Chapter 13: Graphs

Data Abstraction & Problem Solving with C++

Fifth Edition

by Frank M. Carrano

Terminology

- A graph *G* consists of two sets
  - A set *V* of vertices, or nodes
  - A set *E* of edges
- G = {V, E}
- A subgraph
  - Consists of a subset of a graph's vertices and a subset of its edges

Terminology

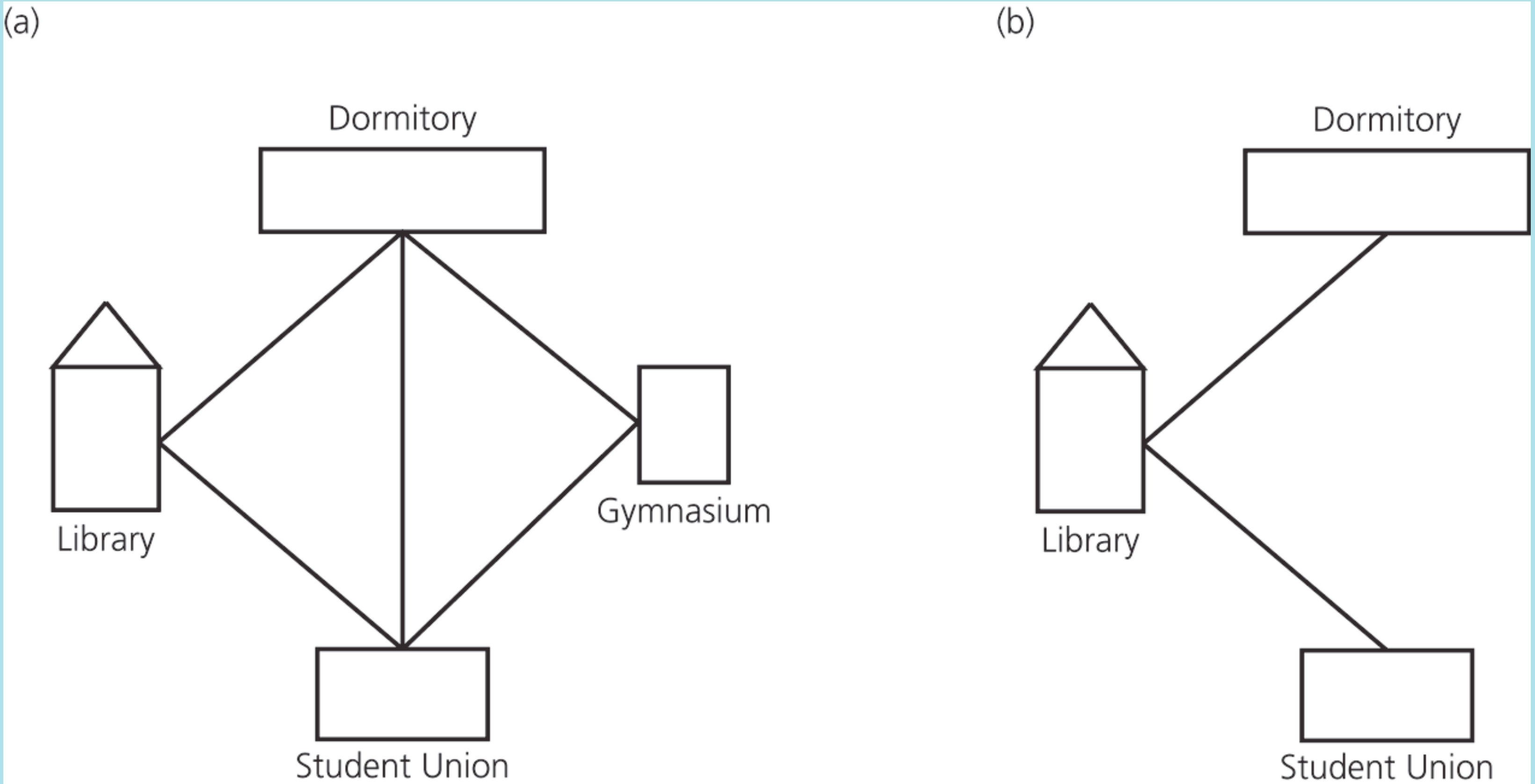


Figure 13-2 (a) A campus map as a graph; (b) a subgraph

Terminology

- Adjacent vertices
  - Two vertices that are joined by an edge
- A path between two vertices
  - A sequence of edges that begins at one vertex and ends at another vertex
  - May pass through the same vertex more than once
- A simple path
  - A path that passes through a vertex only once

Terminology

- A cycle
  - A path that begins and ends at the same vertex
- A simple cycle
  - A cycle that does not pass through a vertex more than once
- A connected graph
  - A graph that has a path between each pair of distinct vertices

Terminology

- A disconnected graph
  - A graph that has at least one pair of vertices without a path between them
- A complete graph
  - A graph that has an edge between each pair of distinct vertices
- A multigraph
  - Not a graph
  - Allows multiple edges between vertices

Terminology

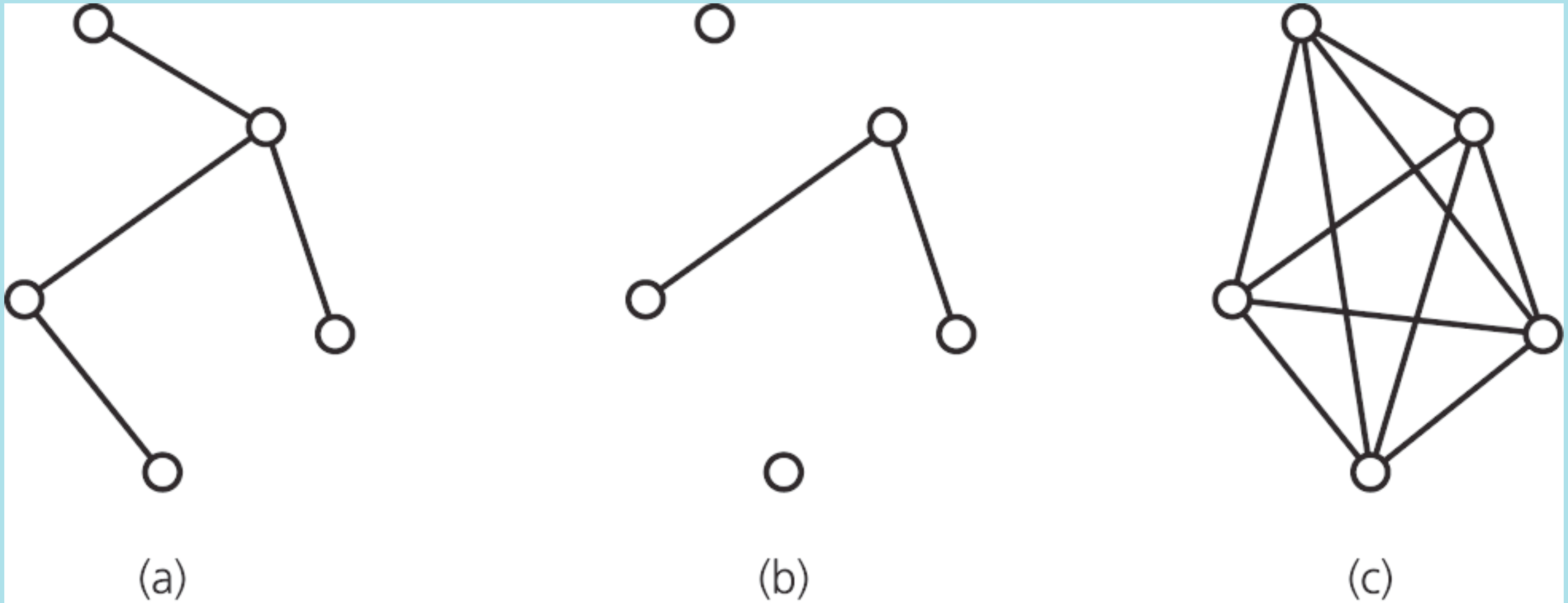


Figure 13-3 Graphs that are (a) connected; (b) disconnected; and (c) complete

Terminology

- Weighted graph
  - A graph whose edges have numeric labels
- Undirected graph
  - Edges do not indicate a direction

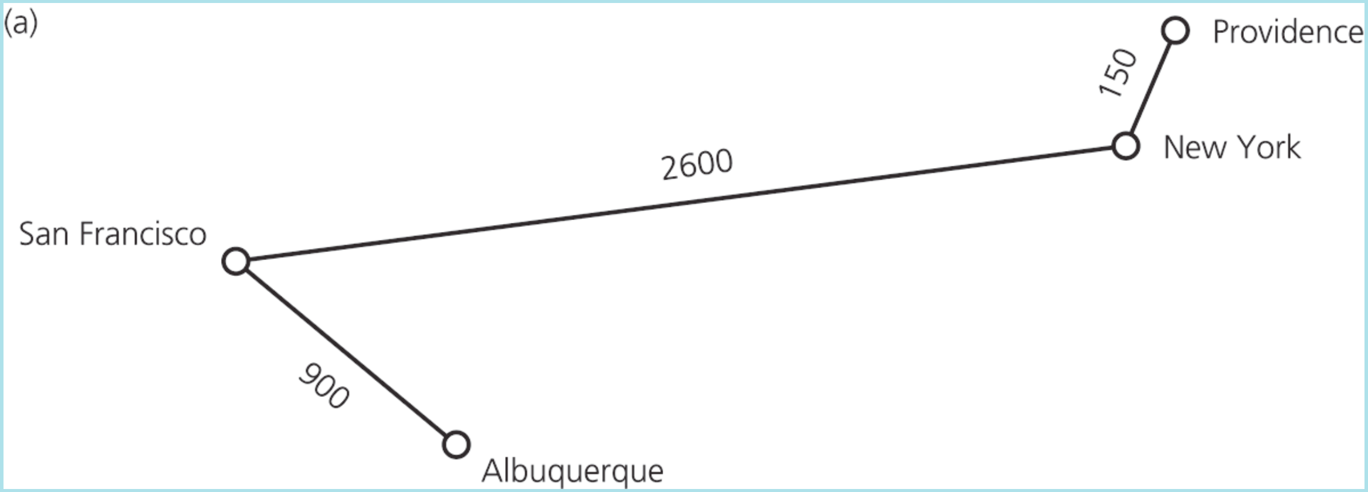


Figure 13-5a A weighted, undirected graph

Terminology

- Directed Graph
  - Each edge has a direction; directed edges
  - Can have two edges between a pair of vertices, one in each direction
  - Directed path is a sequence of directed edges between two vertices
  - Vertex *y* is adjacent to vertex *x* if there is a directed edge from *x* to *y*

Terminology

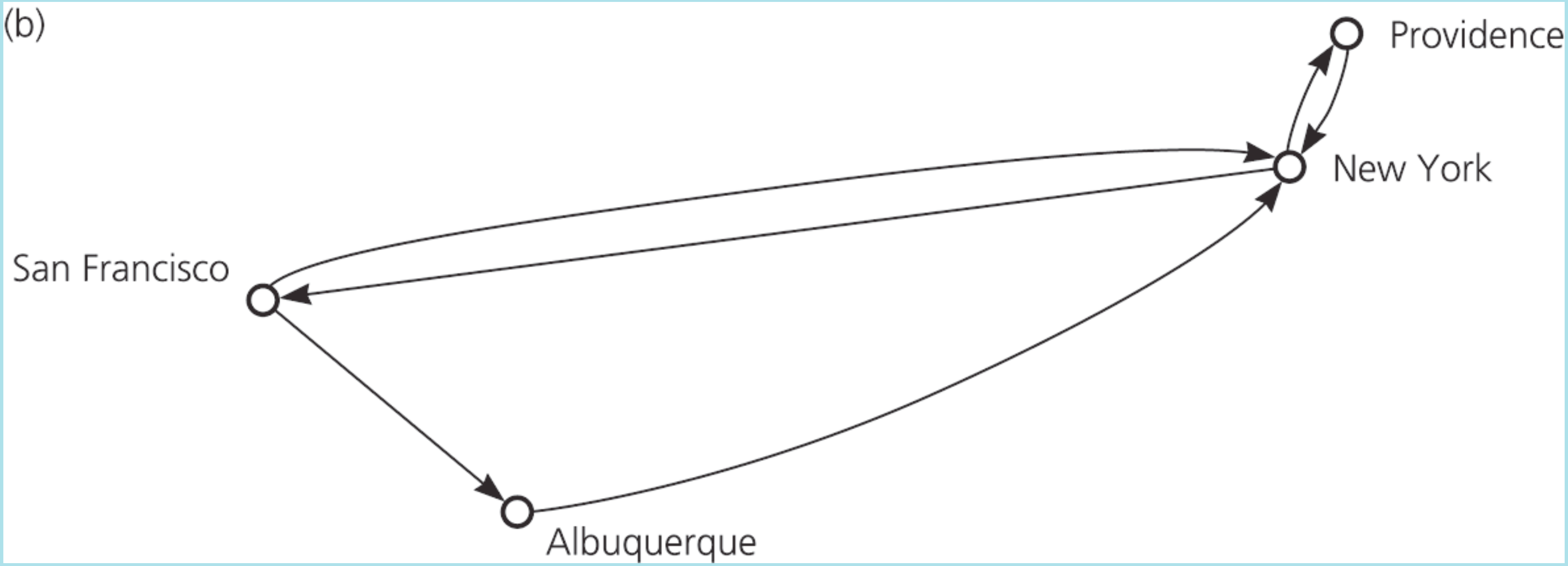


Figure 13-5b A directed, unweighted graph

Graphs As ADTs

- Variations of an ADT graph are possible
  - Vertices may or may not contain values
    - Many problems have no need for vertex values
    - Relationships among vertices is what is important
  - Either directed or undirected edges
  - Either weighted or unweighted edges
- Insertion and deletion operations for graphs apply to vertices and edges
- Graphs can have traversal operations

Implementing Graphs

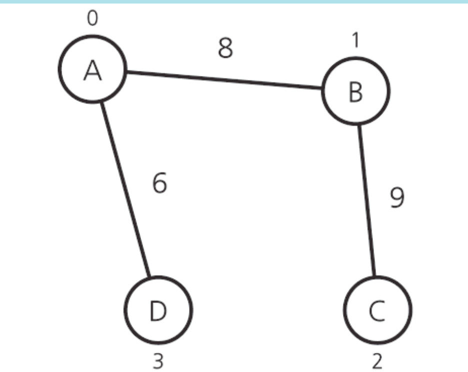
- Most common implementations of a graph
  - Adjacency matrix
  - Adjacency list
- Adjacency matrix for a graph that has n vertices numbered 0, 1, …, n – 1
  - An n by n array `matrix` such that `matrix[i][j]` indicates whether an edge exists from vertex i to vertex j

Implementing Graphs

– For an unweighted graph, `matrix[i][j]` is
  - 1 (or true) if an edge exists from vertex *i* to vertex *j*
  - 0 (or false) if no edge exists from vertex *i* to vertex *j*

– For a weighted graph, `matrix[i][j]` is
  - The weight of the edge from vertex *i* to vertex *j*
  - ∞ if no edge exists from vertex *i* to vertex *j*

(a)



(b)

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

Figure 13-7 (a) A weighted undirected graph and (b) its adjacency matrix

Implementing Graphs

- Adjacency list for a directed graph that has *n* vertices numbered 0, 1, …, $n - 1$
  - An array of *n* linked lists
  - The i^{th} linked list has a node for vertex *j* if and only if an edge exists from vertex *i* to vertex *j*
  - The list's node can contain either
    - Vertex *j*'s value, if any
    - An indication of vertex *j*'s identity

Implementing Graphs

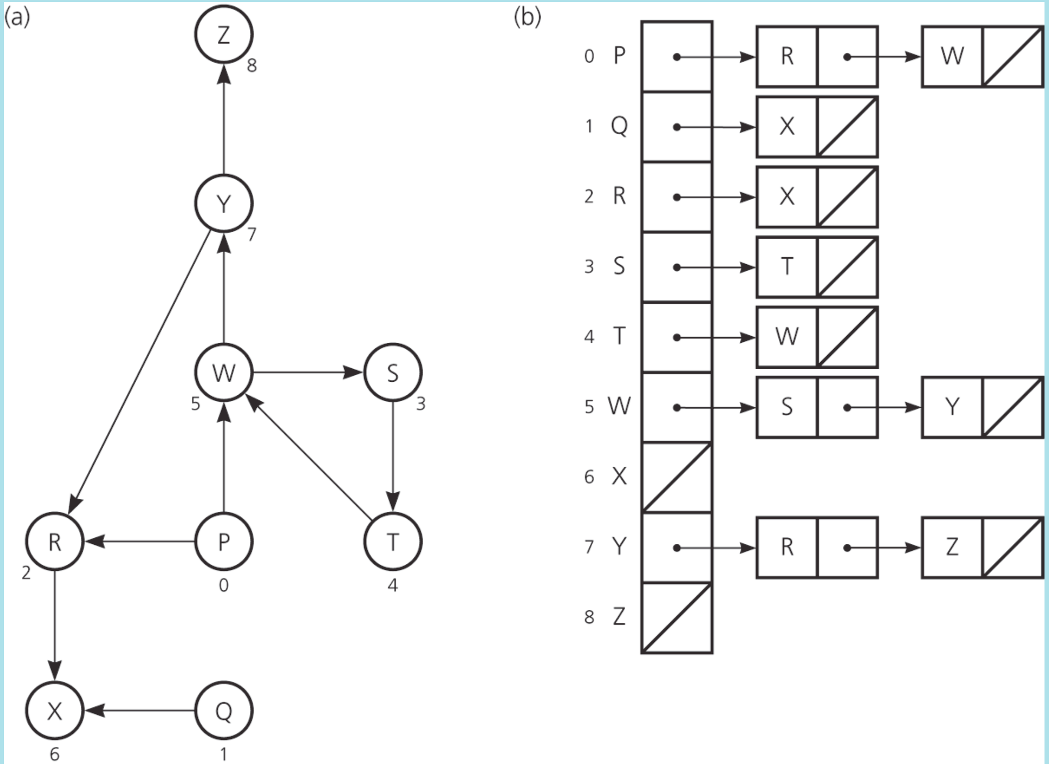


Figure 13-8

(a) A directed graph

(b) its adjacency list

– For an undirected graph, treat each edge as if it were two directed edges in opposite directions

Implementing Graphs

- Two common operations on graphs
  1. Determine whether there is an edge from vertex *i* to vertex *j*
  2. Find all vertices adjacent to a given vertex *i*
- Adjacency matrix
  - Supports operation 1 more efficiently
- Adjacency list
  - Supports operation 2 more efficiently
  - Often requires less space than an adjacency matrix

Implementing a Graph class Using the STL

- An adjacency list representation of a graph can be implemented using a *vector* of *maps*
- For a weighted graph
  - The vector elements represent the vertices of a graph
  - The map for each vertex contains element pairs
    - Each pair consists of an adjacent vertex and an edge weight

Graph Traversals

- Visits all the vertices that it can reach
- Visits all vertices of the graph if and only if the graph is connected
  - A connected component
    - The subset of vertices visited during a traversal that begins at a given vertex
- To prevent indefinite loops
    - Mark each vertex during a visit, and
    - Never visit a vertex more than once

DFS and BFS Traversals

- Depth-First Search (DFS) Traversal
  - Proceeds along a path from a vertex *v* as deeply into the graph as possible before backing up
  - A “last visited, first explored” strategy
  - Has a simple recursive form
  - Has an iterative form that uses a stack

DFS and BFS Traversals

- Breadth-First Search (BFS) Traversal
  - Visits every vertex adjacent to a vertex *v* that it can before visiting any other vertex
  - A “first visited, first explored” strategy
  - An iterative form uses a queue
  - A recursive form is possible, but not simple

DFS and BFS Traversals

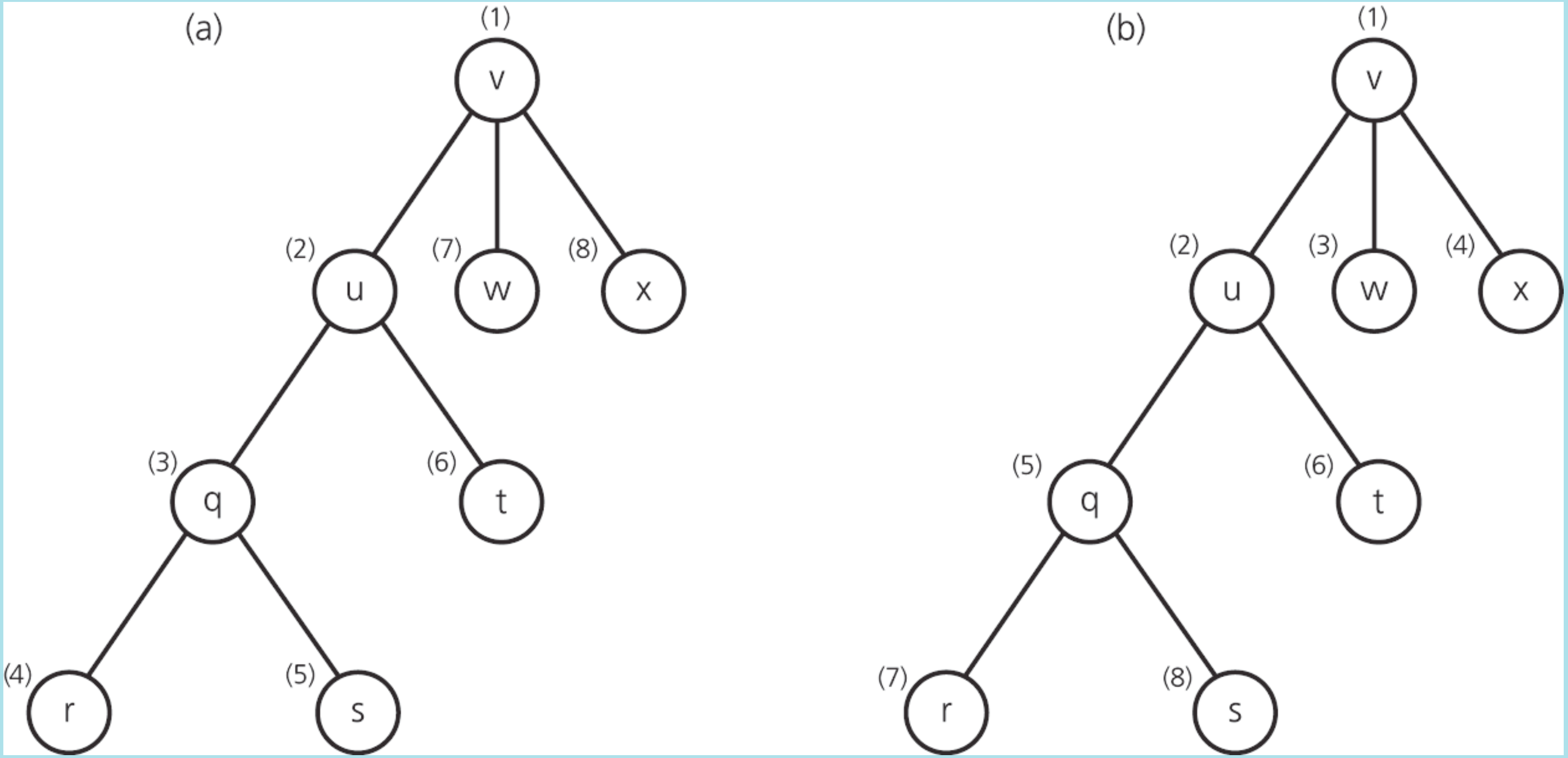


Figure 13-10 Visitation order for (a) a depth-first search; (b) a breadth-first search

Implementing a BFS Class Using the STL

- A class providing a BFS traversal can be implemented using the STL `vector` and `queue` containers
  - Two vectors of integers
    - `mark` stores vertices that have been visited
    - `parents` stores the parent of each vertex for use by other graph algorithms.
  - A queue of `Edges`
    - BFS processes the edges from each vertex's adjacency list in the order that they were pushed onto the queue

Applications of Graphs: Topological Sorting

- Topological order
  - A list of vertices in a directed graph without cycles such that vertex *x* precedes vertex *y* if there is a directed edge from *x* to *y* in the graph
  - Several topological orders are possible for a given graph
- Topological sorting
  - Arranging the vertices into a topological order

Topological Sorting

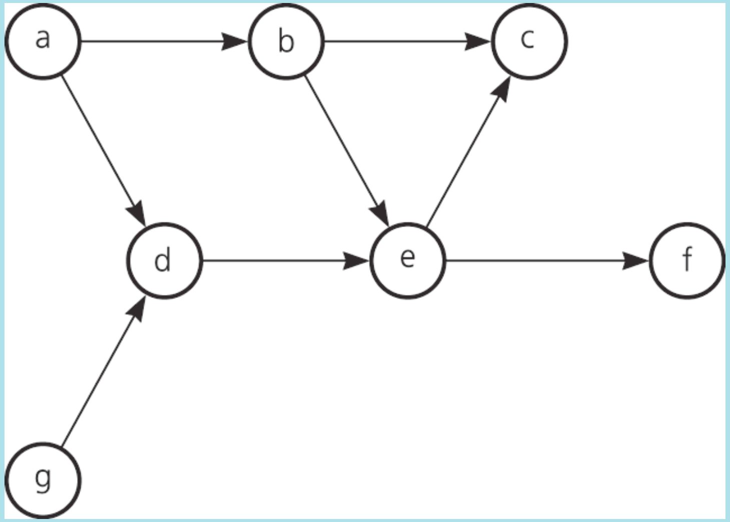


Figure 13-14 A directed graph without cycles

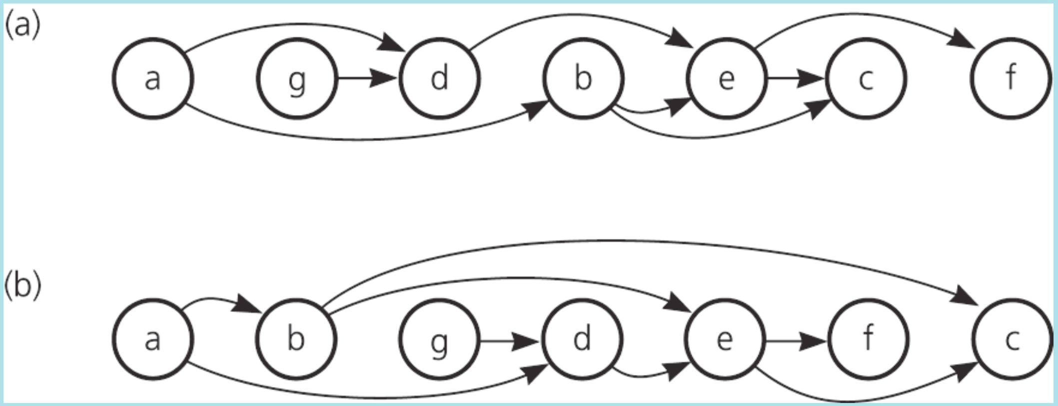


Figure 13-15 The graph in Figure 13-14 arranged according to the topological orders (a) *a, g, d, b, e, c, f* and (b) *a, b, g, d, e, f, c*

Topological Sorting Algorithms

- `topSort1`
  1. Find a vertex that has no successor
  2. Add the vertex to the beginning of a list
  3. Remove that vertex from the graph, as well as all edges that lead to it
  4. Repeat the previous steps until the graph is empty
     - When the loop ends, the list of vertices will be in topological order

Topological Sorting Algorithms

- `topSort2`
  - A modification of the iterative DFS algorithm
  - Push all vertices that have no predecessor onto a stack
  - Each time you pop a vertex from the stack, add it to the beginning of a list of vertices
  - When the traversal ends, the list of vertices will be in topological order

A Trace of `topSort2`

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)	

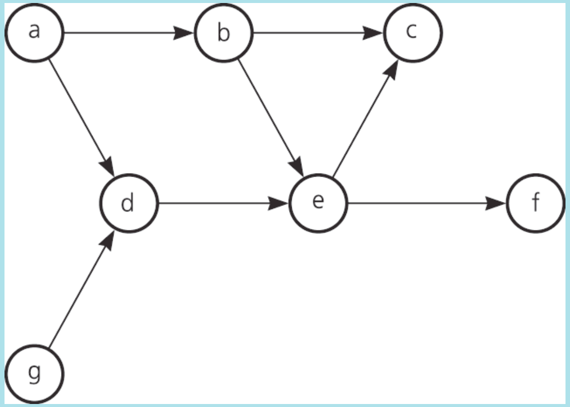


Figure 13-14

Figure 13-17 A trace of `topSort2` for the graph in Figure 13-14

Spanning Trees

- A tree is an undirected connected graph without cycles
- A spanning tree of a connected undirected graph G is
  - A subgraph of G that contains all of G's vertices and enough of its edges to form a tree

Spanning Trees

- To obtain a spanning tree from a connected undirected graph with cycles
  - Remove edges until there are no cycles

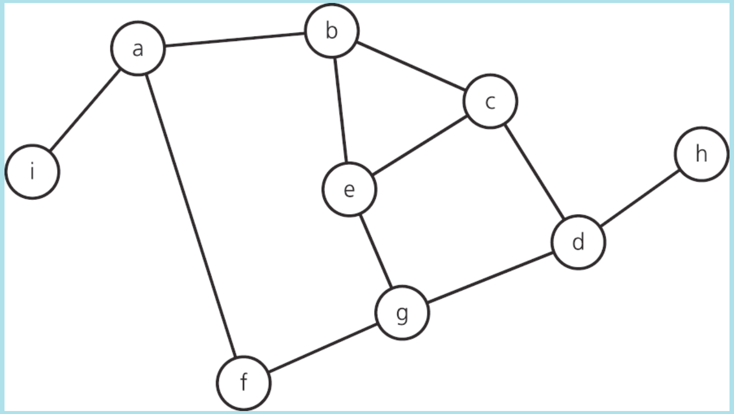


Figure 13-11 A connected graph with cycles

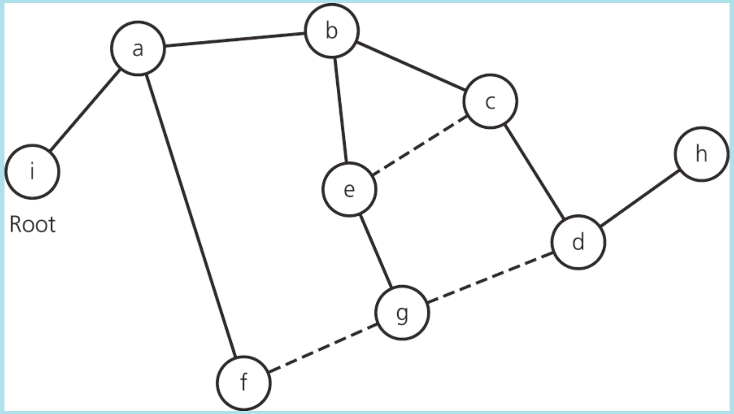


Figure 13-18 A spanning tree for the graph

Spanning Trees

- Detecting a cycle in an undirected connected graph
  - A connected undirected graph that has n vertices must have at least $n - 1$ edges
  - A connected undirected graph that has n vertices and exactly $n - 1$ edges cannot contain a cycle
  - A connected undirected graph that has n vertices and more than $n - 1$ edges must contain at least one cycle

Spanning Trees

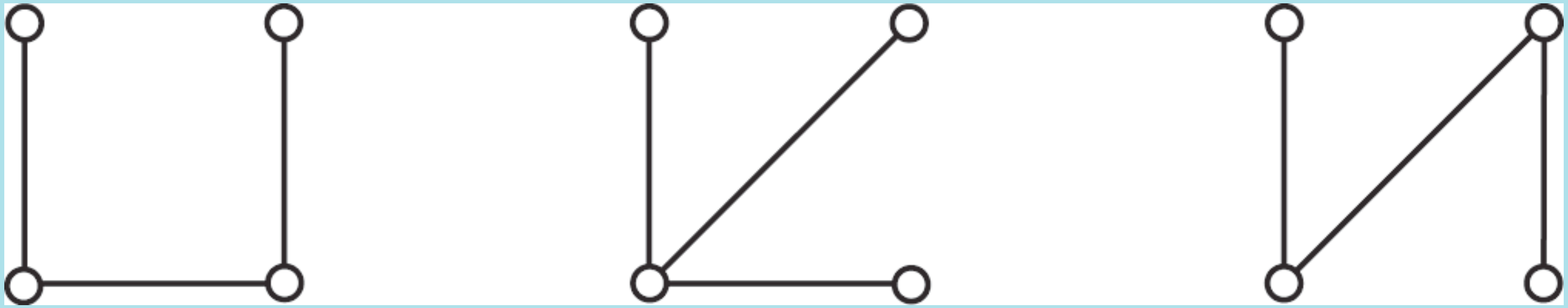


Figure 13-19 Connected graphs that each have four vertices and three edges

The DFS Spanning Tree

- To create a depth-first search (DFS) spanning tree
  - Traverse the graph using a depth-first search and mark the edges that you follow
  - After the traversal is complete, the graph's vertices and marked edges form the spanning tree

The BFS Spanning Tree

- To create a breath-first search (BFS) spanning tree
  - Traverse the graph using a bread-first search and mark the edges that you follow
  - When the traversal is complete, the graph's vertices and marked edges form the spanning tree

Minimum Spanning Trees

- Cost of the spanning tree
  - Sum of the costs of the edges of the spanning tree
- A minimum spanning tree of a connected undirected graph has a minimal edge-weight sum
  - A particular graph could have several minimum spanning trees

Minimum Spanning Trees: Prim's Algorithm

- Find a minimum spanning tree that begins at any given vertex
  1. Find the least-cost edge (v, u) from a visited vertex v to some unvisited vertex u
  2. Mark u as visited
  3. Add the vertex u and the edge (v, u) to the minimum spanning tree
  4. Repeat the above steps until all vertices are visited

Shortest Paths

- Shortest path between two vertices in a weighted graph is
  - The path that has the smallest sum of its edge weights

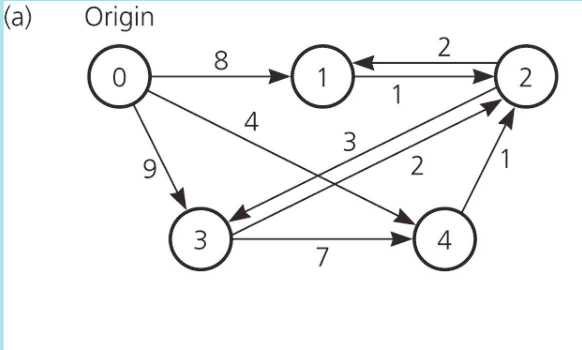
Shortest Paths: Dijkstra's Algorithm

- Find the shortest paths between a given origin and all other vertices
- Dijkstra's algorithm uses
  - A set *vertexSet* of selected vertices
  - An array *weight*, where *weight*[*v*] is the weight of the shortest (cheapest) path from vertex 0 to vertex *v* that passes through vertices in *vertexSet*

Shortest Paths: Dijkstra's Algorithm

Figure 13-24a

A weighted directed graph



			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 13-25 A trace of the shortest-path algorithm applied to the graph

Circuits

- A circuit
  - A special cycle that passes through every vertex (or edge) in a graph exactly once
- Euler circuit
  - A circuit that begins at a vertex *v*, passes through every edge exactly once, and terminates at *v*
  - Exists if and only if each vertex touches an even number of edges

Circuits

- A Hamilton circuit
  - Begins at a vertex *v*, passes through every vertex exactly once, and terminates at *v*
  - The traveling salesman problem
- A planar graph
  - Can be drawn so that no two edges cross
  - The three utilities problem
  - The four-color problem

Summary

- The most common implementations of a graph use either an adjacency matrix or an adjacency list
- Graph searching
  - Depth-first search goes as deep into the graph as it can before backtracking
    - Uses a stack
  - Bread-first search visits all possible adjacent vertices before traversing further into the graph
    - Uses a queue

Summary

- Topological sorting produces a linear order of the vertices in a directed graph without cycles
- Trees are connected undirected graphs without cycles
- A spanning tree of a connected undirected graph is
  - A subgraph that contains all the graph's vertices and enough of its edges to form a tree

Summary

- A minimum spanning tree for a weighted undirected graph is
  - A spanning tree whose edge-weight sum is minimal
- The shortest path between two vertices in a weighted directed graph is
  - The path that has the smallest sum of its edge weights

Summary

- An Euler circuit in an undirected graph is
  - A cycle that begins at vertex *v*, passes through every edge in the graph exactly once, and terminates at *v*
- A Hamilton circuit in an undirected graph is
  - A cycle that begins at vertex *v*, passes through every vertex in the graph exactly once, and terminates at *v*