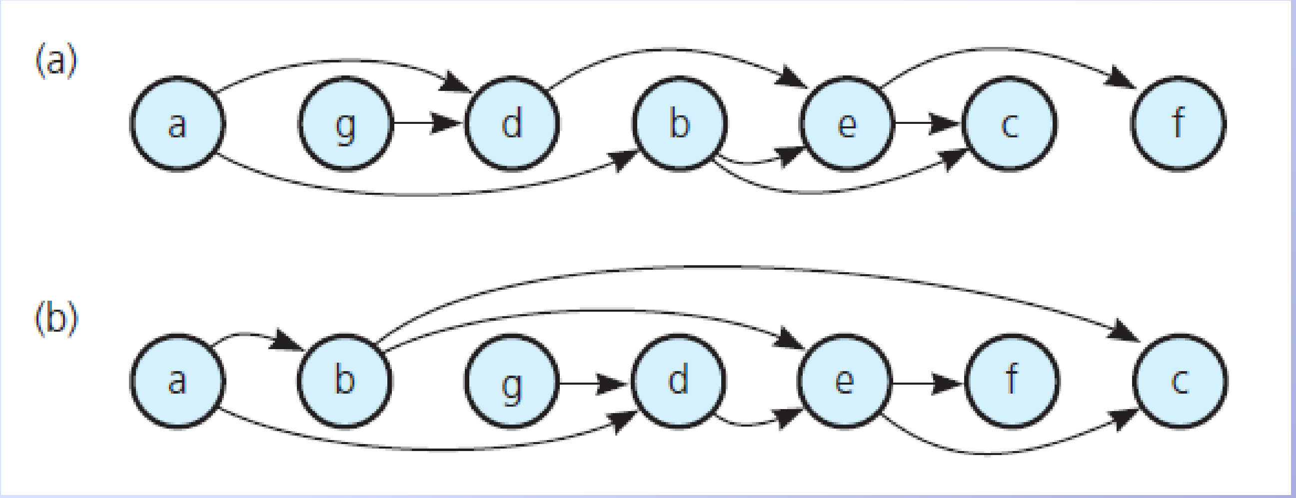


FIGURE 20-15 The graph in Figure 20-14 arranged according to the topological orders (a) *a, g, d, b, e, c, f* and (b) *a, b, g, d, e, f, c*

Applications of Graphs

- Topological sorting algorithm

```
// Arranges the vertices in graph theGraph into a
// topological order and places them in list aList.
topSort1(theGraph: Graph, aList: List)

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

Applications of Graphs

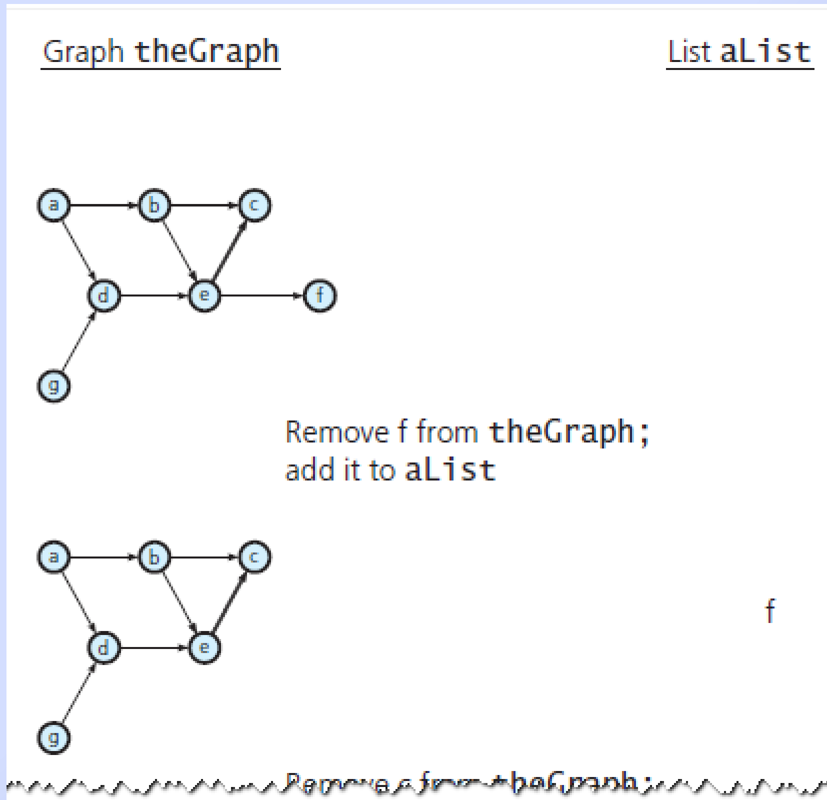


FIGURE 20-16 A trace of topSort1 for the graph in Figure 20-14

Applications of Graphs

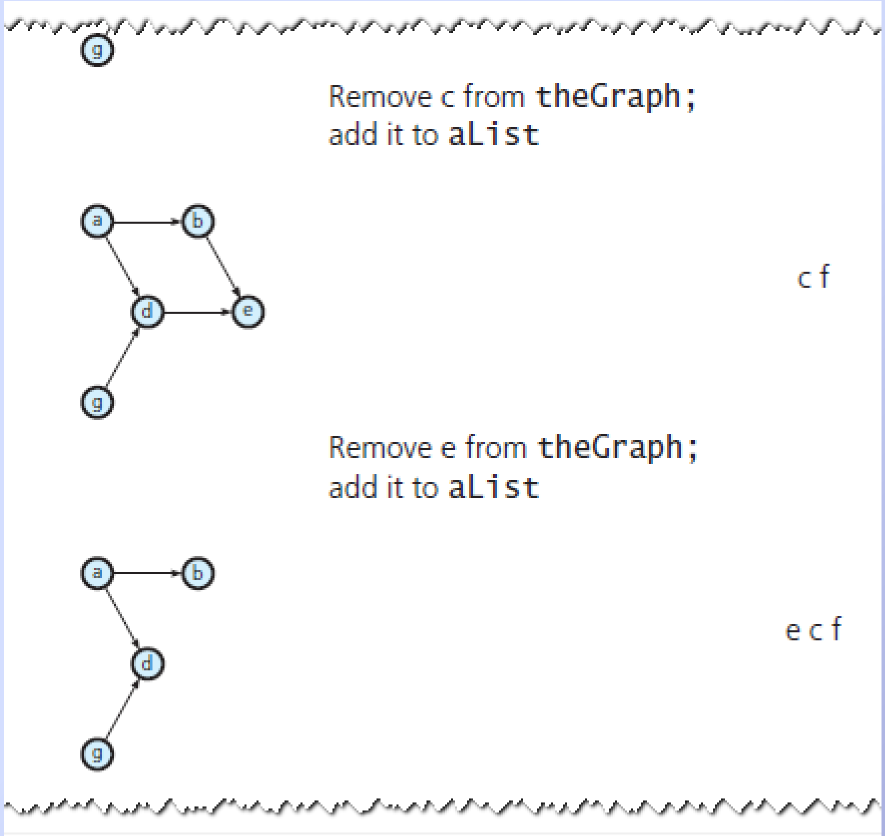


FIGURE 20-16 A trace of topSort1 for the graph in Figure 20-14

Applications of Graphs

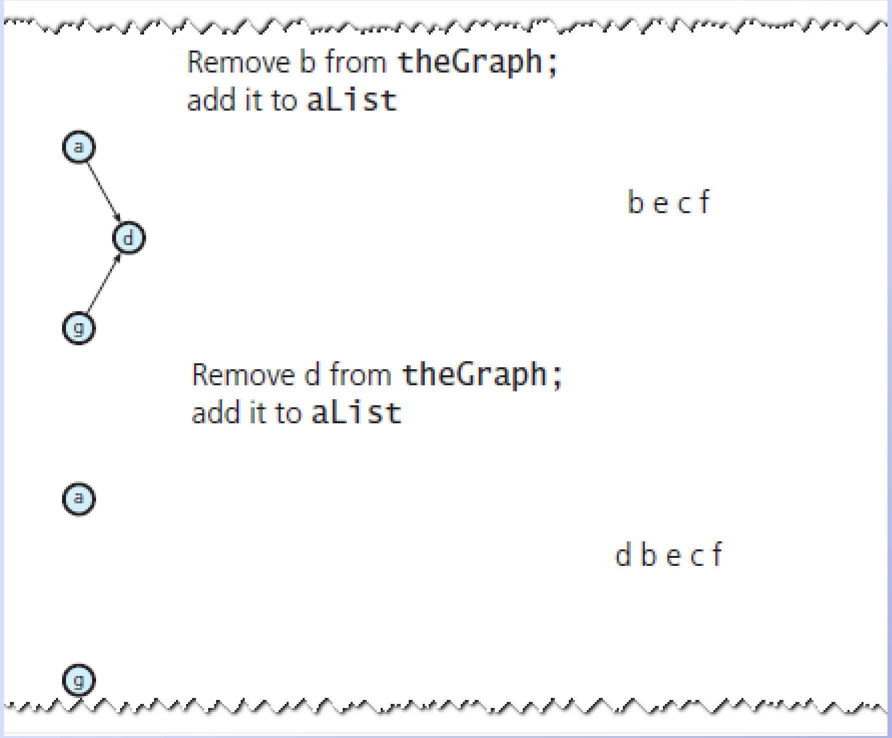


FIGURE 20-16 A trace of topSort1 for the graph in Figure 20-14

Applications of Graphs

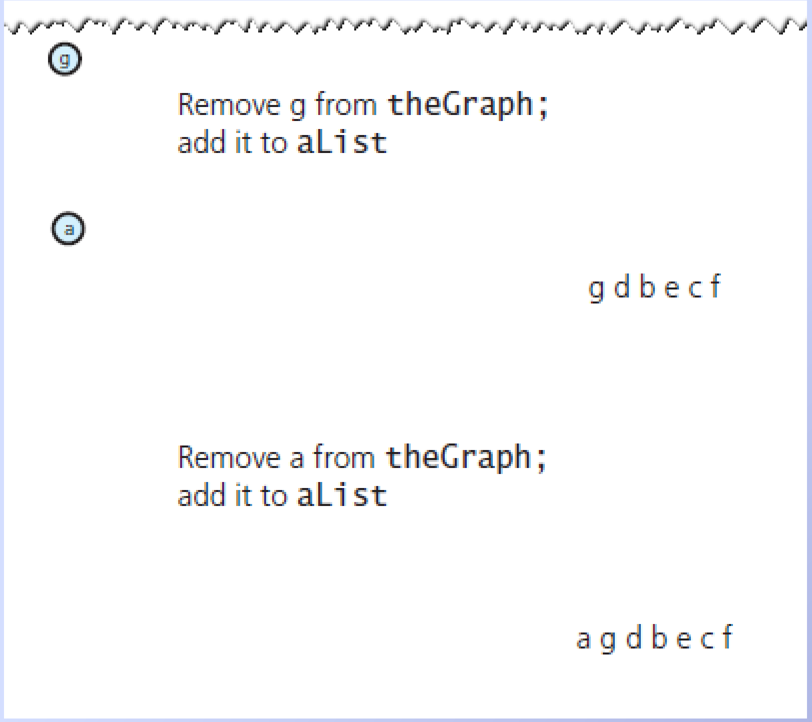


FIGURE 20-16 A trace of topSort1 for the graph in Figure 20-14

Applications of Graphs

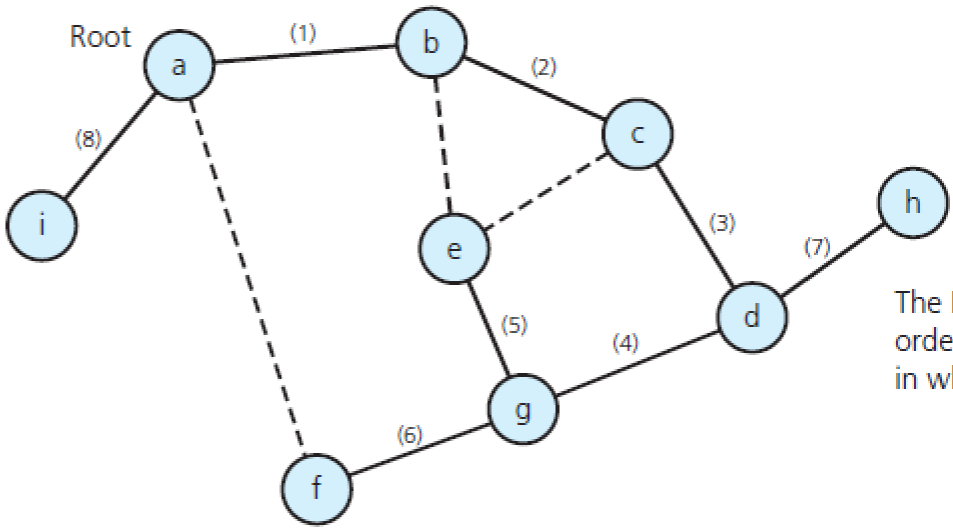
Action	Stack s (bottom to top)	List aList (beginning to end)
Push a	a	
Push g	a g	
Push d	a g d	
Push e	a g d e	c
Push c	a g d e c	c
Pop c, add c to aList	a g d e	f c
Push f	a g d e f	e f c
Pop f, add f to aList	a g d e	d e f c
Pop e, add e to aList	a g d	g d e f c
Pop d, add d to aList	a g	g d e f c
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 20-17 A trace of topSort2 for the graph in Figure 20-14

Spanning Trees

- Tree: an undirected connected graph without cycles
- Observations about undirected graphs
  1. Connected undirected graph with n vertices must have at least n – 1 edges.
  2. Connected undirected graph with n vertices, *exactly* n – 1 edges cannot contain a cycle
  3. A connected undirected graph with n vertices, *more than* n – 1 edges must contain at least one cycle

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 20-20 The DFS spanning tree rooted at vertex a for the graph in Figure 20-11

Spanning Trees

- DFS spanning tree algorithm

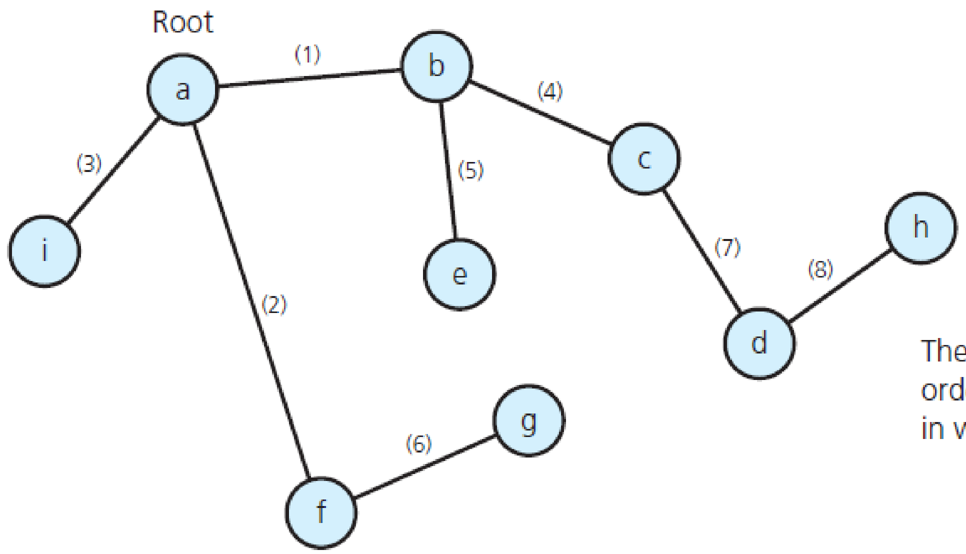
```
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using depth-first search:
// Recursive version.
dfsTree(v: Vertex)

   Mark v as visited

   for (each unvisited vertex u adjacent to v)
   {
      Mark the edge from u to v
      dfsTree(u)
   }
```

Spanning Trees

- BFS spanning tree algorithm

```
// Forms a spanning tree for a connected undirected graph
// beginning at vertex v by using breadth-first search:
// Iterative version.
bfsTree(v: Vertex)

   q = a new empty queue

   // Add v to queue and mark it
   q.enqueue(v)
   Mark v as visited

   while (!q.isEmpty())
   {
      q.dequeue(w)

      // Loop invariant: there is a path from vertex w to
      // every vertex in the queue q
      for (each unvisited vertex u adjacent to w)
      {
         Mark u as visited
         Mark edge between w and u
         q.enqueue(u)
      }
   }
```

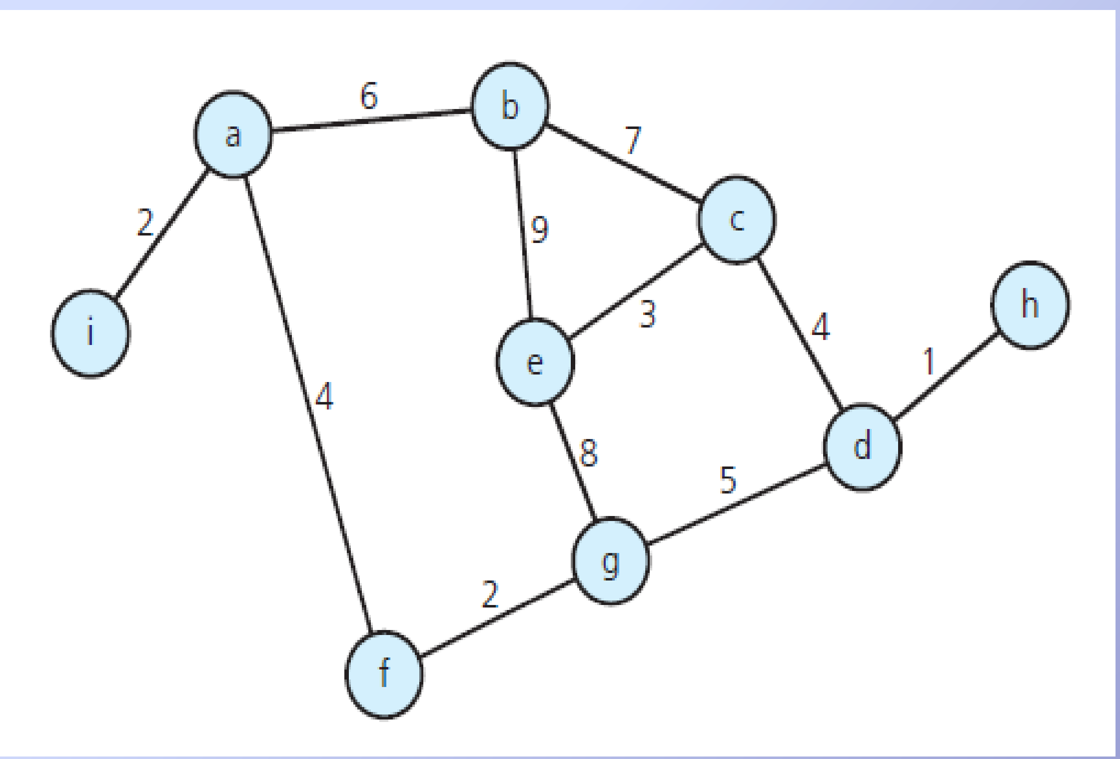
Spanning Trees



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 20-21 The BFS spanning tree rooted at vertex a for the graph in Figure 20-11

Minimum Spanning Trees



A minimum spanning tree of a connected undirected graph has a minimal edge-weight sum

FIGURE 20-22 A weighted, connected, undirected graph

Minimum Spanning Trees

- Minimum spanning tree algorithm

```
// Determines a minimum spanning tree for a weighted,
// connected, undirected graph whose weights are
// nonnegative, beginning with any vertex v.
primsAlgorithm(v: Vertex)

   Mark vertex v as visited and include it in the minimum spanning tree
   while (there are unvisited vertices)
   {
      Find the least-cost edge (v, u) from a visited vertex v to some unvisited vertex u
      Mark u as visited
      Add the vertex u and the edge (v, u) to the minimum spanning tree
   }
```

Minimum Spanning Trees

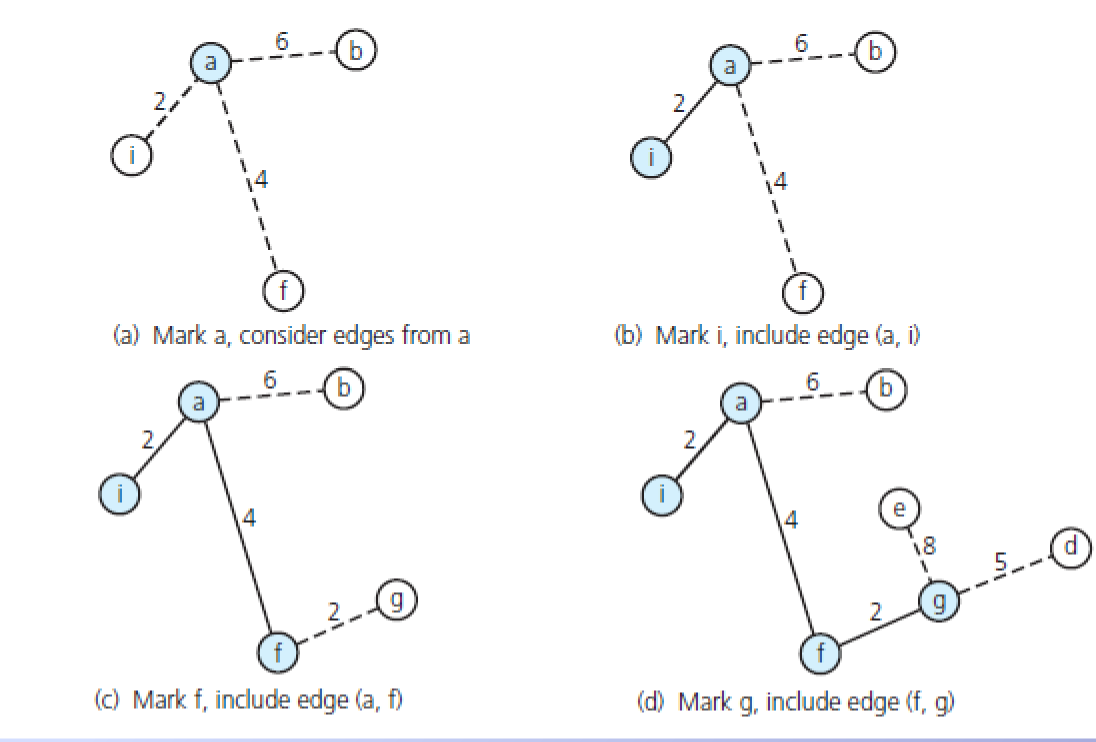


FIGURE 20-23 A trace of primsAlgorithm for the graph in Figure 20-22 , beginning at vertex a

Minimum Spanning Trees

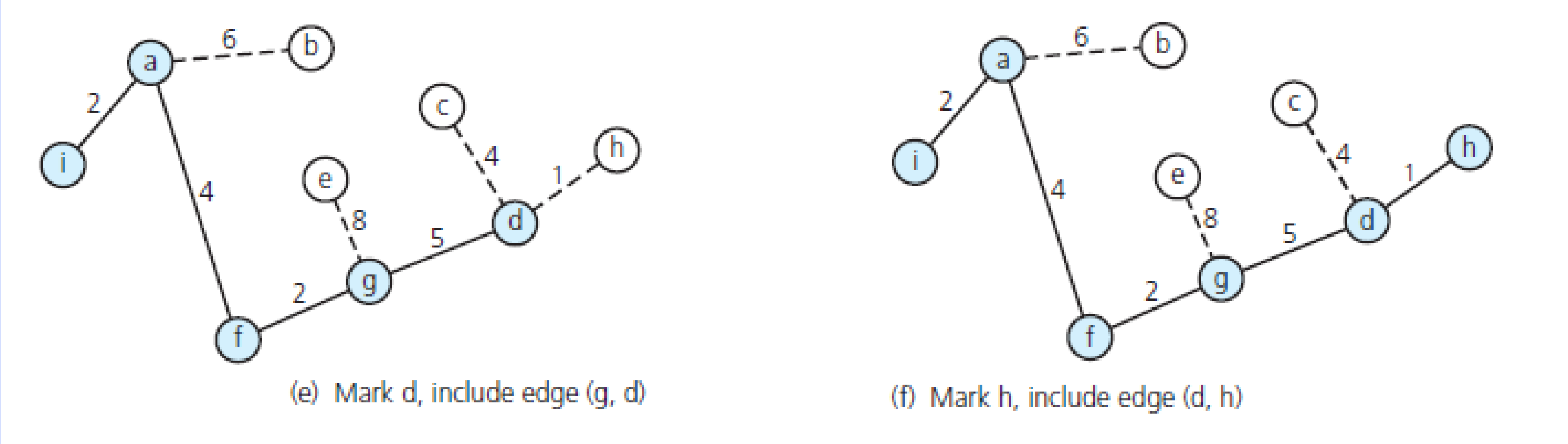


FIGURE 20-23 A trace of primsAlgorithm for the graph in Figure 20-22 , beginning at vertex a

Minimum Spanning Trees

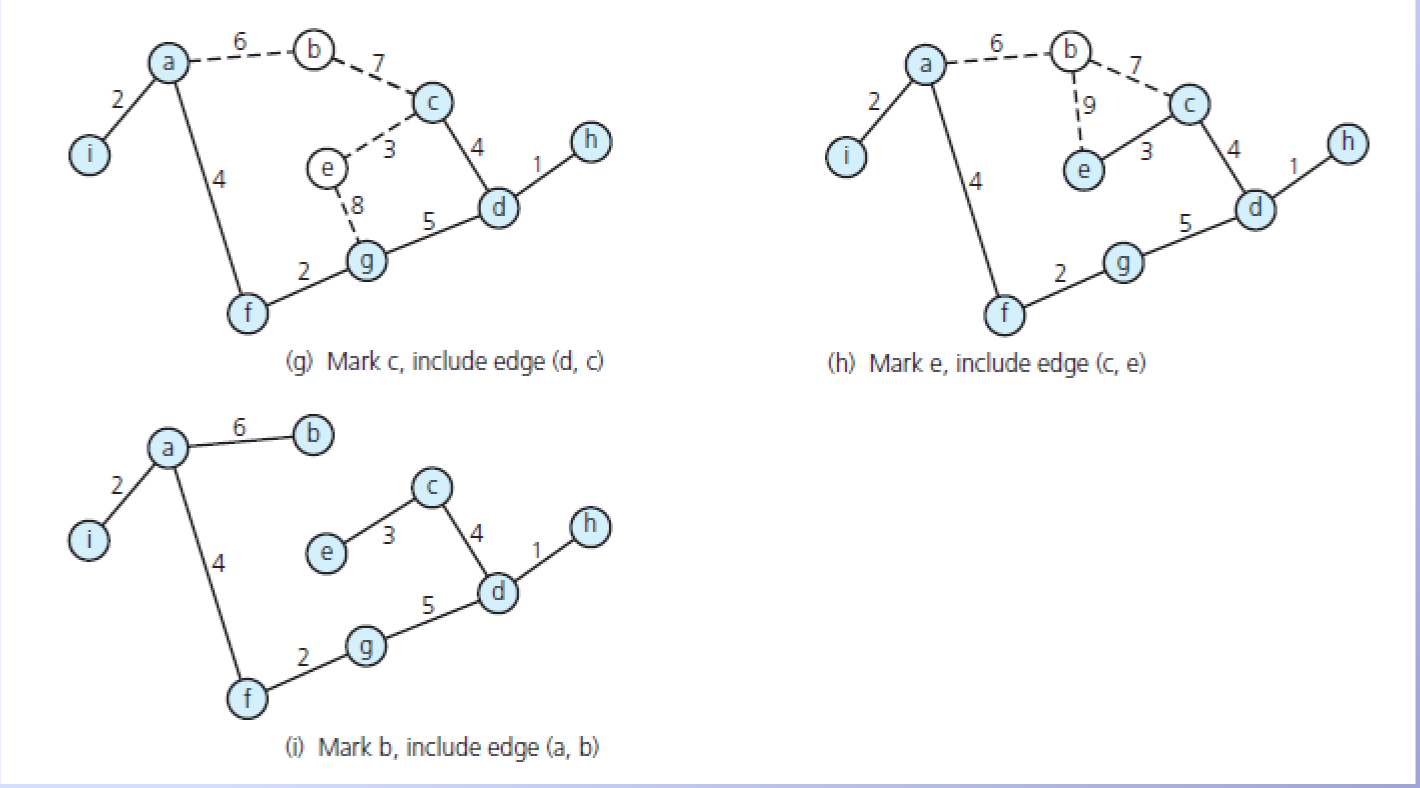
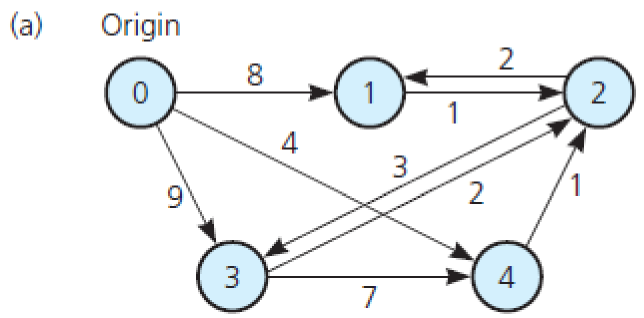


FIGURE 20-23 A trace of primsAlgorithm for the graph in Figure 20-22 , beginning at vertex a

Shortest Paths

- Shortest path between two vertices in a weighted graph has smallest edge-weight sum



(b)

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

FIGURE 20-24 (a) A weighted directed graph and (b) its adjacency matrix

Shortest Paths

- Dijkstra's shortest-path algorithm

```
// Finds the minimum-cost paths between an origin vertex
// (vertex 0) and all other vertices in a weighted directed
// graph theGraph; theGraph's weights are nonnegative.
shortestPath(theGraph: Graph, weight: WeightArray)

   // Step 1: initialization
   Create a set vertexSet that contains only vertex 0
   n = number of vertices in theGraph
   for (v = 0 through n - 1)
      weight[v] = matrix[0][v]

   // Steps 2 through n
   // Invariant: For v not in vertexSet, weight[v] is the
   // smallest weight of all paths from 0 to v that pass
   // through only vertices in vertexSet before reaching
   // v. For v in vertexSet, weight[v] is the smallest
   // weight of all paths from 0 to v (including paths
   // outside vertexSet) and the shortest path
```

Shortest Paths

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

FIGURE 20-25 A trace of the shortest-path algorithm applied to the graph in Figure 20-24 a

Shortest Paths

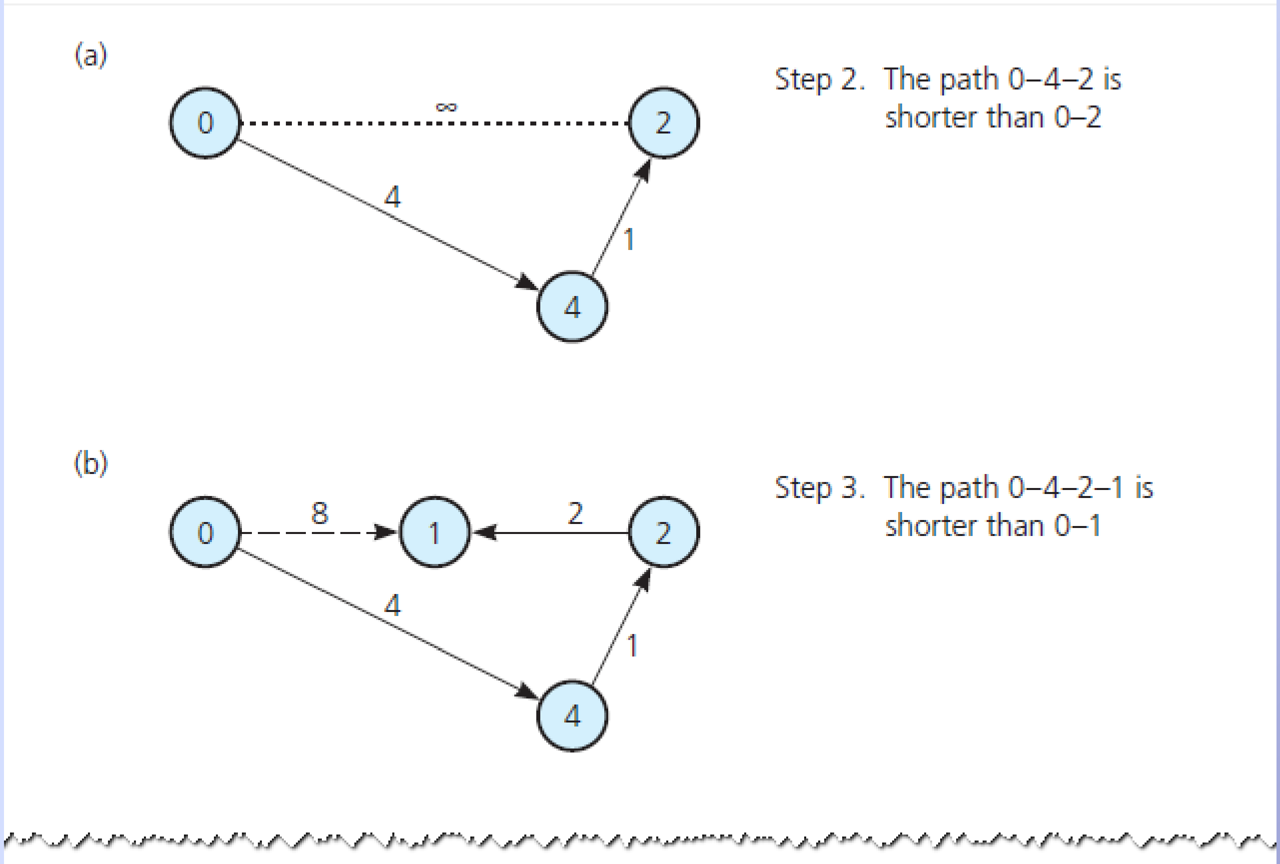


FIGURE 20-26 Checking weight [u] by examining the graph: (a) weight [2] in step 2; (b) weight [1] in step 3;

Shortest Paths

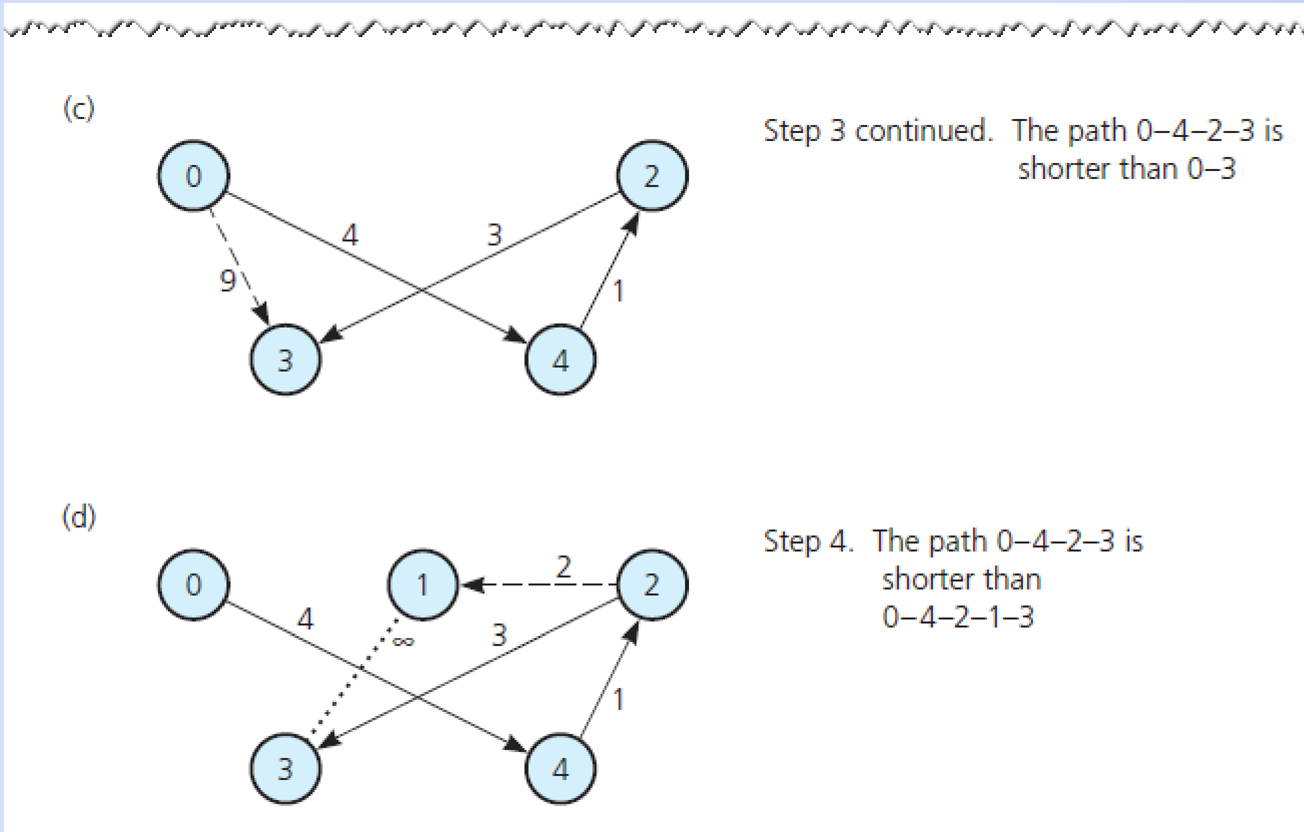


FIGURE 20-26 Checking weight [u] by examining the graph(c) weight [3] in step 3; (d) weight [3] in step 4

Shortest Paths

- Dijkstra’s shortest-path algorithm, ctd.

```
// smallest weight of all paths from 0 to v that pass
// through only vertices in vertexSet before reaching
// v. For v in vertexSet, weight[v] is the smallest
// weight of all paths from 0 to v (including paths
// outside vertexSet), and the shortest path
// from 0 to v lies entirely in vertexSet.
for (step = 2 through n)
{
   Find the smallest weight[v] such that v is not in vertexSet
   Add v to vertexSet

   // Check weight[u] for all u not in vertexSet
   for (all vertices u not in vertexSet)
      if (weight[u] > weight[v] + matrix[v][u])
         weight[u] = weight[v] + matrix[v][u]
}
```

Shortest Paths

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

FIGURE 20-25 A trace of the shortest-path algorithm applied to the graph in Figure 20-24 a

Shortest Paths

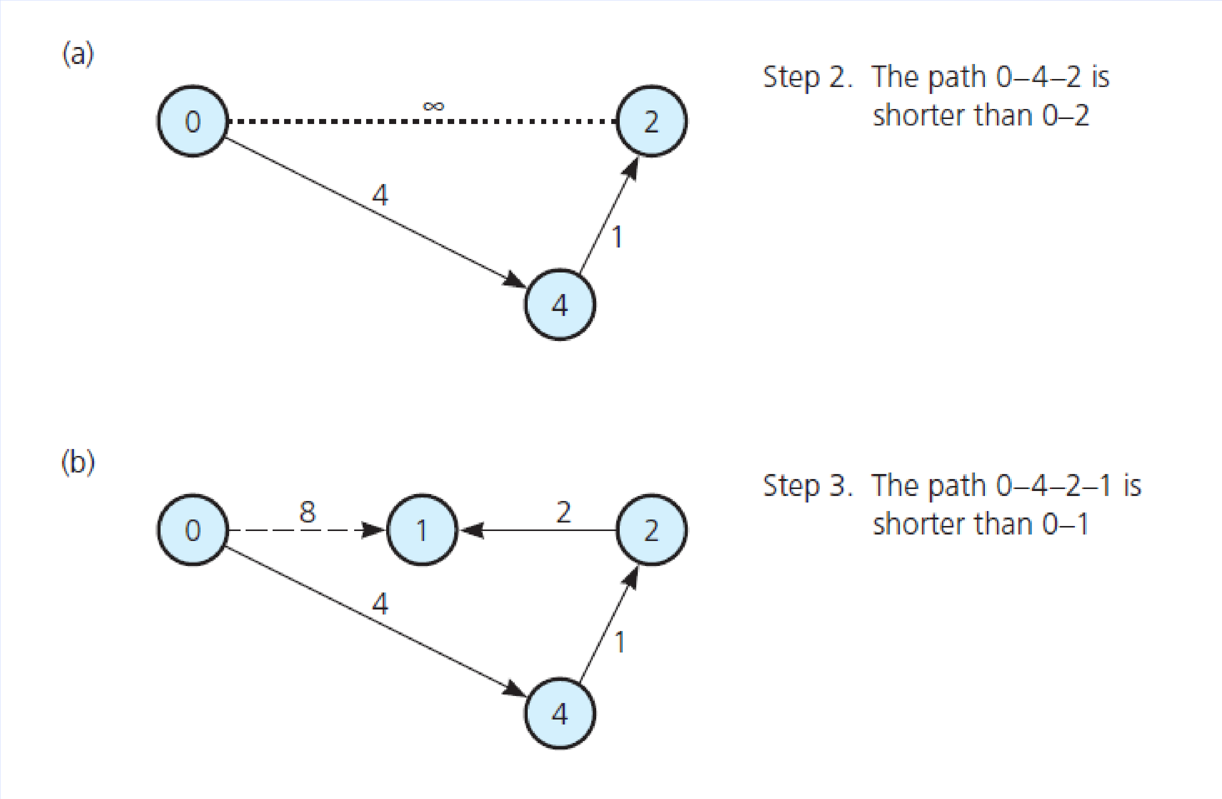


FIGURE 20-26 Checking weight [u] by examining the graph: (a) weight [2] in step 2; (b) weight [1] in step 3;

Shortest Paths

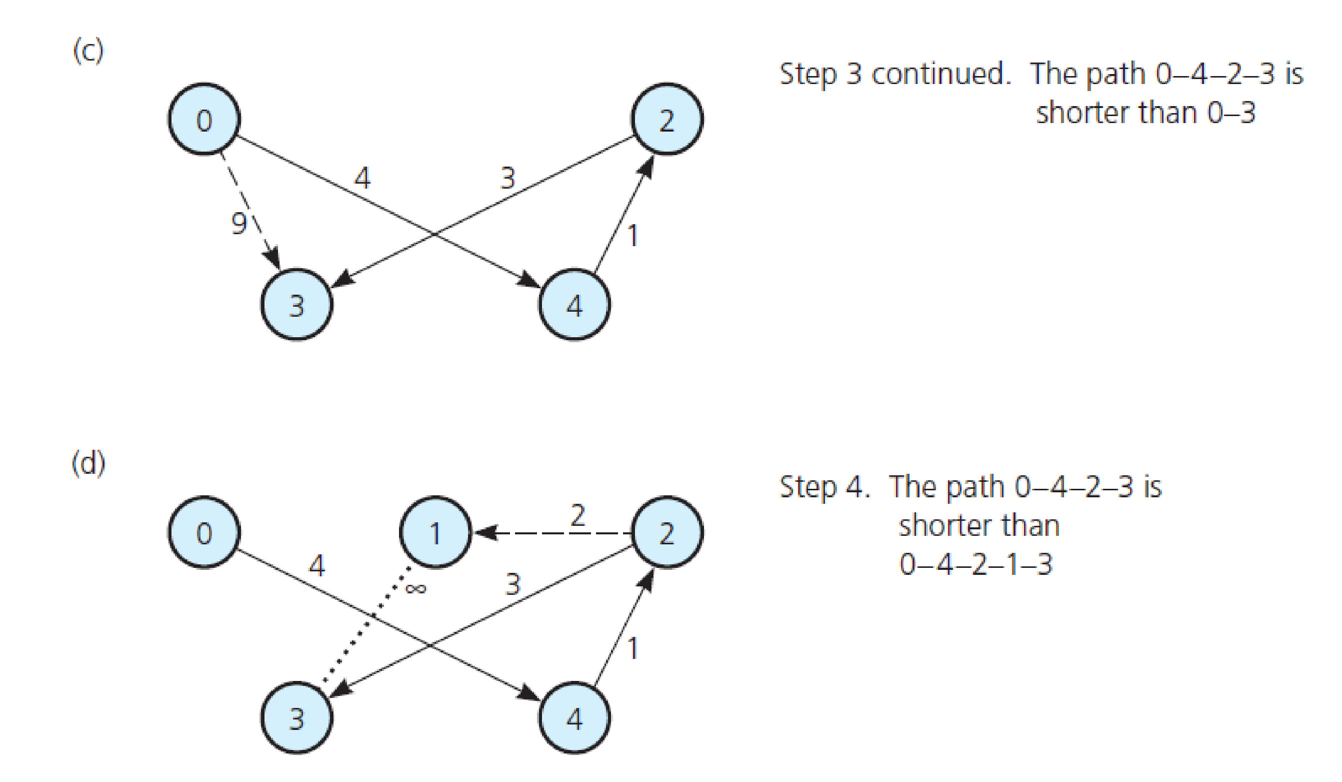


FIGURE 20-26 Checking weight [u] by examining the graph: (c) weight [3] in step 3; (d) weight [3] in step 4

Circuits

- Another name for a type of cycle common in statement of certain problems
- Circuits either visit every vertex once or visit every edge once
- An *Euler circuit* begins at a vertex v, passes through every edge exactly once, and terminates at v

Circuits

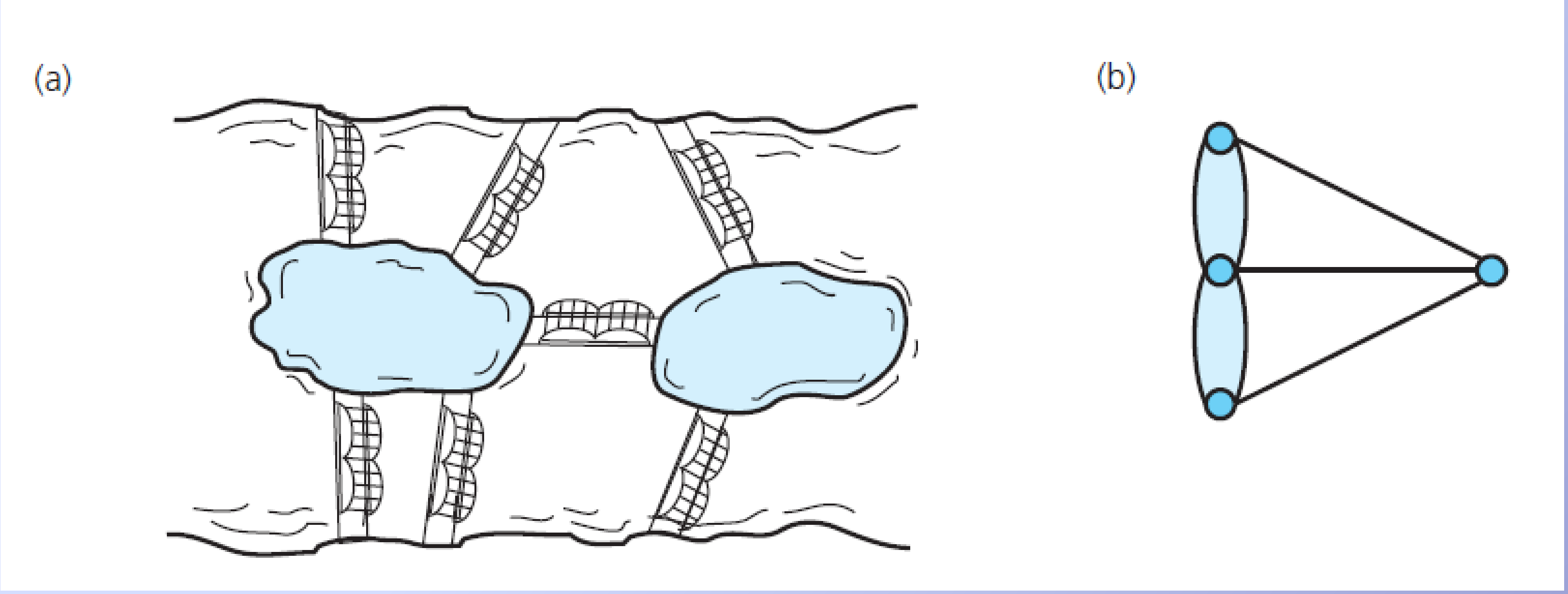


FIGURE 20-27 (a) Euler's bridge problem and (b) its multigraph representation

Circuits

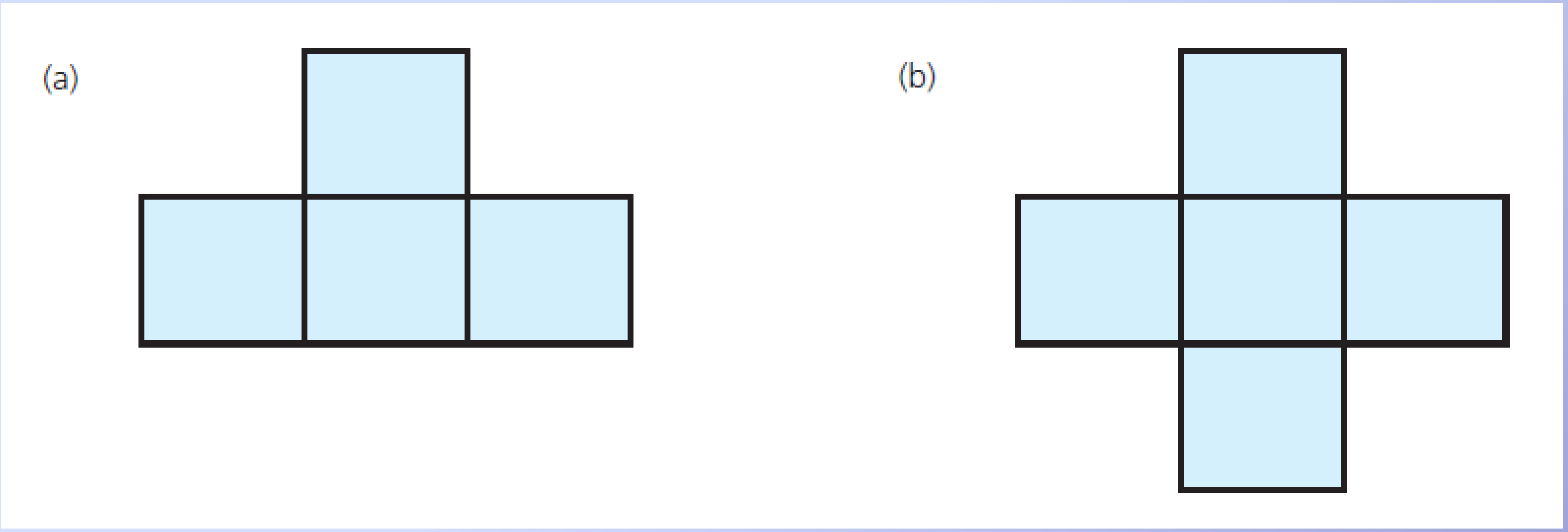


FIGURE 20-28 Pencil and paper drawings

Circuits

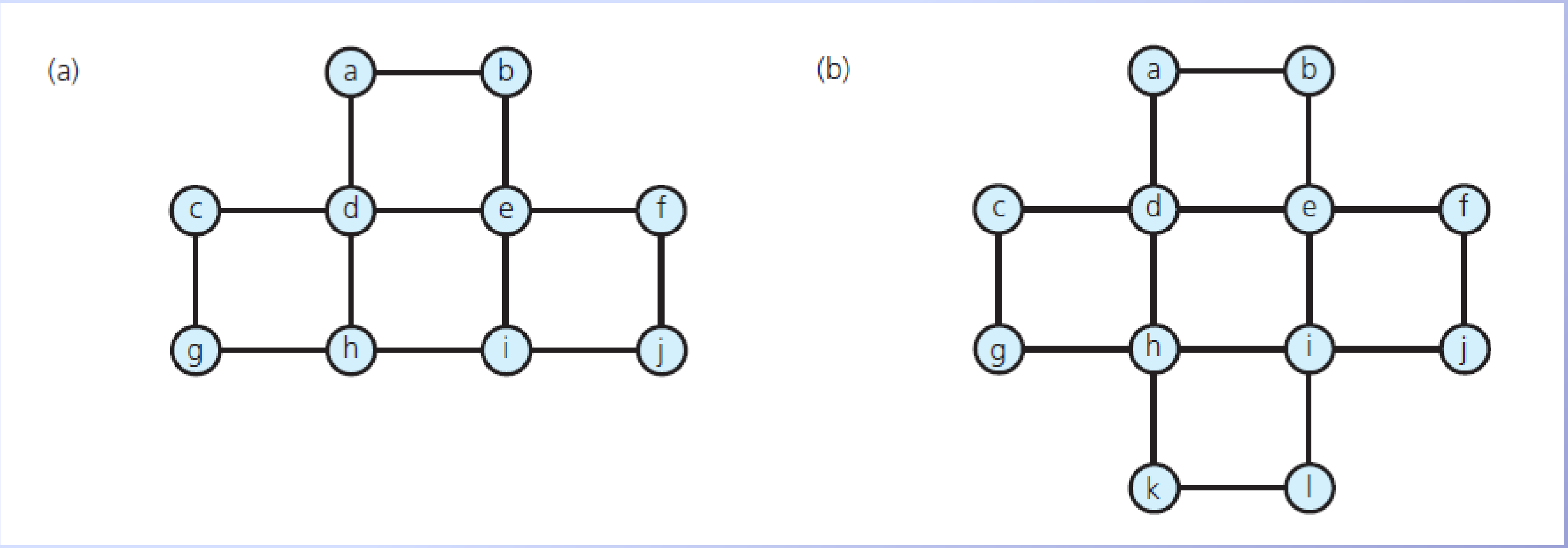


FIGURE 20-29 Connected undirected graphs based on the drawings in Figure 20-28

Circuits

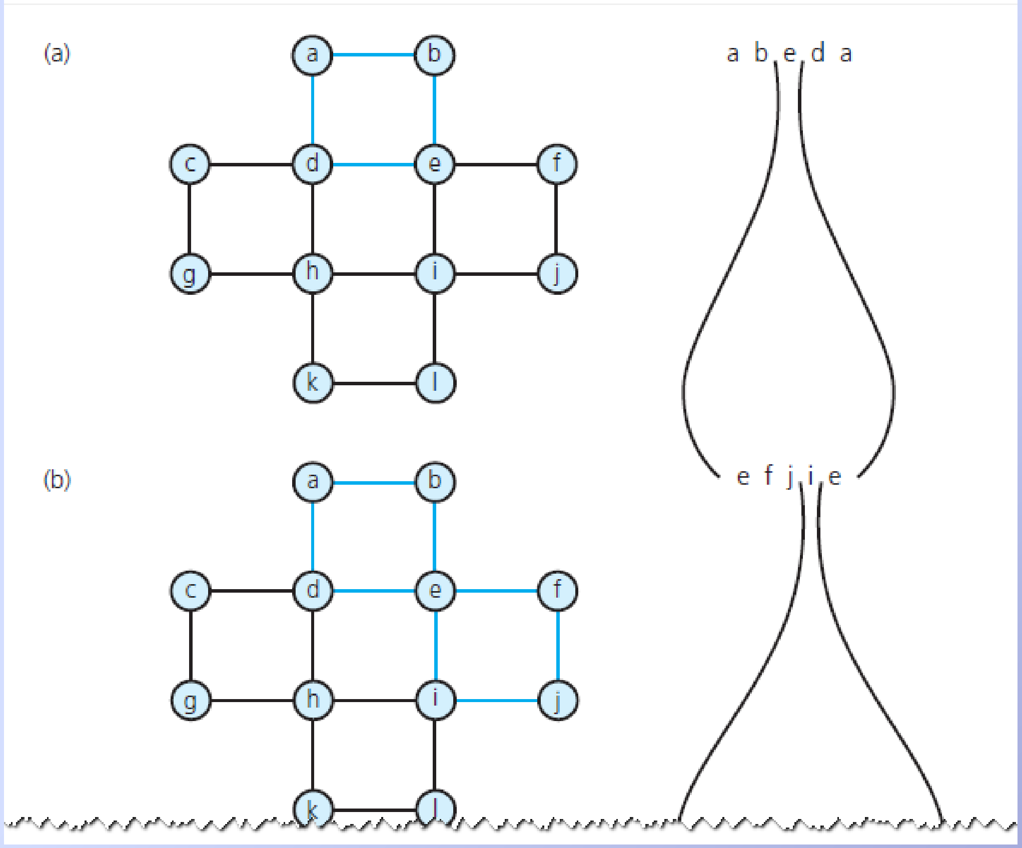


FIGURE 20-30 The steps to determine an Euler circuit for the graph in Figure 20-29 b

Circuits

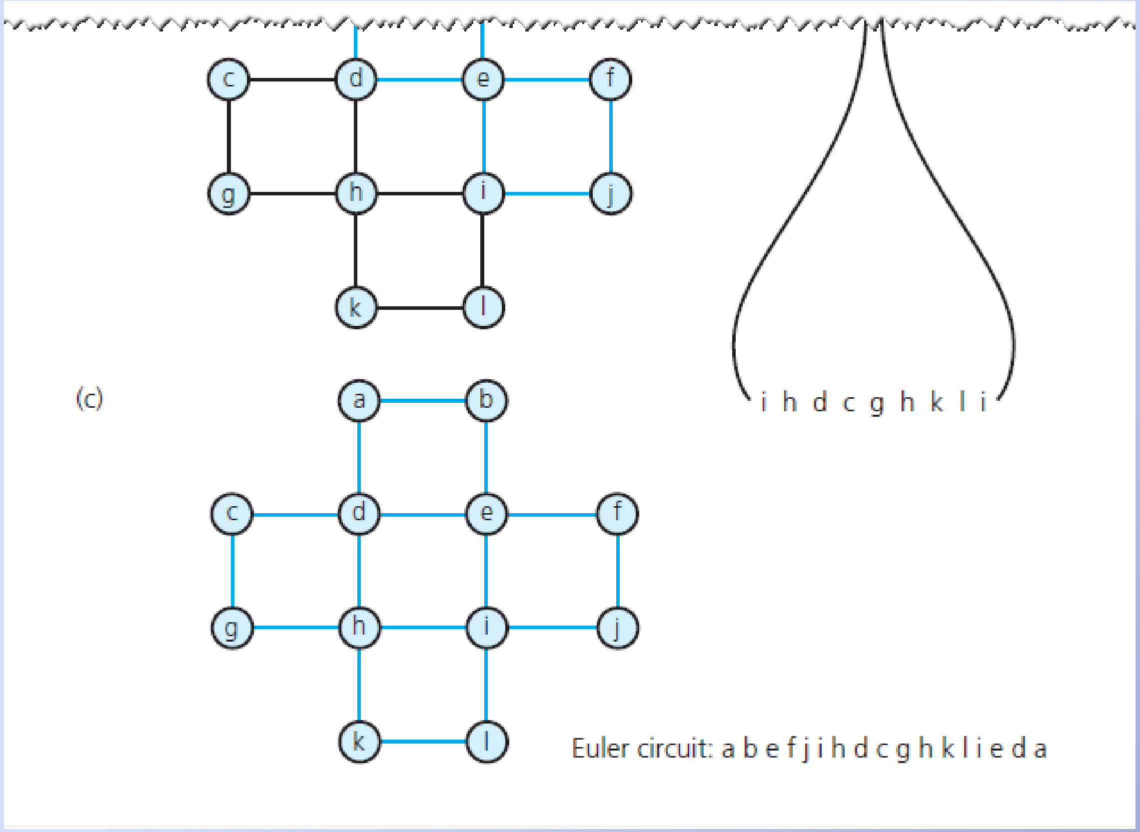


FIGURE 20-30 The steps to determine an Euler circuit for the graph in Figure 20-29 b

Some Difficult Problems

- Hamilton circuit
  - Path that begins at a vertex v, passes through every vertex in the graph exactly once, and terminates at v .
- The traveling salesperson problem
  - Variation of Hamilton circuit
  - Involves a weighted graph that represents a road map
  - Circuit traveled must be the least expensive

Some Difficult Problems

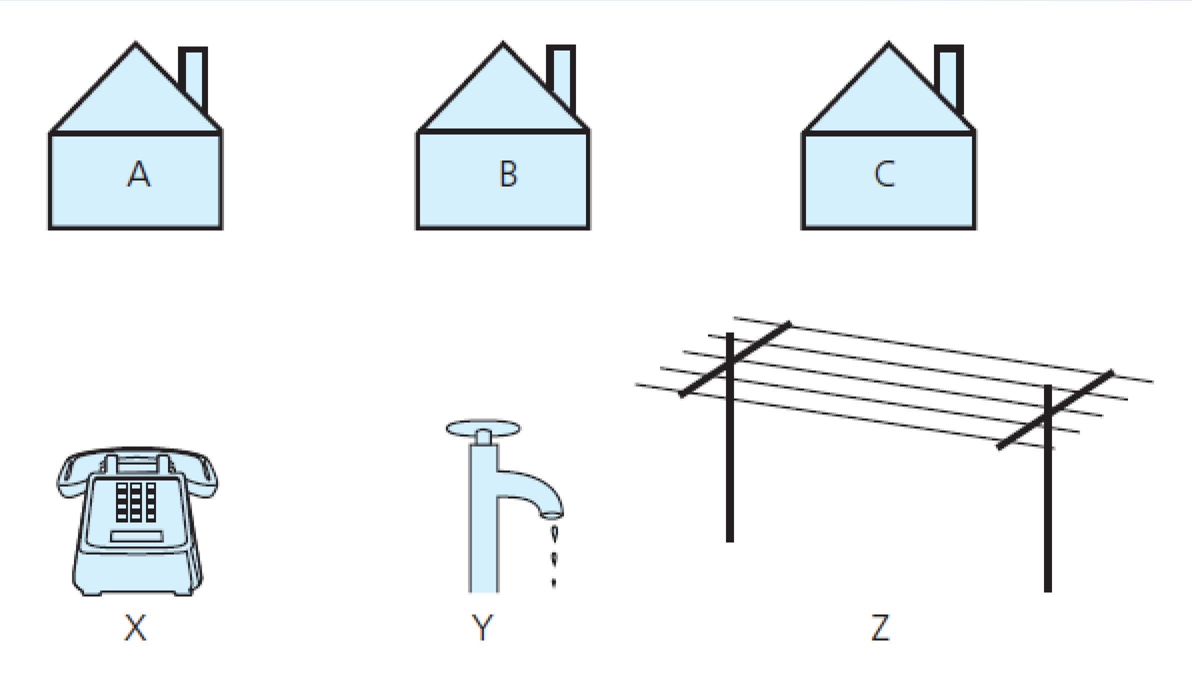


FIGURE 20-31 The three utilities problem

Some Difficult Problems

- Planar graph
  - Can draw it in a plane in at least one way so that no two edges cross
- The four-color problem
  - Given a planar graph, can you color the vertices so that no adjacent vertices have the same color, if you use at most four colors?

Some Difficult Problems

1. Describe the graphs in Figure 20-32 . For example, are they directed? Connected? Complete? Weighted?
2. Use the depth-first strategy and the breadth-first strategy to traverse the graph in Figure 20-32 a, beginning with vertex 0. List the vertices in the order in which each traversal visits them.

Some Difficult Problems

3. Write the adjacency matrix for the graph in Figure 20-32 a.
4. Add an edge to the directed graph in Figure 20-14 that runs from vertex d to vertex b. Write all possible topological orders for the vertices in this new graph.
5. Is it possible for a connected undirected graph with fi ve vertices and four edges to contain a simple cycle? Explain.

Some Difficult Problems

6. Draw the DFS spanning tree whose root is vertex 0 for the graph in Figure 20-33 .
7. Draw the minimum spanning tree whose root is vertex 0 for the graph in Figure 20-33 .
8. What are the shortest paths from vertex 0 to each vertex of the graph in Figure 20-24 a? (Note the weights of these paths in Figure 20-25 .)

Some Difficult Problems

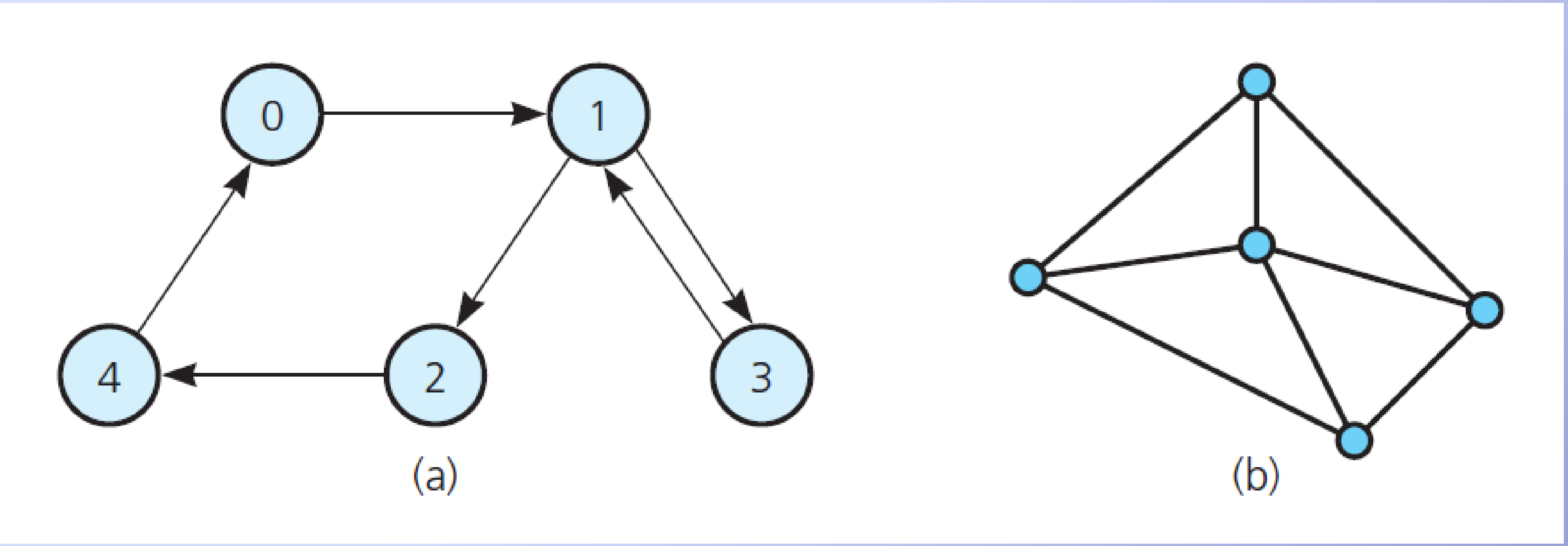


FIGURE 20-32 Graphs for Checkpoint Questions 1, 2, and 3

Some Difficult Problems

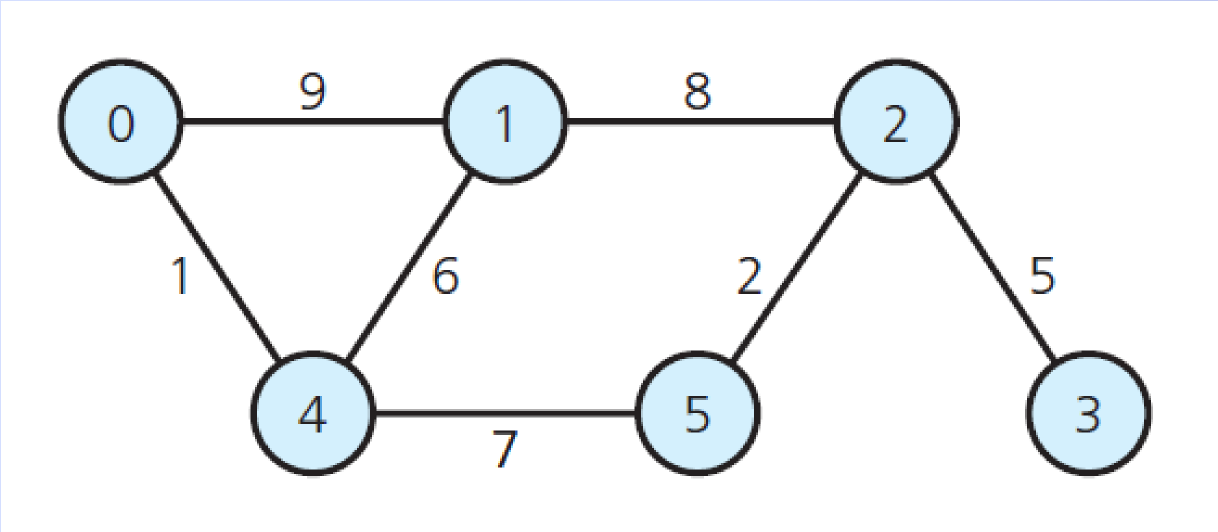


FIGURE 20-33 A graph for Checkpoint Questions 6 and 7 and for Exercises 1 and 4