Graphs

Chapter 9

Chapter Objectives

- To become familiar with graph terminology and the different types of graphs
- To study a Graph ADT and different implementations of the Graph ADT
- To learn the breadth-first and depth-first search traversal algorithms
- To learn some algorithms involving weighted graphs
- To study some applications of graphs and graph algorithms

Graph Terminology

- A graph is a data structure that consists of a set of vertices and a set of edges between pairs of vertices
- Edges represent paths or connections between the vertices
- The set of vertices and the set of edges must both be finite and neither one be empty

Visual Representation of Graphs

- Vertices are represented as points or labeled circles and edges are represented as lines joining the vertices
- The physical layout of the vertices and their labeling are not relevant

FIGURE 12.1
Graph Given in Example 12.1

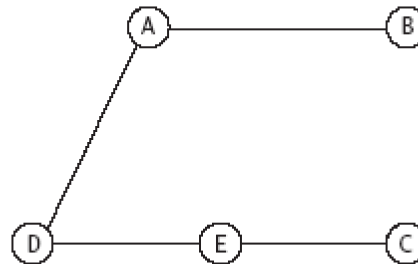
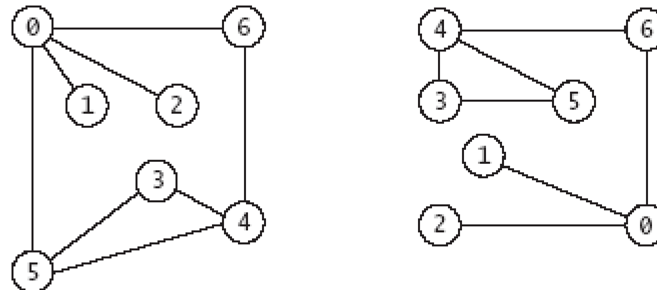


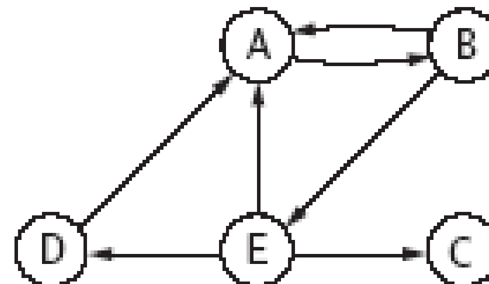
FIGURE 12.2
Two Representations of the Same Graph



Directed and Undirected Graphs

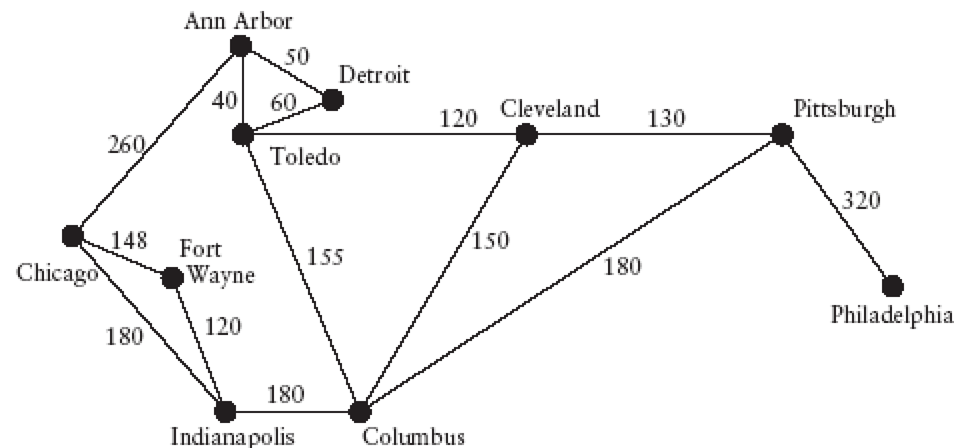
- The edges of a graph are directed if the existence of an edge from A to B does not necessarily guarantee that there is a path in both directions
- A graph with directed edges is called a directed graph
- A graph with undirected edges is an undirected graph or simply a graph

FIGURE 12.3
Example of a Directed Graph



Directed and Undirected Graphs (continued)

- The edges in a graph may have values associated with them known as their weights
- A graph with weighted edges is known as a weighted graph



Paths and Cycles

- A vertex is adjacent to another vertex if there is an edge to it from that other vertex
- A path is a sequence of vertices in which each successive vertex is adjacent to its predecessor
- In a simple path, the vertices and edges are distinct except that the first and last vertex may be the same
- A cycle is a simple path in which only the first and final vertices are the same
- If a graph is not connected, it is considered unconnected, but will still consist of connected endpoints

Paths and Cycles (continued)

FIGURE 12.6
Not a Simple Path

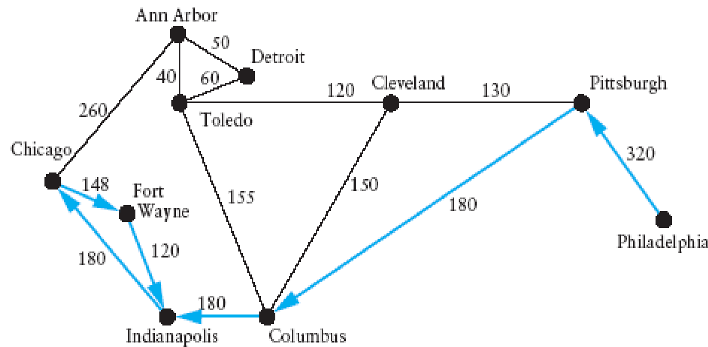


FIGURE 12.7
A Cycle

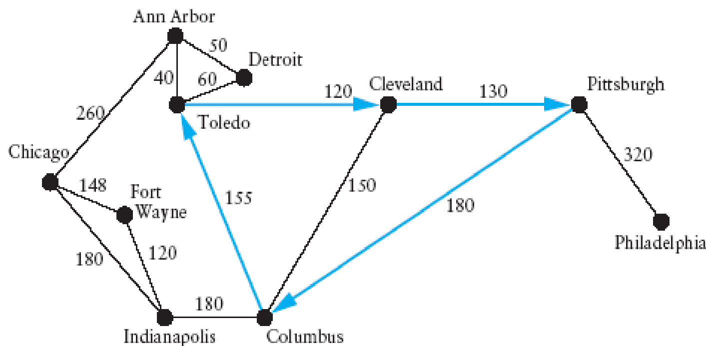
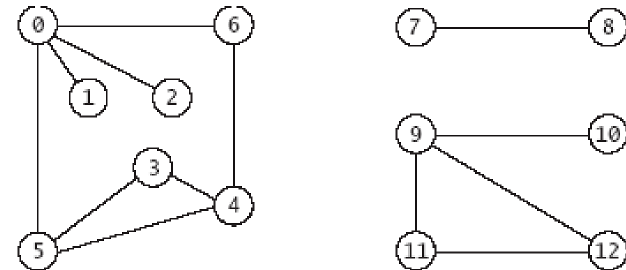


FIGURE 12.8
Example of an
Unconnected Graph



The Graph ADT and Edge Class

- Java does not provide a Graph ADT
- In making our own, we need to be able to do the following
    - Create a graph with the specified number of vertices
    - Iterate through all of the vertices in the graph
    - Iterate through the vertices that are adjacent to a specified vertex
    - Determine whether an edge exists between two vertices
    - Determine the weight of an edge between two vertices
    - Insert an edge into the graph

The Graph ADT and Edge Class (continued)

TABLE 12.1
The Edge Class

Data Field	Attribute
`private int dest`	The destination vertex for an edge.
`private int source`	The source vertex for an edge.
`private double weight`	The weight.
Constructor	**Purpose**
`public Edge(int source, int dest)`	Constructs an Edge from `source` to `dest`. Sets the `weight` to 1.0.
`public Edge(int source, int dest, double w)`	Constructs an Edge from `source` to `dest`. Sets the `weight` to `w`.
Method	**Behavior**
`public boolean equals(Object o)`	Compares two edges for equality. Edges are equal if their source and destination vertices are the same. The weight is not considered.
`public int getDest()`	Returns the destination vertex.
`public int getSource()`	Returns the source vertex.
`public double getWeight()`	Returns the weight.
`public int hashCode()`	Returns the hash code for an edge. The hash code depends only on the source and destination.
`public String toString()`	Returns a string representation of the edge.

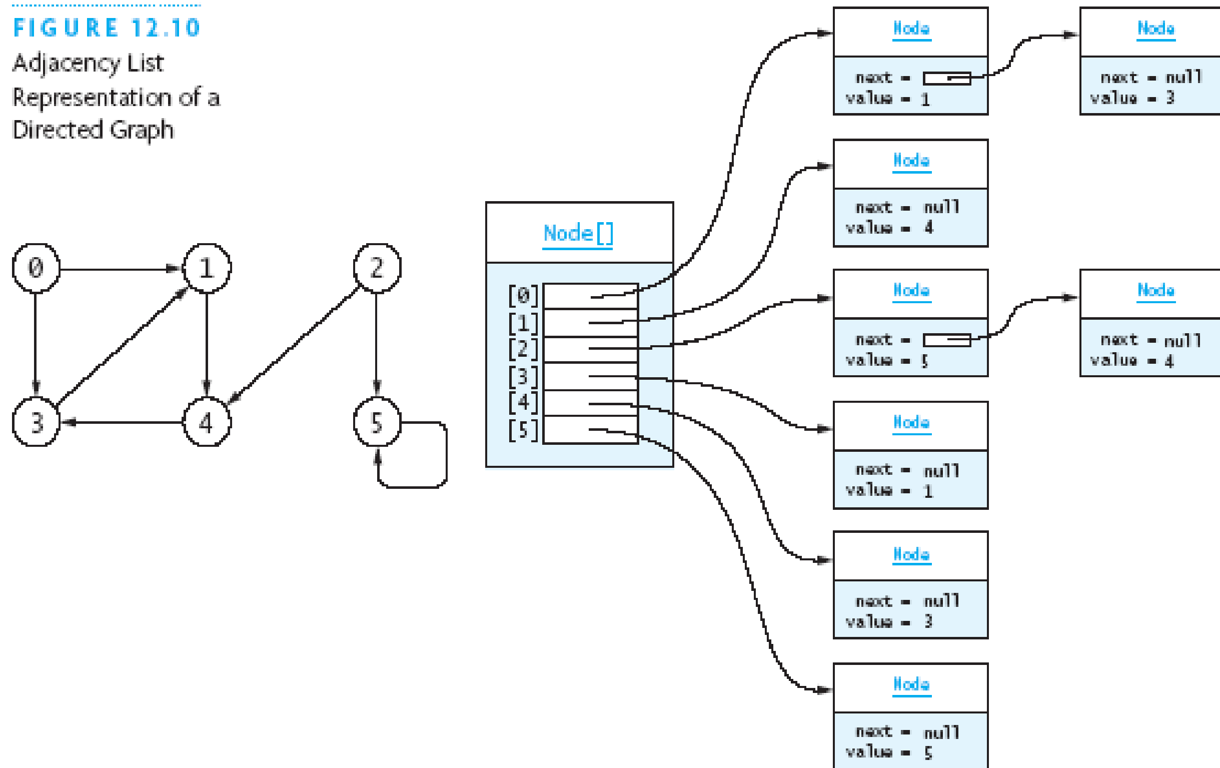
Implementing the Graph ADT

- Because graph algorithms have been studied and implemented throughout the history of computer science, many of the original publications of graph algorithms and their implementations did not use an object-oriented approach and did not even use abstract data types
- Two representations of graphs are most common
  - Edges are represented by an array of lists called adjacency lists, where each list stores the vertices adjacent to a particular vertex
- Edges are represented by a two dimensional array, called an adjacency matrix

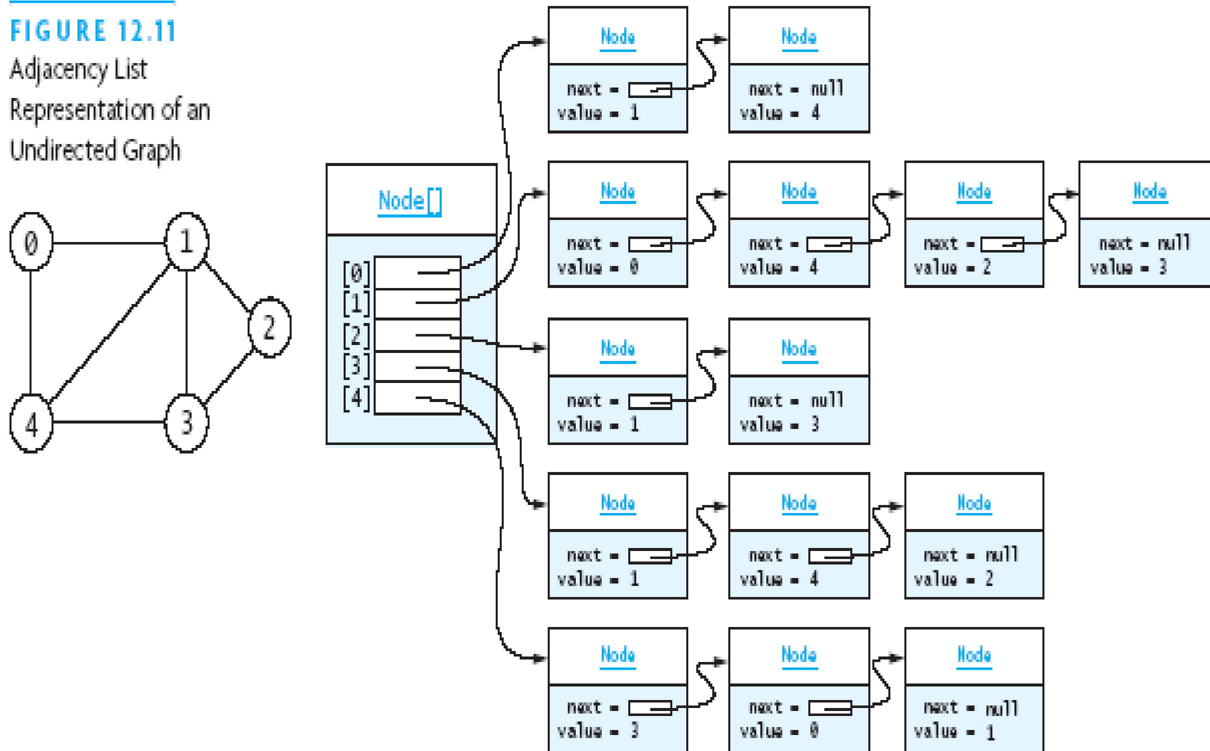
Adjacency List

- An adjacency list representation of a graph uses an array of lists
- One list for each vertex

Adjacency List (continued)



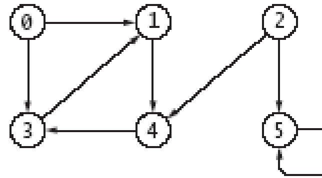
Adjacency List (continued)



Adjacency Matrix

- Uses a two-dimensional array to represent a graph
- For an unweighted graph, the entries can be Boolean values
- For a weighted graph, the matrix would contain the weights

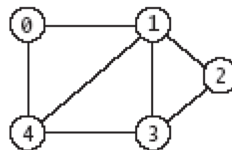
FIGURE 12.12
A Directed Graph and the Corresponding Adjacency Matrix



Column

Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		1.0		
[1]					1.0	
[2]					1.0	1.0
[3]		1.0				
[4]				1.0		
[5]						1.0

FIGURE 12.13
Undirected Graph and Adjacency Matrix Representation

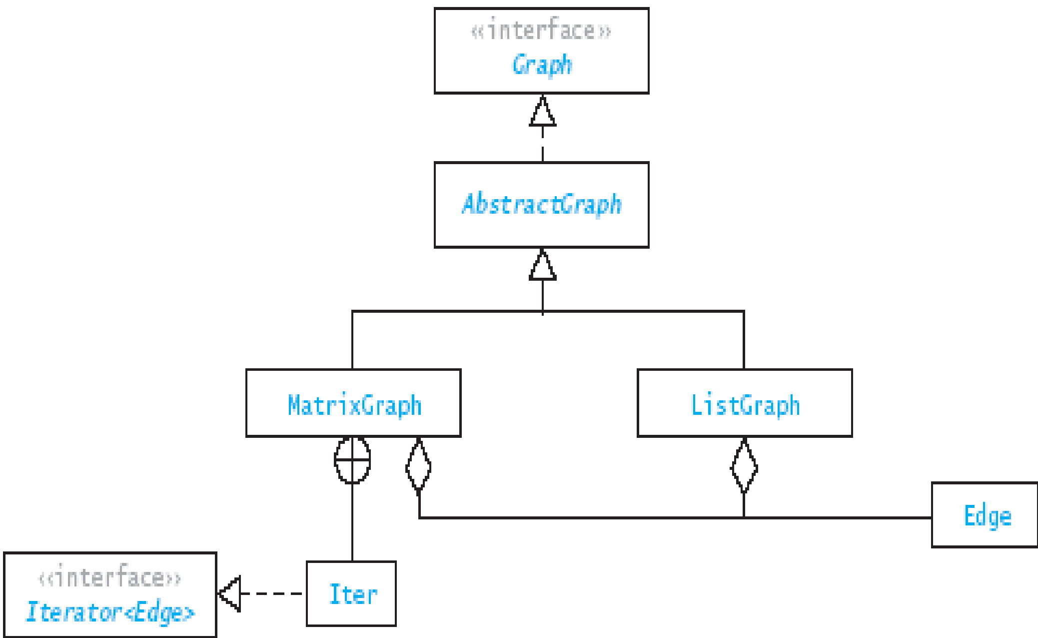


Column

Row	[0]	[1]	[2]	[3]	[4]
[0]		1.0			1.0
[1]	1.0		1.0	1.0	1.0
[2]		1.0		1.0	
[3]		1.0	1.0		1.0
[4]	1.0	1.0		1.0	

Overview of the Graph Class Hierarchy

FIGURE 12.14
UML Class Diagram of Graph Class Hierarchy



Class AbstractGraph

TABLE 12.2
The Abstract Class AbstractGraph

Data Field	Attribute
`private boolean directed`	**true** if this is a directed graph.
`private int numV`	The number of vertices.
Constructor	**Purpose**
`public AbstractGraph(int numV, boolean directed)`	Constructs an empty graph with the specified number of vertices and with the specified `directed` flag. If `directed` is **true**, this is a directed graph.
Method	**Behavior**
`public int getNumV()`	Gets the number of vertices.
`public boolean isDirected()`	Returns **true** if the graph is a directed graph.
`public void loadEdgesFromFile(BufferedReader bR)`	Loads edges from a data file.
`public static Graph createGraph (BufferedReader bR, boolean isDirected, String type)`	Factory method to create a graph and load the data from an input file.

The ListGraph Class

TABLE 12.3
The ListGraph Class

Data Field	Attribute
private List<Edge>[] edges	An array of Lists to contain the edges that originate with each vertex.
Constructor	**Purpose**
public ListGraph(int numV, boolean directed)	Constructs a graph with the specified number of vertices and directionality.
Method	**Behavior**
public Iterator<Edge> edgeIterator(int source)	Returns an iterator to the edges that originate from a given vertex.
public Edge getEdge(int source, int dest)	Gets the edge between two vertices.
public void insert(Edge e)	Inserts a new edge into the graph.
public boolean isEdge(int source, int dest)	Determines whether an edge exists from vertex source to dest.

Traversals of Graphs

- Most graph algorithms involve visiting each vertex in a systematic order
- Most common traversal algorithms are the breadth first and depth first search

Breadth-First Search

- In a breadth-first search, we visit the start first, then all nodes that are adjacent to it next, then all nodes that can be reached by a path from the start node containing two edges, three edges, and so on
- Must visit all nodes for which the shortest path from the start node is length k before we visit any node for which the shortest path from the start node is length k+1
- There is no special start vertex

Example of a Breadth-First Search

FIGURE 12.15
Graph to Be Traversed Breadth First

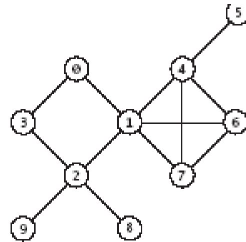
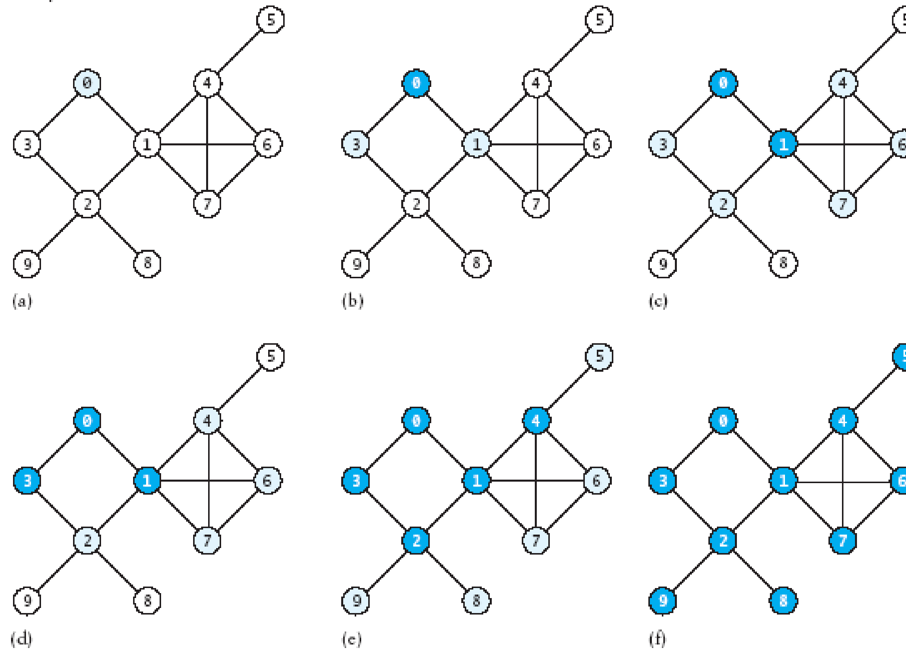


FIGURE 12.16
Example of a Breadth-First Search



Algorithm for Breadth-First Search

Algorithm for Breadth-First Search

1. Take an arbitrary start vertex, mark it identified (color it light blue), and place it in a queue.
2. `while` the queue is not empty
3.   Take a vertex, *u*, out of the queue and visit *u*.
4.   `for` all vertices, *v*, adjacent to this vertex, *u*
5.   `if` *v* has not been identified or visited
6.   Mark it identified (color it light blue).
7.   Insert vertex *v* into the queue.
8.   We are now finished visiting *u* (color it dark blue).

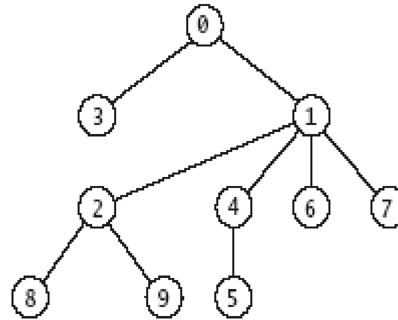
Algorithm for Breadth-First Search (continued)

TABLE 12.4
Trace of Breadth-First Search of Graph in Figure 12.15

Vertex Being Visited	Queue Contents After Visit	Visit Sequence
0	1 3	0
1	3 2 4 6 7	0 1
3	2 4 6 7	0 1 3
2	4 6 7 8 9	0 1 3 2
4	6 7 8 9 5	0 1 3 2 4
6	7 8 9 5	0 1 3 2 4 6
7	8 9 5	0 1 3 2 4 6 7
8	9 5	0 1 3 2 4 6 7 8
9	5	0 1 3 2 4 6 7 8 9
5	empty	0 1 3 2 4 6 7 8 9 5

Algorithm for Breadth-First Search (continued)

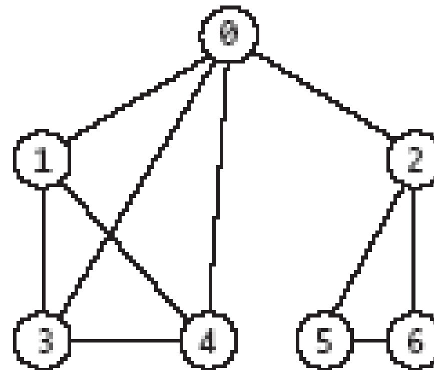
FIGURE 12.17
Breadth First Search Tree of Graph in Figure 12.15



Depth-First Search

- In depth-first search, you start at a vertex, visit it, and choose one adjacent vertex to visit; then, choose a vertex adjacent to that vertex to visit, and so on until you go no further; then back up and see whether a new vertex can be found

FIGURE 12.18
Graph to Be Traversed Depth First



Depth-First Search (continued)

FIGURE 12.19
Example of Depth-First Search

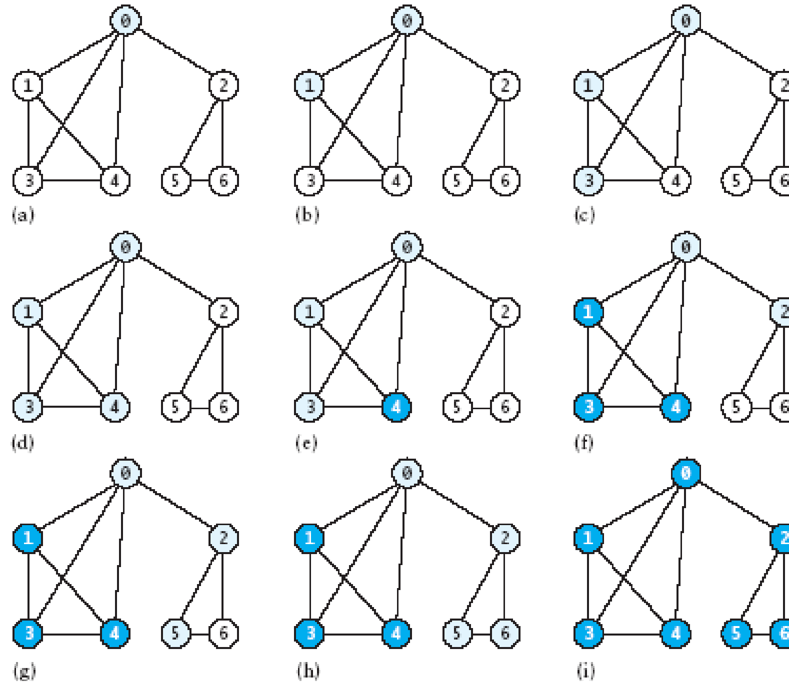
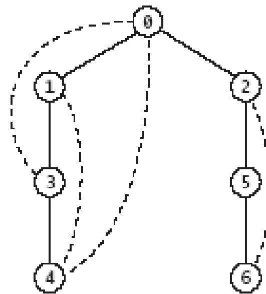


FIGURE 12.20
Depth-First Search Tree of Figure 12.18



Depth-First Search (continued)

Algorithm for Depth-First Search

1. Mark the current vertex, *u*, visited (color it light blue), and enter it in the discovery order list
2. for each vertex, *v*, adjacent to the current vertex, *u*
3. if *v* has not been visited
4. Set parent of *v* to *u*.
5. Recursively apply this algorithm starting at *v*.
6. Mark *u* finished (color it dark blue) and enter *u* into the finish order list.

TABLE 12.5
Trace of Depth-First Search of Figure 12.19

Operation	Adjacent Vertices	Discovery (Visit) Order	Finish Order
Visit 0	1, 2, 3, 4	0	
Visit 1	0, 3, 4	0, 1	
Visit 3	0, 1, 4	0, 1, 3	
Visit 4	0, 1, 3	0, 1, 3, 4	
Finish 4			4
Finish 3			4, 3
Finish 1			4, 3, 1
Visit 2	0, 5, 6	0, 1, 3, 4, 2	
Visit 5	2, 6	0, 1, 3, 4, 2, 5	
Visit 6	2, 5	0, 1, 3, 4, 2, 5, 6	
Finish 6			4, 3, 1, 6
Finish 5			4, 3, 1, 6, 5
Finish 2			4, 3, 1, 6, 5, 2
Finish 0			4, 3, 1, 6, 5, 2, 0

Implementing Depth-First Search

TABLE 12.6
Class `DepthFirstSearch`

Data Field	Attribute
`private int discoverIndex`	The index that indicates the discovery order.
`private int[] discoveryOrder`	The array that contains the vertices in discovery order.
`private int finishIndex`	The index that indicates the finish order.
`private int[] finishOrder`	The array that contains the vertices in finish order.
`private Graph graph`	A reference to the graph being searched.
`private int[] parent`	The array of predecessors in the depth-first search tree.
`private boolean[] visited`	An array of **`boolean`** values to indicate whether or not a vertex has been visited.
Constructor	**Purpose**
`public DepthFirstSearch(Graph graph)`	Constructs the depth-first search of the specified graph selecting the start vertices in ascending vertex order.
`public DepthFirstSearch(Graph graph, int[] order)`	Constructs the depth-first search of the specified graph selecting the start vertices in the specified order. The first vertex visited is `order[0]`.
Method	**Behavior**
`public void depthFirstSearch(int s)`	Recursively searches the graph starting at vertex `s`.
`public int[] getDiscoveryOrder()`	Gets the discovery order.
`public int[] getFinishOrder()`	Gets the finish order.
`public int[] getParent()`	Gets the parents in the depth-first search tree.

Shortest Path Through a Maze

FIGURE 12.21

Recursive Solution to a Maze

Shortest Path Through a Maze (continued)

FIGURE 12.22
Graph Representation of the Maze in Figure 12.21

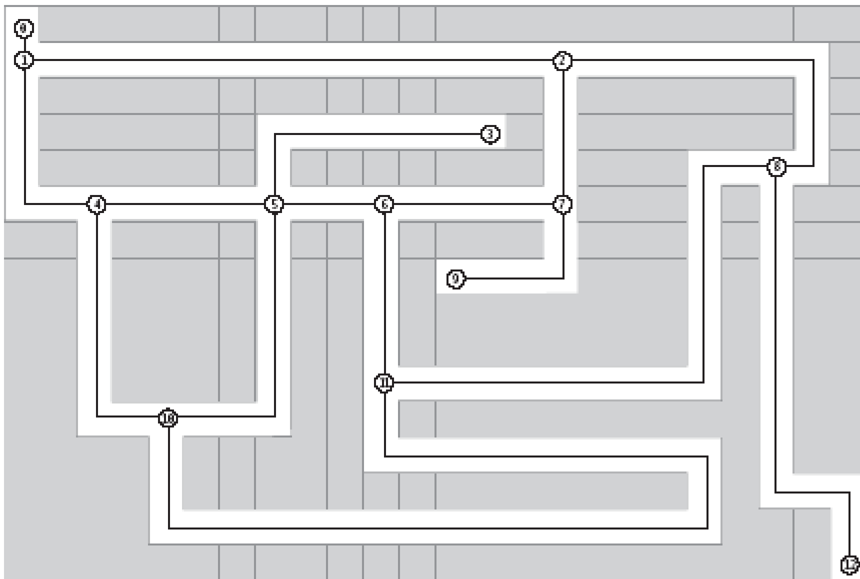
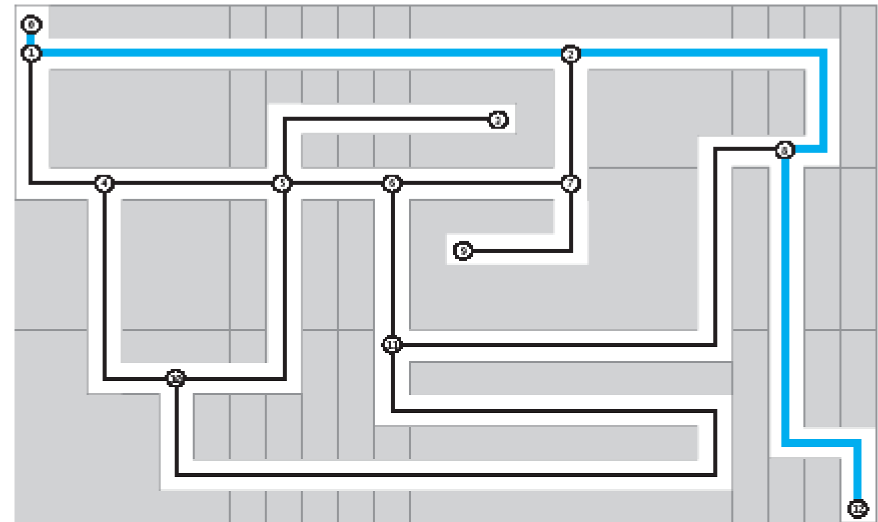


FIGURE 12.23
Solution to Maze in Figure 12.21



Topological Sort of a Graph

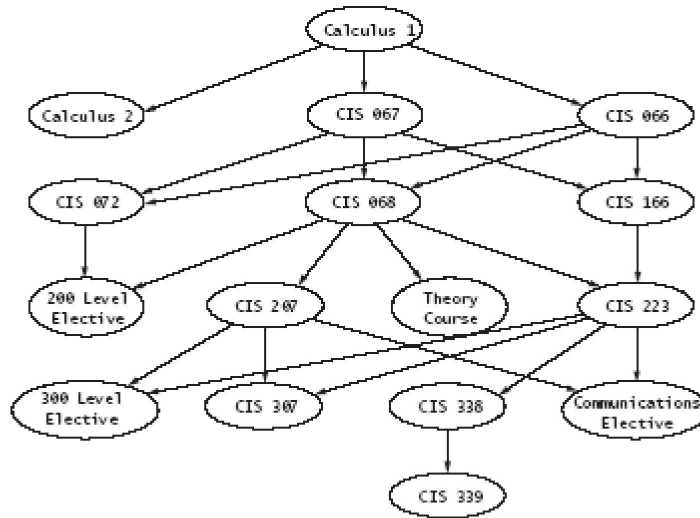


FIGURE 12.24
Prerequisites for a Computer Science Program

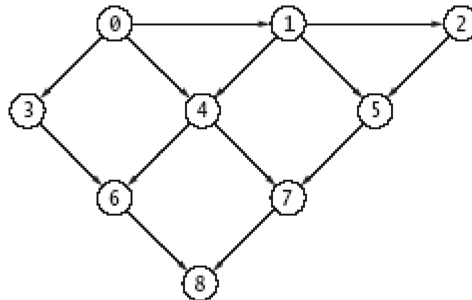


FIGURE 12.25
Example of a Directed Acyclic Graph

Algorithms Using Weighted Graphs

- Finding the shortest path from a vertex to all other vertices
  - Solution formulated by Dijkstra

Dijkstra's Algorithm

```
1.  Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.  for all v in V–S
3.          Set p[v] to s.
4.          if there is an edge (s, v)
5.                  Set d[v] to w(s, v).
            else
6.                  Set d[v] to ∞.
7.  while V–S is not empty
8.          for all u in V–S, find the smallest d[u].
9.          Remove u from V–S and add u to S.
10.         for all v adjacent to u in V–S
11.                 if d[u] + w(u, v) is less than d[v].
12.                         Set d[v] to d[u] + w(u, v).
13.                         Set p[v] to u.
```

Algorithms Using Weighted Graphs (continued)

- A minimum spanning tree is a subset of the edges of a graph such that there is only one edge between each vertex, and all of the vertices are connected
- The cost of a spanning tree is the sum of the weights of the edges
- We want to find the minimum spanning tree or the spanning tree with the smallest cost
- Solution formulated by R.C. Prim and is very similar to Dijkstra’s algorithm

Prim's Algorithm

Prim's Algorithm for Finding the Minimum Spanning Tree

```
1.   Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.   for all v in V–S
3.        Set p[v] to s.
4.        if there is an edge (s, v)
5.             Set d[v] to w(s, v).
          else
6.             Set d[v] to ∞.
7.   while V–S is not empty
8.        for all u in V–S, find the smallest d[u].
9.        Remove u from V–S and add it to S.
10.       Insert the edge (u, p[u]) into the spanning tree.
11.       for all v in V–S
12.            if w(u, v) < d[v]
13.                 Set d[v] to w(u, v).
14.                 Set p[v] to u.
```

Chapter Review

- A graph consists of a set of vertices and a set of edges
- In an undirected graph, if (u,v) is an edge, then there is a path from vertex u to vertex v, and vice versa
- In a directed graph, if (u,v) is an edge, then (v,u) is not necessarily an edge
- If there is an edge from one vertex to another, then the second vertex is adjacent to the first
- A graph is considered connected if there is a patch from each vertex to every other vertex

Chapter Review (continued)

- A tree is a special case of a graph
- Graphs may be represented by an array of adjacency lists
- Graphs may be represented by a two-dimensional square array called an adjacency matrix
- A breadth-first search of a graph finds all vertices reachable from a given vertex via the shortest path
- A depth-first search of a graph starts at a given vertex and then follows a path of unvisited vertices until it reaches a point where there are no unvisited vertices that are reachable

Chapter Review (continued)

- A topological sort determines an order for starting activities which are dependent on the completion of other activities
- Dijkstra's algorithm finds the shortest path from a start vertex to all other vertices
- Prim's algorithm finds the minimum spanning tree for a graph