Chapter 14
Graphs and Paths
**Figure 14.1**
A directed graph
**figure 14.2**

Adjacency list representation of the graph shown in Figure 14.1; the nodes in list \( i \) represent vertices adjacent to \( i \) and the cost of the connecting edge.
1 import java.io.FileReader;
2 import java.io.InputStreamReader;
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.util.StringTokenizer;
6
7 import java.util.Collection;
8 import java.util.List;
9 import java.util.LinkedList;
10 import java.util.Map;
11 import java.util.HashMap;
12 import java.util.Iterator;
13 import java.util.Queue;
14 import java.util.PriorityQueue;
15 import java.util.NoSuchElementException;

**figure 14.3**
The import directives for the Graph class.
**Figure 14.4**

An abstract scenario of the data structures used in a shortest-path calculation, with an input graph taken from a file. The shortest weighted path from A to C is A to B to E to D to C (cost is 76).

<table>
<thead>
<tr>
<th>Input</th>
<th>Graph table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>D (0)</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>E (4)</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>B (3)</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>A (2)</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>C (1)</td>
</tr>
</tbody>
</table>

- **A** to **B**: 12
- **B** to **E**: 11
- **A** to **C**: 19
- **B** to **D**: 23
- **C** to **D**: 87
- **D** to **C**: 10

---

**Dictionary**

- D (0)
- E (4)
- B (3)
- A (2)
- C (1)
**figure 14.5**

Data structures used in a shortest-path calculation, with an input graph taken from a file; the shortest weighted path from A to C is A to B to E to D to C (cost is 76).

**Legend:** Dark-bordered boxes are Vertex objects. The unshaded portion in each box contains the name and adjacency list and does not change when shortest-path computation is performed. Each adjacency list entry contains an Edge that stores a reference to another Vertex object and the edge cost. Shaded portion is `dist` and `prev`, filled in after shortest path computation runs.

Dark arrows emanate from `vertexMap`. Light arrows are adjacency list entries. Dashed arrows are the `prev` data member that results from a shortest-path computation.
// Represents an edge in the graph.
class Edge {
    public Vertex dest;       // Second vertex in Edge
    public double cost;       // Edge cost

    public Edge( Vertex d, double c )
    {
        dest = d;
        cost = c;
    }
}
// Represents a vertex in the graph.
class Vertex {
    public String name; // Vertex name
    public List<Edge> adj; // Adjacent vertices
    public double dist; // Cost
    public Vertex prev; // Previous vertex on shortest path
    public int scratch; // Extra variable used in algorithm

    public Vertex(String nm) {
        name = nm; adj = new LinkedList<Edge>(); reset();
    }

    public void reset() {
        dist = Graph.INFINITY; prev = null; pos = null; scratch = 0;
    }
}
1 // Graph class: evaluate shortest paths.
2 //
3 // CONSTRUCTION: with no parameters.
4 //
5 // ****************************PUBLIC OPERATIONS*****************************
6 // void addEdge( String v, String w, double cw )
7 // -- Add additional edge
8 // void printPath( String s )  -- Print path after alg is run
9 // void unweighted( String s )  -- Single-source unweighted
10 // void dijkstra( String s )   -- Single-source weighted
11 // void negative( String s )  -- Single-source negative weighted
12 // void acyclic( String s )   -- Single-source acyclic
13 // ****************************ERRORS*****************************
14 // Some error checking is performed to make sure that graph is ok
15 // and that graph satisfies properties needed by each
16 // algorithm. Exceptions are thrown if errors are detected.
17 //
18 public class Graph
19 {  
20     public static final double INFINITY = Double.MAX_VALUE;
21  
22     public void addEdge( String sourceName, String destName, double cost )
23     { /* Figure 14.10 */ }
24     public void printPath( String destName )
25     { /* Figure 14.13 */ }
26     public void unweighted( String startName )
27     { /* Figure 14.22 */ }
28     public void dijkstra( String startName )
29     { /* Figure 14.27 */ }
30     public void negative( String startName )
31     { /* Figure 14.29 */ }
32     public void acyclic( String startName )
33     { /* Figure 14.32 */ }
34  
35     private Vertex getVertex( String vertexName )
36     { /* Figure 14.9 */ }
37     private void printPath( Vertex dest )
38     { /* Figure 14.12 */ }
39     private void clearAll()  
40     { /* Figure 14.11 */  
41  
42     private Map<String,Vertex> vertexMap = new HashMap<String,Vertex>();
43  }
44  
45 // Used to signal violations of preconditions for
46 // various shortest path algorithms.
47 class GraphException extends RuntimeException
48 {  
49     public GraphException( String name )
50     { super( name ); }
51 }
/**
 * If vertexName is not present, add it to vertexMap.
 * In either case, return the Vertex.
 */
private Vertex getVertex( String vertexName )
{
    Vertex v = vertexMap.get( vertexName );
    if( v == null )
    {
        v = new Vertex( vertexName );
        vertexMap.put( vertexName, v );
    }
    return v;
}
/**
 * Add a new edge to the graph.
 */

public void addEdge( String sourceName, String destName, double cost )
{
    Vertex v = getVertex( sourceName );
    Vertex w = getVertex( destName );
    v.adj.add( new Edge( w, cost ) );
}

**figure 14.10**

Add an edge to the graph
figure 14.11

Private routine for initializing the output members for use by the shortest-path algorithms

```java
/**
 * Initializes the vertex output info prior to running any shortest path algorithm.
 */
private void clearAll()
{
    for (Vertex v : vertexMap.values())
    
        v.reset();
}
A recursive routine for printing the shortest path

```java
/**
 * Recursive routine to print shortest path to dest
 * after running shortest path algorithm. The path
 * is known to exist.
 */

private void printPath( Vertex dest )
{
    if( dest.prev != null )
    {
        printPath( dest.prev );
        System.out.print( " to " );
    }
    System.out.print( dest.name );
}
```
A routine for printing the shortest path by consulting the graph table (see Figure 14.5)

```java
/**
 * Driver routine to handle unreaches and print total cost.
 * It calls recursive routine to print shortest path to
 * destNode after a shortest path algorithm has run.
 */

public void printPath( String destName )
{
    Vertex w = vertexMap.get( destName );
    if( w == null )
        throw new NoSuchElementException();
    else if( w.dist == INFINITY )
        System.out.println( destName + " is unreachable" );
    else
    {
        System.out.print( "(Cost is: " + w.dist + ") " );
        printPath( w );
        System.out.println( );
    }
}
```
/**
 * A main routine that
 * 1. Reads a file (supplied as a command-line parameter)
 * containing edges.
 * 2. Forms the graph.
 * 3. Repeatedly prompts for two vertices and
 * runs the shortest path algorithm.
 * The data file is a sequence of lines of the format
 * source destination.
 */

public static void main( String [] args )
{
    Graph g = new Graph();
    try
    {
        FileReader fin = new FileReader( args[0] );
        BufferedReader graphFile = new BufferedReader( fin );

        // Read the edges and insert
        String line;
        while( ( line = graphFile.readLine() ) != null )
        {
            StringTokenizer st = new StringTokenizer( line );

            try
            {
                if( st.countTokens( ) != 3 )
                {
                    System.err.println( "Skipping bad line "+ line );
                    continue;
                }
                String source = st.nextToken( );
                String dest = st.nextToken( );
                int cost = Integer.parseInt( st.nextToken( ) );
                g.addEdge( source, dest, cost );
            }
            catch( NumberFormatException e )
            {
                System.err.println( "Skipping bad line "+ line );
            }
            catch( IOException e )
            {
                System.err.println( e );
            }
        }
        System.out.println( "File read... ");
        System.out.println( g.vertexMap.size() + " vertices" );
        BufferedReader in = new BufferedReader(
            new InputStreamReader( System.in ) );
        while( processRequest( in, g ) )
    }
}
/**
 * Process a request; return false if end of file.
 */
public static boolean processRequest( BufferedReader in, Graph g )
{
    String startName = null;
    String destName = null;
    String alg = null;

    try
    {
        System.out.print( "Enter start node:"; 
        if( ( startName = in.readLine() ) == null )
            return false;
        System.out.print( "Enter destination node:"; 
        if( ( destName = in.readLine() ) == null )
            return false;
        System.out.print( " Enter algorithm (u, d, n, a): "; 
        if( ( alg = in.readLine() ) == null )
            return false;

        if( alg.equals( "u" ) )
            g.unweighted( startName );
        else if( alg.equals( "d" ) )
            g.dijkstra( startName );
        else if( alg.equals( "n" ) )
            g.negative( startName );
        else if( alg.equals( "a" ) )
            g.acyclic( startName );

        g.printPath( destName );
    }
    catch( IOException e )
    {
        System.err.println( e );
    }
    catch( NoSuchElementException e )
    {
        System.err.println( e );
    }
    catch( GraphException e )
    {
        System.err.println( e );
    }
    return true;
}
The graph after the starting vertex has been marked as reachable in zero edges.
Figure 14.17

The graph after all the vertices whose path length from the starting vertex is 1 have been found.
**Figure 14.18**
The graph after all the vertices whose shortest path from the starting vertex is 2 have been found.
**Figure 14.19**

The final shortest paths
Figure 14.20
If \( w \) is adjacent to \( v \) and there is a path to \( v \), there also is a path to \( w \).
figure 14.21
Searching the graph in the unweighted shortest-path computation. The darkest-shaded vertices have already been completely processed, the lightest vertices have not yet been used as \( v \), and the medium-shaded vertex is the current vertex, \( v \). The stages proceed left to right, top to bottom, as numbered.
/**
 * Single-source unweighted shortest-path algorithm.
 */
public void unweighted( String startName )
{
    clearAll( );

    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    Queue<Vertex> q = new LinkedList<Vertex>( );
    q.add( start ); start.dist = 0;

    while( !q.isEmpty( ) )
    {
        Vertex v = q.remove( );

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;

            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.prev = v;
                q.add( w );
            }
        }
    }

figure 14.22
The unweighted shortest-path algorithm, using breadth-first search
The eyeball is at $v$ and $w$ is adjacent, so $D_w$ should be lowered to 6.
Figure 14.24

If $D_v$ is minimal among all unseen vertices and if all edge costs are nonnegative, $D_v$ represents the shortest path.
Stages of Dijkstra’s algorithm. The conventions are the same as those in Figure 14.21.
// Represents an entry in the priority queue for Dijkstra's algorithm.
class Path implements Comparable<Path>
{
    public Vertex dest;    // w
    public double cost;    // d(w)

    public Path( Vertex d, double c )
    {
        dest = d;
        cost = c;
    }

    public int compareTo( Path rhs )
    {
        double otherCost = rhs.cost;
        return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
    }
}
/**
 * Single-source weighted shortest-path algorithm.
 */
public void dijkstra( String startName )
{
    PriorityQueue<Path> pq = new PriorityQueue<Path>( );
    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );
    clearAll( );
pq.add( new Path( start, 0 ) ); start.dist = 0;
    int nodesSeen = 0;
    while( !pq.isEmpty() && nodesSeen < vertexMap.size( ) )
    {
        Path vrec = pq.remove( );
        Vertex v = vrec.dest;
        if( v.isScratch != 0 ) // already processed v
            continue;
        v.isScratch = 1;
        nodesSeen++;
        for( Edge e : v.adj )
        {
            Vertex w = e.dest;
            double cvw = e.cost;
            if( cvw < 0 )
                throw new GraphException( "Graph has negative edges" );
            if( w.dist > v.dist + cvw )
            {
                w.dist = v.dist + cvw;
                w.prev = v;
                pq.add( new Path( w, w.dist ) );
            }
        }
    }
}
A graph with a negative-cost cycle
/**
 * Single-source negative-weighted shortest-path algorithm.
 */

public void negative( String startName )
{
    clearAll( );

    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    Queue<Vertex> q = new LinkedList<Vertex>( );
    q.add( start ); start.dist = 0; start.scratch++;

    while( !q.isEmpty() )
    {
        Vertex v = q.removeFirst( );
        if( v.scratch++ > 2 * vertexMap.size( ) )
            throw new GraphException( "Negative cycle detected" );

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;
            double cvw = e.cost;

            if( w.dist > v.dist + cvw )
            {
                w.dist = v.dist + cvw;
                w.prev = v;
                // Enqueue only if not already on the queue
                if( w.scratch++ % 2 == 0 )
                    q.add( w );
                else
                    w.scratch--; // undo the enqueue increment
            }
        }
    }
}

**figure 14.29**
A negative-weighted, shortest-path algorithm: Negative edges are allowed.
A topological sort. The conventions are the same as those in Figure 14.21.
Figure 14.31
The stages of acyclic graph algorithm. The conventions are the same as those in Figure 14.21.
/**
 * Single-source negative-weighted acyclic-graph shortest-path algorithm.
 */
public void acyclic( String startName )
{
    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

clearAll();
    Queue<Vertex> q = new LinkedList<Vertex>();
    start.dist = 0;

    // Compute the indegrees
    Collection<Vertex> vertexSet = vertexMap.values();
    for( Vertex v : vertexSet )
        for( Edge e : v.adj )
            e.destscratch++;

    // Enqueue vertices of indegree zero
    for( Vertex v : vertexSet )
        if( v.scratch == 0 )
            q.add( v );

    int iterations;
    for( iterations = 0; !q.isEmpty(); iterations++ )
    {
        Vertex v = q.remove();

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;
            double cw = e.cost;

            if( --w.scratch == 0 )
                q.add( w );

            if( v.dist == INFINITY )
                continue;

            if( w.dist > v.dist + cw )
            {
                w.dist = v.dist + cw;
                w.prev = v;
            }
        }
    }

    if( iterations != vertexMap.size() )
        throw new GraphException( "Graph has a cycle!" );
}
**Figure 14.33**

An activity-node graph
An event-node graph
Figure 14.35
Earliest completion times
**Figure 14.36**

Latest completion times
**Figure 14.37**

Earliest completion time, latest completion time, and slack (additional edge item)
<table>
<thead>
<tr>
<th>Type of Graph Problem</th>
<th>Running Time</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unweighted</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Weighted, no negative edges</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Weighted, negative edges</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Weighted, acyclic</td>
<td>$O(</td>
<td>E</td>
</tr>
</tbody>
</table>

**Figure 14.38**
Worst-case running times of various graph algorithms