Graphs

Chapter 12

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

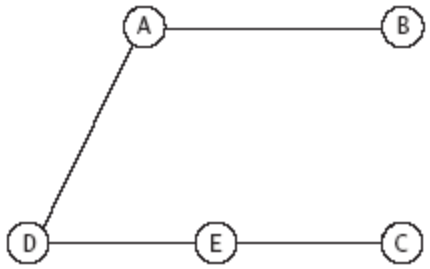


FIGURE 12.1
Graph Given in Example 12.1

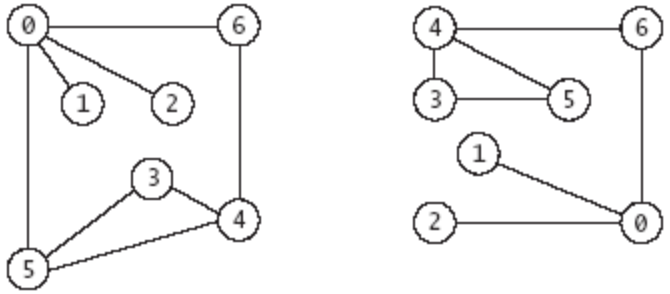
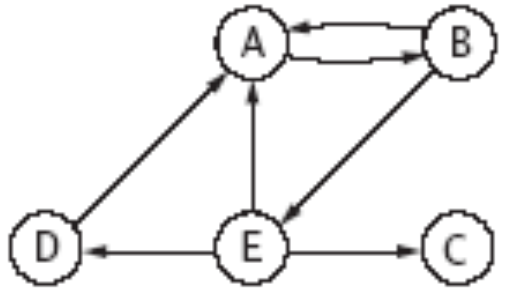


FIGURE 12.2
Two Representations of the Same Graph

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**

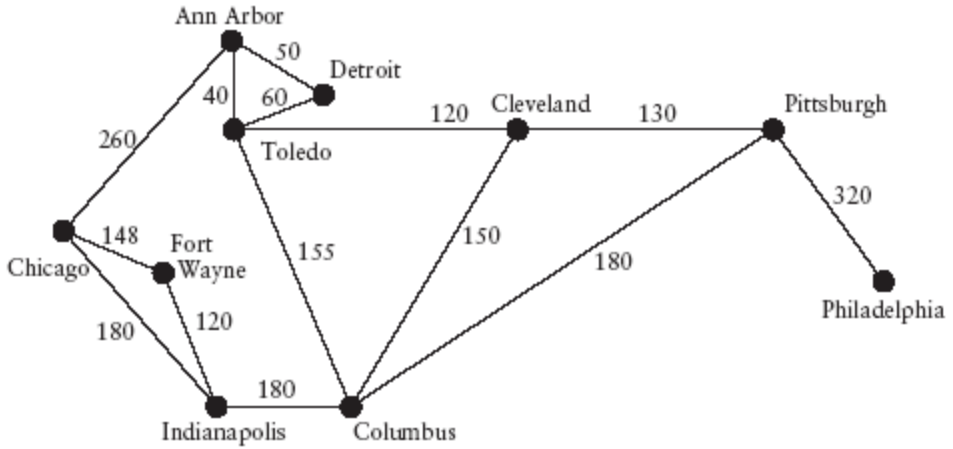
FIGURE 12.3
Example of a Directed Graph



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**

FIGURE 12.4
Example of 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

Paths and Cycles (cont'd)

- A graph is **connected** if there is a path from every node to every other node; otherwise, it is **unconnected**
- A **connected component** of a graph is a maximally large subset of vertices in which every node is connected to every other node

Paths and Cycles (continued)

FIGURE 12.6
Not a Simple Path

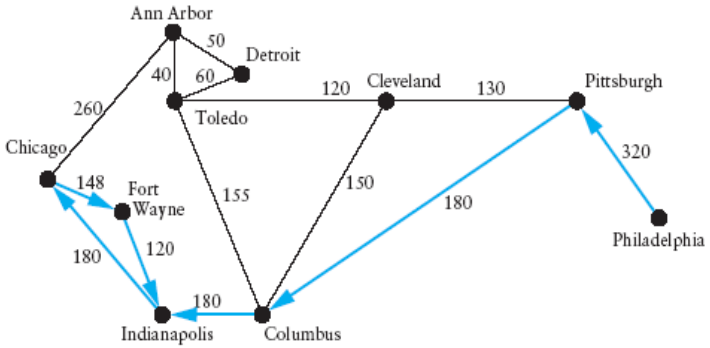


FIGURE 12.7
A Cycle

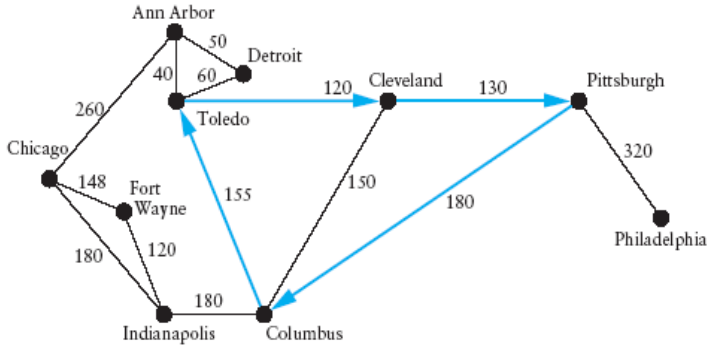
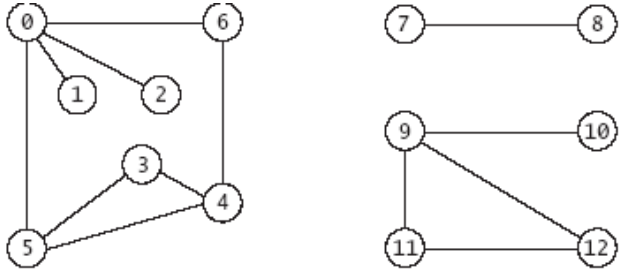


FIGURE 12.8
Example of an
Unconnected Graph



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

Edge Class

Data Field	Attribute
private int source	The index to the source vertex
private int dest	The index to the destination vertex
private double weight	The weight assigned to the edge
Constructor	**Purpose**
public Edge(int source, int dest)	Constructor with 0 weight
public Edge(int source, int dest, double weight)	Constructor with a specific weight value
Method	**Action**
public boolean equals(Object o)	Check equality
public int getSource() public int getDest() public double getWeight()	Getters
public String toString()	Conversion to a String object
public int hashCode()	Hash code

Implementing the Graph ADT

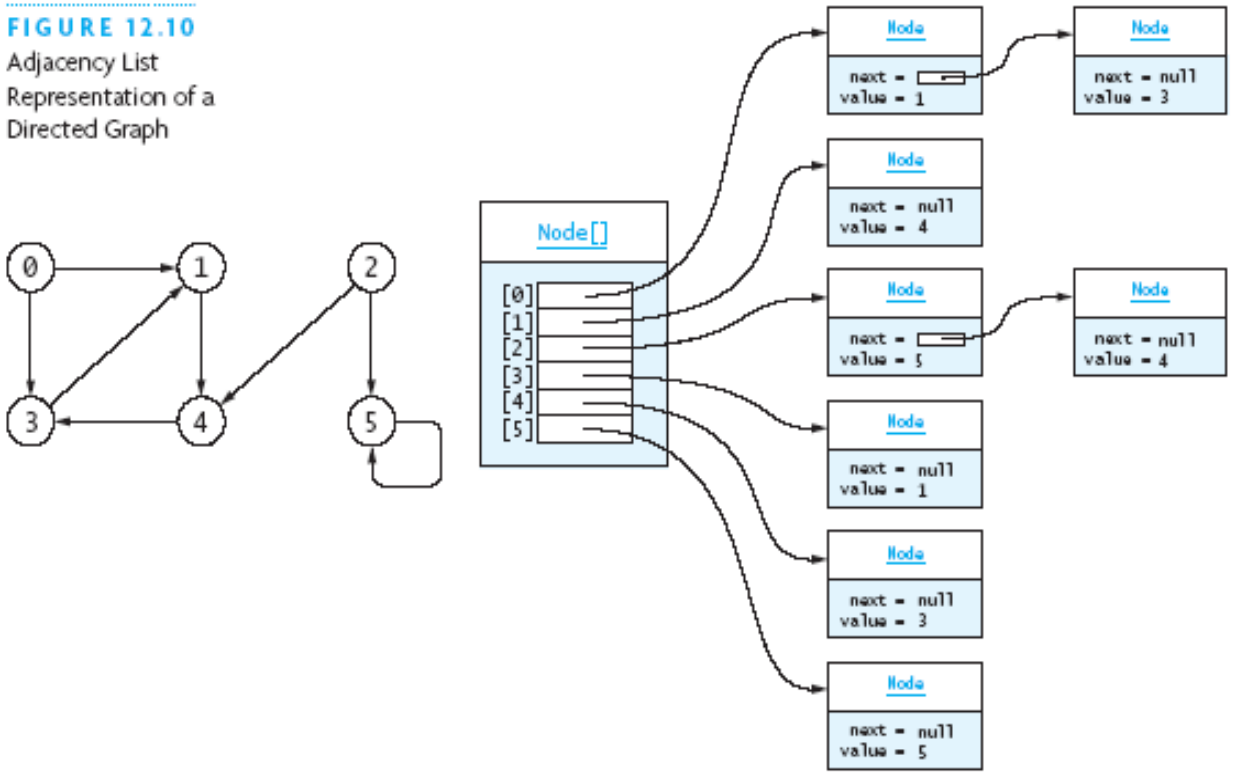
- Two representations of graphs are most common
  - Edges are represented by an array of lists called **adjacency lists**, where each list stores the vertices adjacent to a particular vertex
  - Edges are represented by a two dimensional array, called an **adjacency matrix**

Adjacency List

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

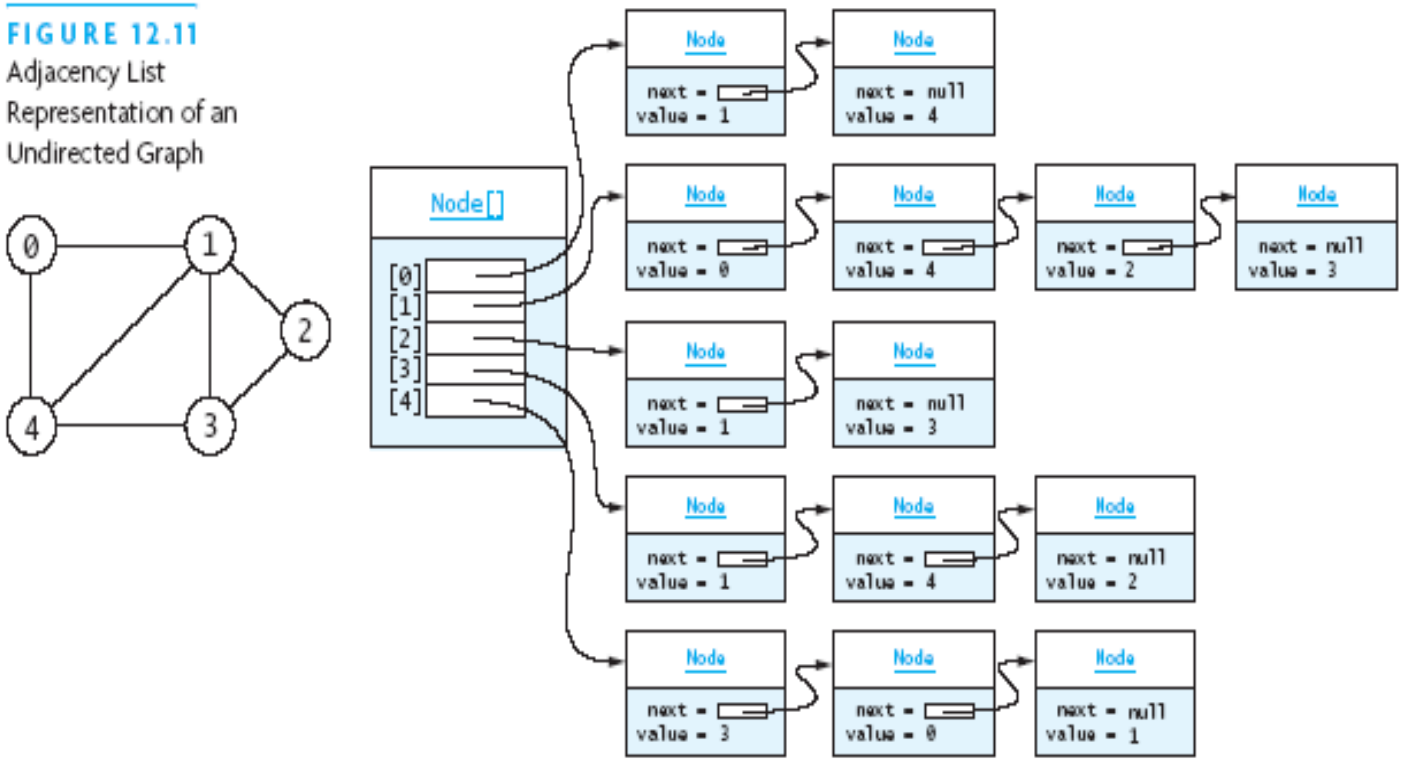
Adjacency List (continued)

FIGURE 12.10
Adjacency List Representation of a Directed Graph



Adjacency List (continued)

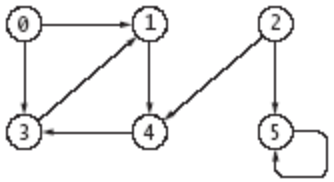
FIGURE 12.11
Adjacency List Representation of an Undirected Graph



Adjacency Matrix

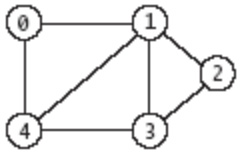
- Uses a two-dimensional array to represent a graph
- For an unweighted graph, the entries can be Boolean values
- For a weighted graph, the matrix would contain the weights

FIGURE 12.12
A Directed Graph and the Corresponding Adjacency Matrix



Row \ Column	[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

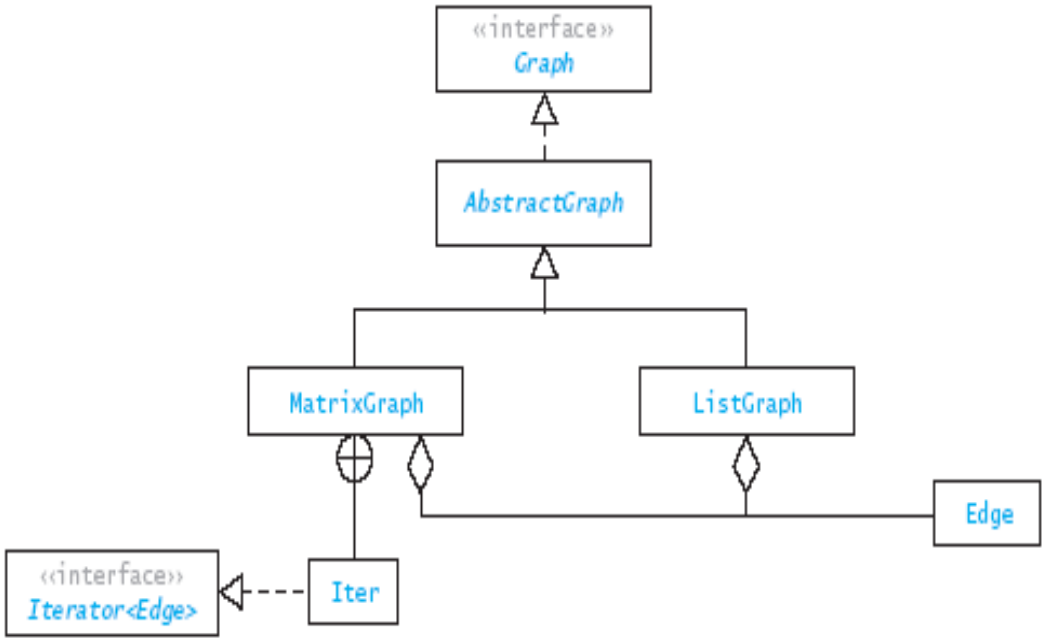


Row \ Column	[0]	[1]	[2]	[3]	[4]
[0]		1.0			1.0
[1]	1.0		1.0	1.0	1.0
[2]		1.0		1.0	
[3]		1.0	1.0		1.0
[4]	1.0	1.0		1.0	

Overview of the Graph Class Hierarchy

FIGURE 12.14

UML Class Diagram of Graph Class Hierarchy



Abstract Class AbstractGraph

Data Field	Attribute
private boolean directed	Boolean value indicating whether or not the graph is directed
private int numV	The number of vertices
Constructor	**Purpose**
public AbstractGraph(boolean isDirected, int numV)	
Method	**Action**
public boolean isDirected() public int getNumV()	Getters
public void loadEdgesFromFile (BufferedReader bR)	Load data from a buffered reader object
public static Graph CreateGraph (BufferedReader bR, boolean directed, String type)	Factory method for graph construction

ListGraph Class

Data Field	Attribute
private List<Edge>[] edges	List object to maintain list of edges
Constructor	**Purpose**
public ListGraph(boolean directed, int numV)	Constructor
Method	**Action**
public Iterator<Edge> edgeInterator(int source)	Produce an iterator of the edges from a given source vertex
public Edge getEdge(int source, int dest)	Returns an edge from a given source to a given destination
public void insert(Edge e)	Insert an edge
public boolen isEdge(int source, int dest)	Return whether there is an edge from source to dest

Graph Traversals

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

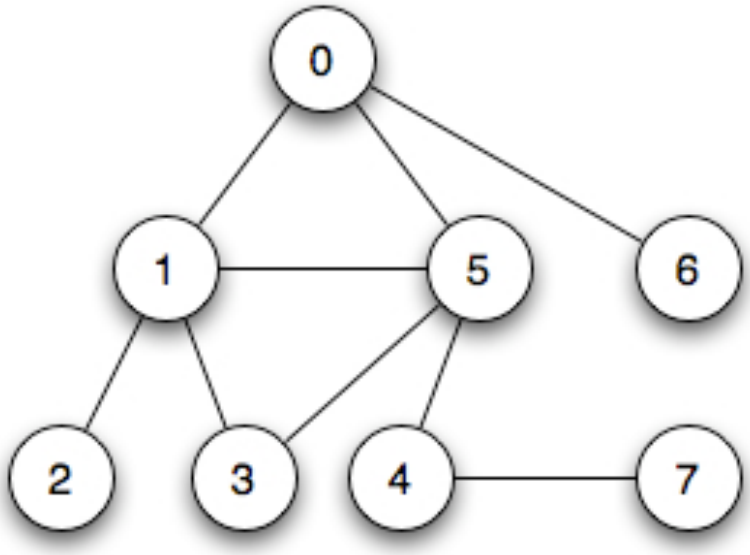
Breadth-First Search

- Visit the first vertex, then visit all the neighbors of the first vertex, then neighbors of the neighbors of the first vertex, then neighbors of the neighbors of the neighbors of the first vertex, and so on.
- In other words, it visits the in the increasing order of the number of edges that must be followed to visit starting from the first vertex.
- Can be started from any first vertex.

Algorithm for Breadth-First Search Using a Queue

- Create a queue *myQ* consisting only of the first vertex
- Create an array *isVisited* of size numV whose initial values are all false except for the start vertex
- Output the start vertex
- While (at least one array element of *isVisited* is false) do:
  - Poll a vertex v from *myQ*
  - For each neighbor w of v:
    - If isVisited has the value false for w:
      - Offer w
      - Set the value of isVisited for w to true
      - Output w

Example



Queue	Output	Array
0	0	TFFFFFFF
156	0156	TTFFFTTF
5623	015623	TTTTFTTF
6234	0156234	TTTTTTTF
234	0156234	TTTTTTTF
34	0156234	TTTTTTTF
4	0156234	TTTTTTTF
7	01562347	TTTTTTTT

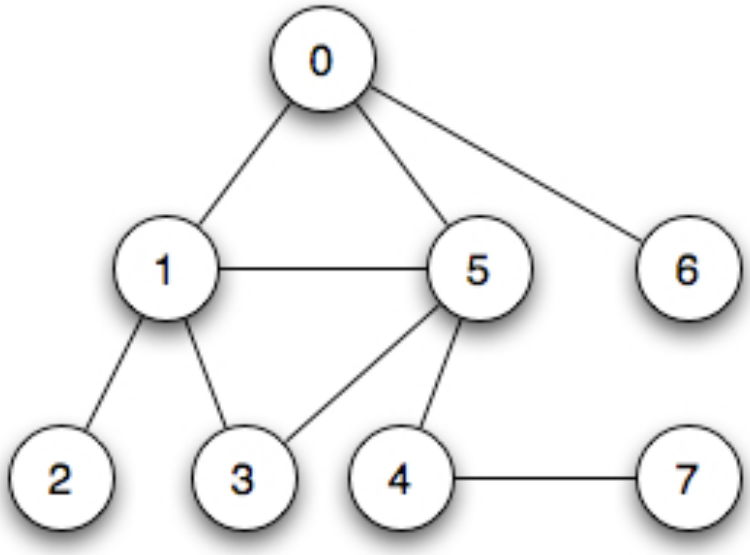
Depth-First Search

- 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

Algorithm for DFS using a Recursive Call

- Create an array *isVisited* of size numV whose initial entries are all false
- For each *u* in range [0,numV-1]:
  - Set *p* to null
  - If *isVisited*[u] is false, then execute search at *u* with *p* as *pVertex*
- Search at *u* with *pVertex*:
  - Output *u*
  - Record *pVertex* as parent of *u*
  - Set isVisited[u] to true
  - For each neighbor *v* of *u*:
    - If *isVisited*[*v*] is false, execute search at *v* with *u* as *pVertex*

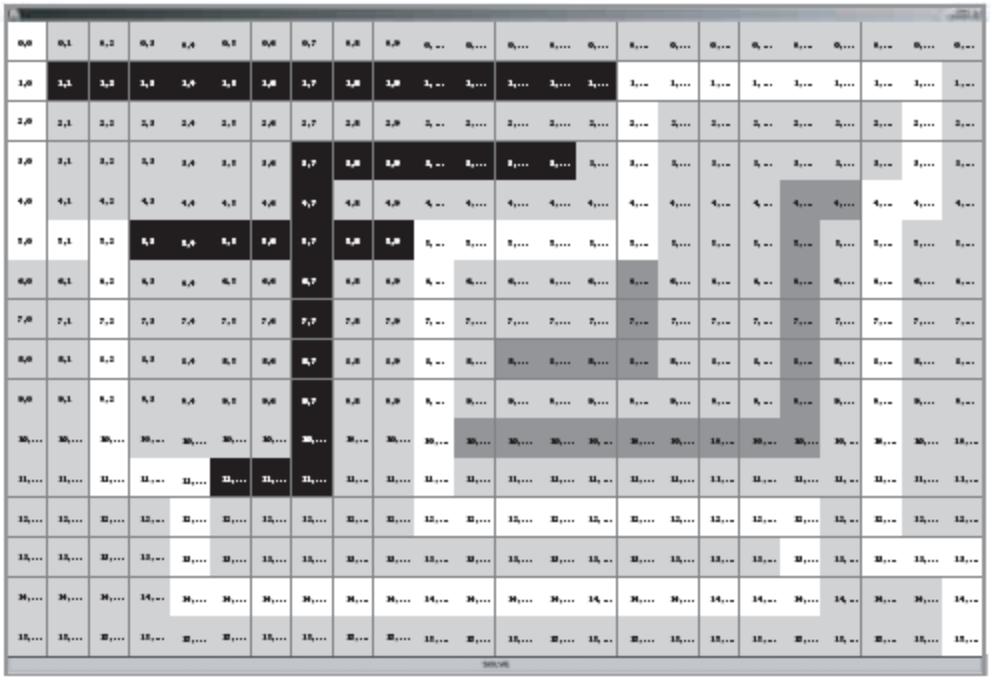
Example of DFS



Stack	Output	isVisited
0	0	TFFFFFFF
01	01	TTFFFFFF
012	012	TTTFFFFF
013	0123	TTTTFFFF
0135	01235	TTTTFTFF
01354	012354	TTTTTTFF
013547	0123547	TTTTTTFT
06	01235476	TTTTTTTT

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

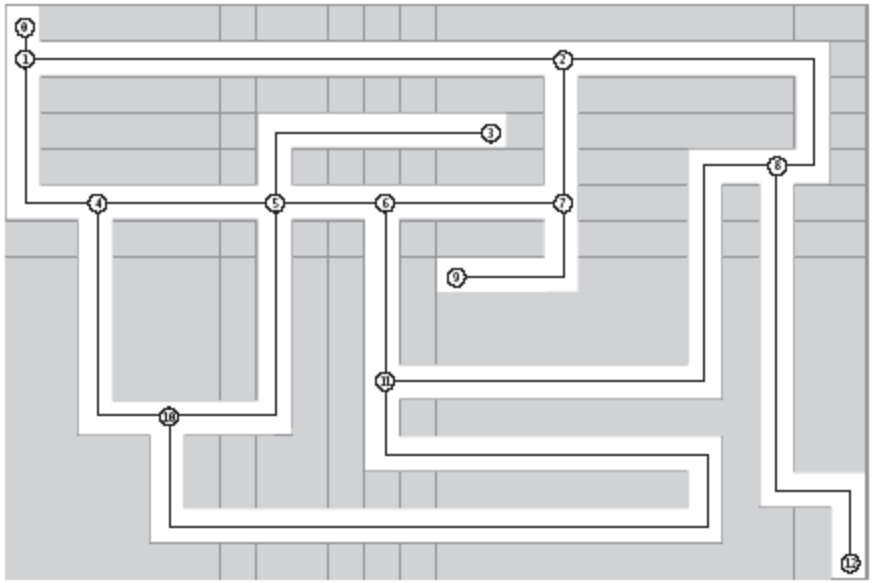
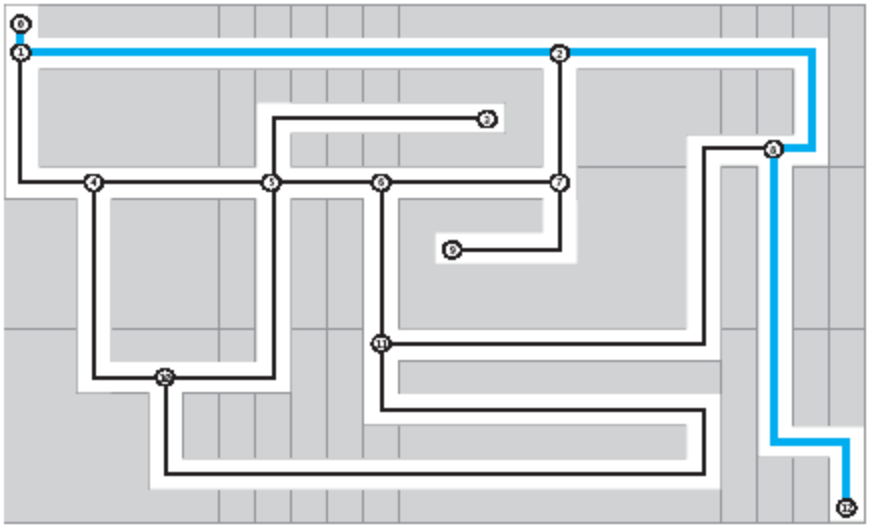


FIGURE 12.23
Solution to Maze in Figure 12.21

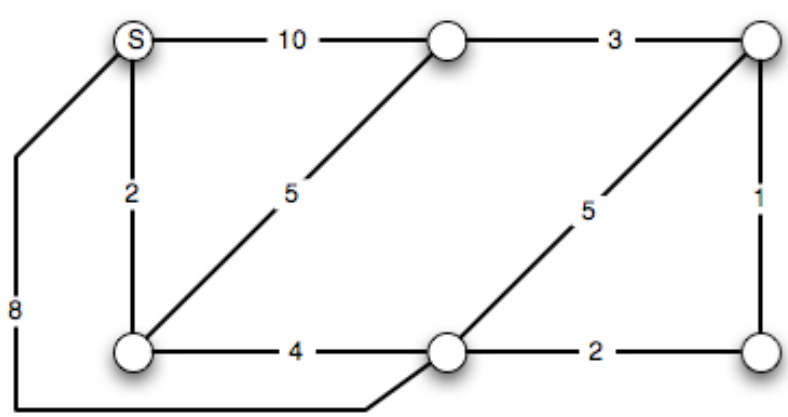


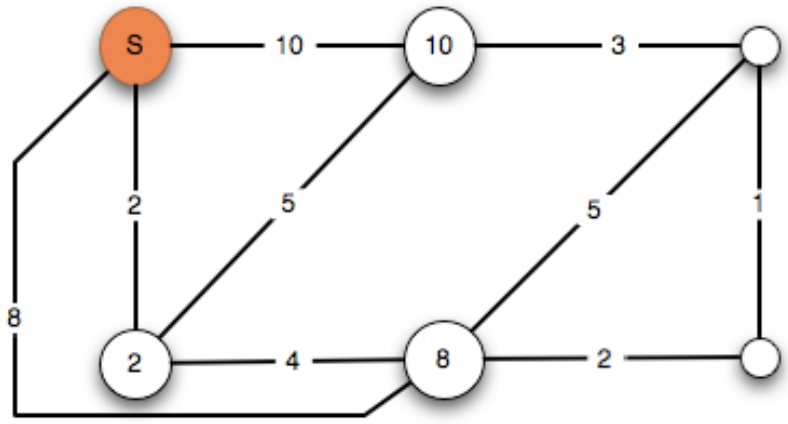
Algorithms Using Weighted Graphs

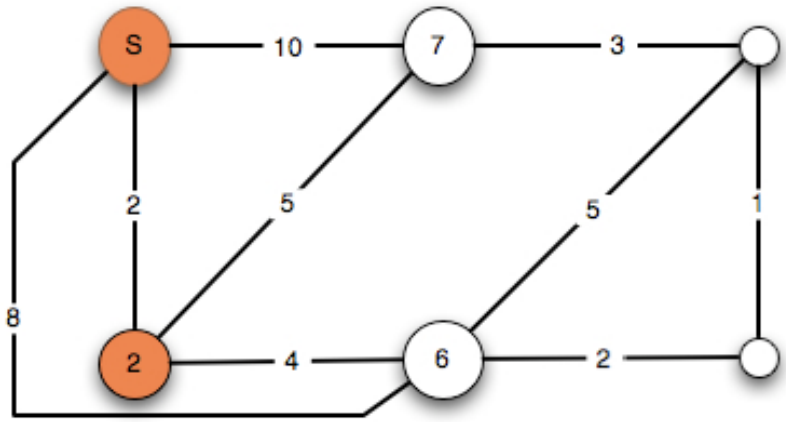
- Finding the shortest path from a vertex to all other vertices
  - Solution formulated by Dijkstra

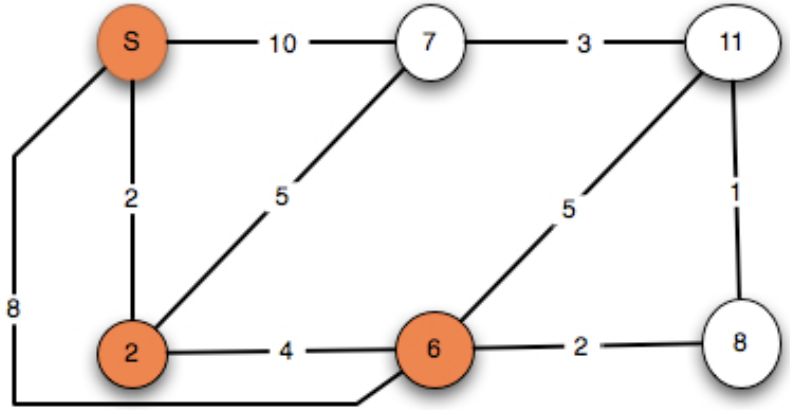
Dijkstra's Algorithm

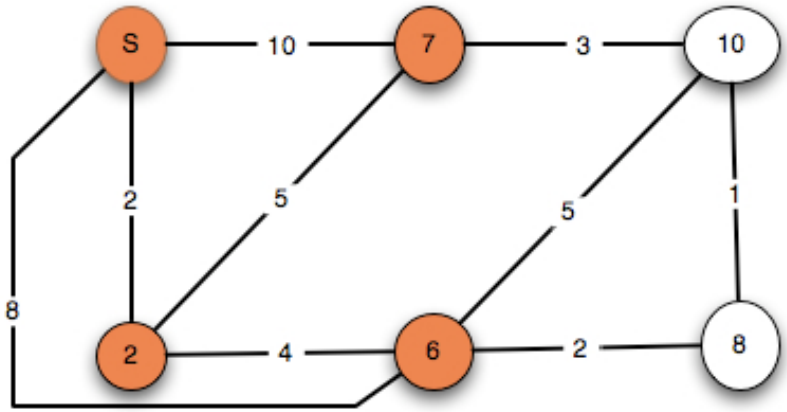
```
1.  Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.  for all v in V–S
3.          Set p[v] to s.
4.          if there is an edge (s, v)
5.                  Set d[v] to w(s, v).
            else
6.                  Set d[v] to ∞.
7.  while V–S is not empty
8.          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.
```

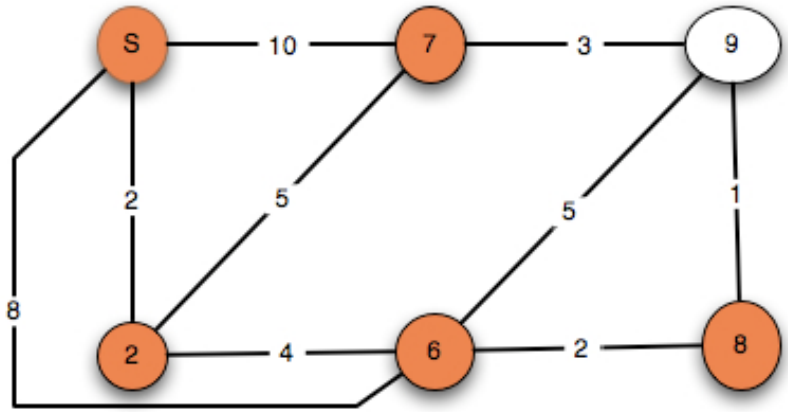

S
10
3
2
5
5
1
8
4
2


S
10
10
3
2
5
5
1
8
2
4
8
2


S
10
7
3
2
5
5
1
8
2
4
6
2


S
10
7
3
11
2
5
5
1
8
2
4
6
2
8


S
10
7
3
10
2
5
5
1
8
2
4
6
2
8


S
10
7
3
9
2
5
5
1
8
2
4
6
2
8

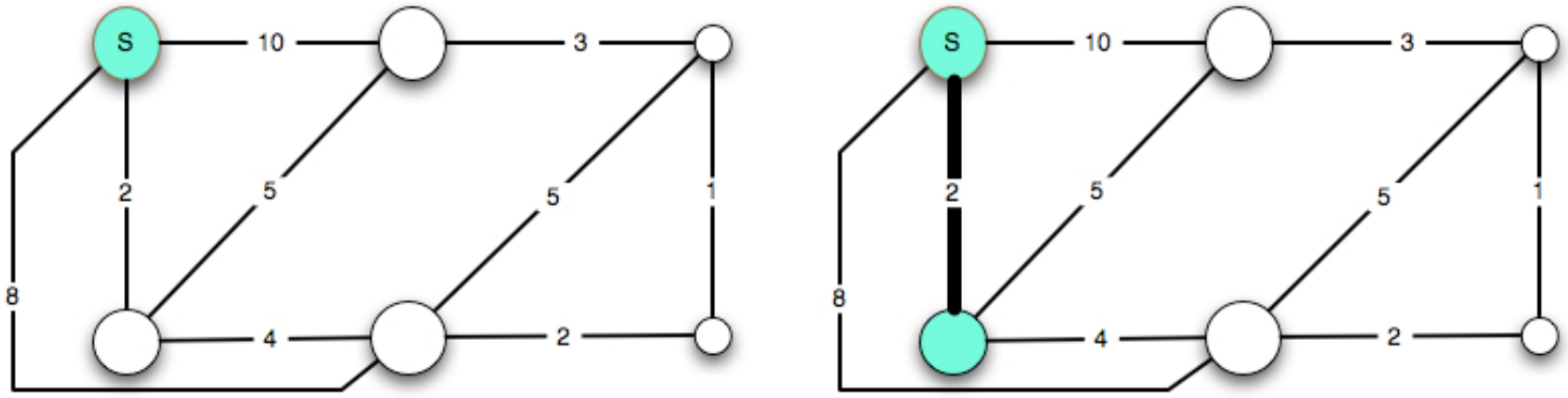
Algorithms Using Weighted Graphs (continued)

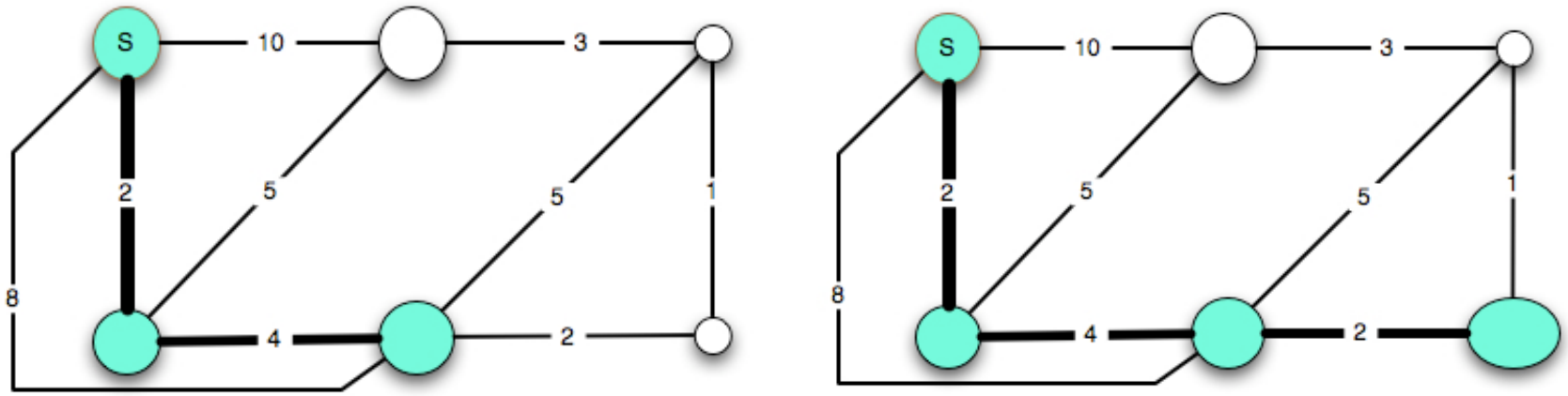
- A minimum spanning tree is a subset of the edges of a graph such that there is only one edge between each vertex, and all of the vertices are connected
- The cost of a spanning tree is the sum of the weights of the edges
- We want to find the minimum spanning tree or the spanning tree with the smallest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

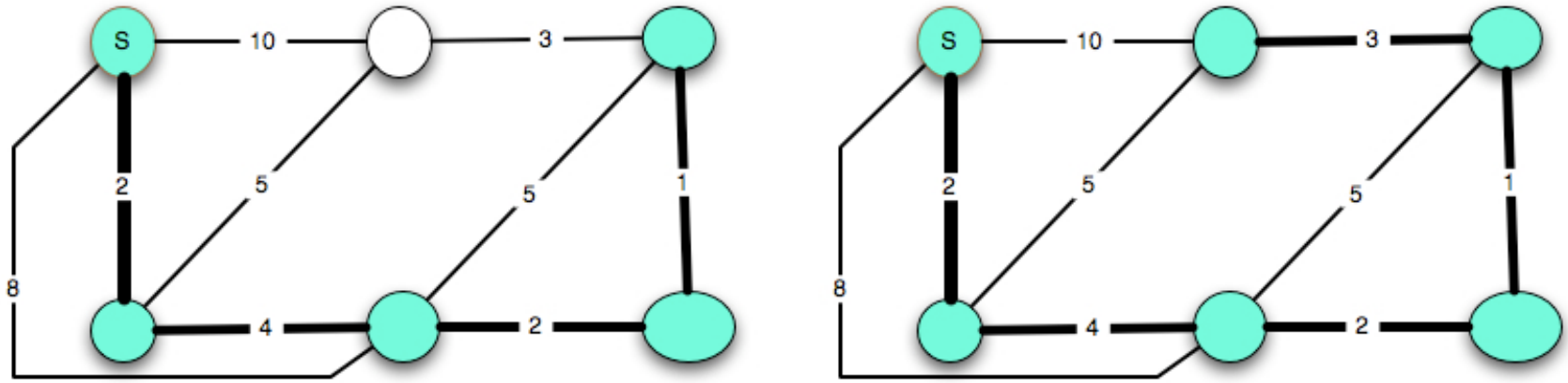
Prim's Algorithm

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.
```


S
10
3
2
5
5
1
8
4
2
S
10
3
2
5
5
1
8
4
2


S
10
3
2
5
5
1
8
4
2
S
10
3
2
5
5
1
8
4
2


S
10
3
2
5
5
1
8
4
2
S
10
3
2
5
5
1
8
4
2