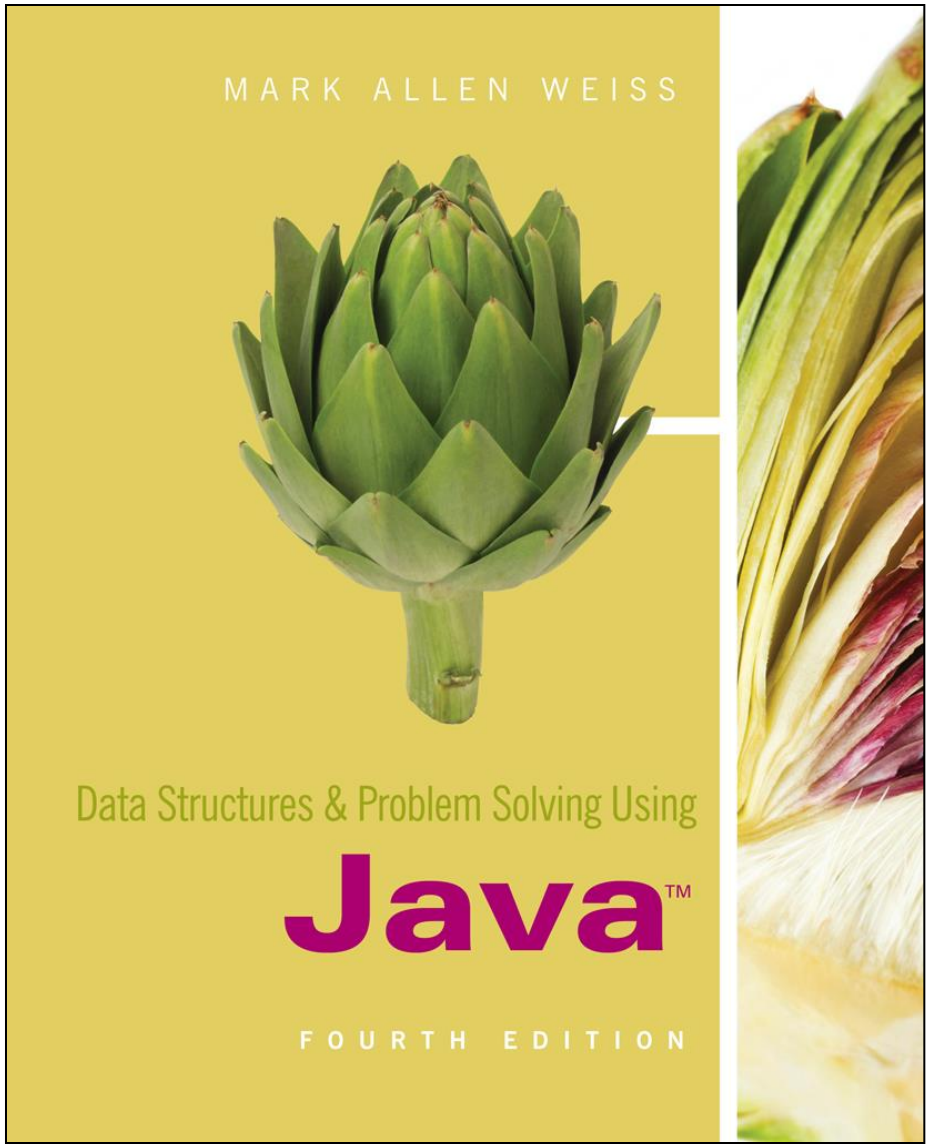
Data Structures
Lesson 10

BSc in Computer Science
University of New York, Tirana

Assoc. Prof. Marenglen Biba

Chapter 14

Graphs and Paths



Addison-Wesley
is an imprint of
PEARSON

Graphs

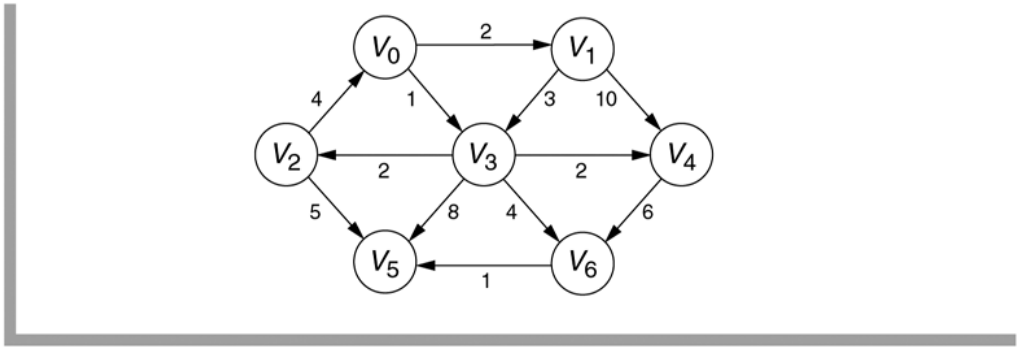
- Graphs are fundamental data structures
- Graph are used in the calculation of shortest paths.
- The computation of shortest paths is a fundamental application in computer science because many interesting situations can be modeled by a graph.
- Finding the fastest routes for a mass transportation system, and routing electronic mail through a network of computers are but a few examples.

Definitions

- A graph consists of a set of vertices and a set of edges that connect the vertices.
- That is, G = (V, E), where V is the set of vertices and E is the set of edges.
- Each edge is a pair (v, w), where v, w ε V.
- Vertices are sometimes called nodes, and edges are sometimes called arcs.
- If the edge pair is ordered, the graph is called a directed graph. Directed graphs are sometimes called digraphs.
- In a digraph, vertex w is adjacent to vertex v if and only if (v, w) ε E.
- Sometimes an edge has a third component, called the edge cost (or weight) that measures the cost of traversing the edge.
- In this class, all graphs are directed.

Directed Graphs

figure 14.1
A directed graph



V = {V0, V1, V2, V3, V4, V5, V6}

$$E = \left\{ \begin{array}{l} (V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), (V_1, V_4, 10) \\ (V_3, V_4, 2), (V_3, V_6, 4), (V_3, V_5, 8), (V_3, V_2, 2) \\ (V_2, V_0, 4), (V_2, V_5, 5), (V_4, V_6, 6), (V_6, V_5, 1) \end{array} \right\}$$

The following vertices are adjacent to V3: V2, V4, V5, and V6. Note that V0 and V1 are not adjacent to V3.
For this graph, |V| = 7 and |E| = 12.

Paths

- A path in a graph is a sequence of vertices connected by edges.
- In other words, w_1, w_2, ..., w_N the sequence of vertices is such that $(w_i, w_{i+1}) \varepsilon E$ for $1<i<N$.
- The path length is the number of edges on the path — namely, N - 1 — also called the unweighted path length.
- The weighted path length is the sum of the costs of the edges on the path.
- For example, V0, V3, V5 is a path from vertex V0 to V5.
- The path length is two edges — the shortest path between V0 and V5, and the weighted path length is 9.

Cost of path

- If cost is important, the weighted shortest path between these vertices has cost 6 and is V0, V3, V6, V5.
- A path may exist from a vertex to itself.
- If this path contains no edges, the path length is 0, which is a convenient way to define an otherwise special case.
- A simple path is a path in which all vertices are distinct, except that the first and last vertices can be the same.

Cycles

- A cycle in a directed graph is a path that begins and ends at the same vertex and contains at least one edge.
- That is, it has a length of at least 1 such that $w_1 = w_N$.
- This cycle is simple if the path is simple.
- A directed acyclic graph (DAG) is a type of directed graph having no cycles.

Real life application

- An example of a real-life situation that can be modeled by a graph is the airport system.
- Each airport is a vertex.
- If there is a nonstop flight between two airports, two vertices are connected by an edge.
- The edge could have a weight representing time, distance, or the cost of the flight.
- In an undirected graph, an edge (v, w) would imply an edge (w, v).
- However, the costs of the edges might be different because flying in different directions might take longer (depending on prevailing winds) or cost more (depending on local taxes).
- Thus we use a directed graph with both edges listed, possibly with different weights. Naturally, we want to determine quickly the best flight between any two airports; best could mean the path with the fewest edges or one, or all, of the weight measures (distance, cost, and so on).

Real life application

- A second example of a real-life situation that can be modeled by a graph is the routing of electronic mail through computer networks.
- Vertices represent computers, the edges represent links between pairs of computers, and the edge costs represent communication costs (phone bill per megabyte), delay costs (seconds per megabyte), or combinations of these and other factors.

Dense graph

- For most graphs, there is likely at most one edge from any vertex v to any other vertex w (allowing one edge in each direction between v and w).
- Consequently, $|E| < |V|^2$.
- When most edges are present, we have $|E| = \Theta(|V|^2)$.
- Such a graph is considered to be a dense graph — that is, it has a large number of edges, generally quadratic.
- In most applications, however, a sparse graph is the norm.

Sparse graphs

- In the airport model, we do not expect direct flights between every pair of airports.
- Instead, a few airports are very well connected and most others have relatively few flights.
- In a complex mass transportation system involving buses and trains, for any one station we have only a few other stations that are directly reachable and thus represented by an edge.
- Moreover, in a computer network most computers are attached to a few other local computers.
- So, in most cases, the graph is relatively sparse, where $|E| = \Theta(|V|)$ or perhaps slightly more (there is no standard definition of sparse).
- The algorithms that we should develop, then, must be efficient for sparse graphs.

Representation

- The first thing to consider is how to represent a graph internally. Assume that the vertices are sequentially numbered starting from 0, as the graph shown in Figure 14.1 suggests.
- One simple way to represent a graph is to use a two-dimensional array called an adjacency matrix.
- For each edge (v, w), we set a[v][w] equal to the edge cost; nonexistent edges can be initialized with a logical INFINITY.
- The initialization of the graph seems to require that the entire adjacency matrix be initialized to INFINITY.
- Then, as an edge is encountered, an appropriate entry is set.
- In this scenario, the initialization takes $O(|V|^2)$ time.
- Although the quadratic initialization cost can be avoided, the space cost is still $O(|V|^2)$, which is fine for dense graphs but completely unacceptable for sparse graphs.

Adjacency lists

- For sparse graphs, a better solution is an adjacency list, which represents a graph by using linear space.
- For each vertex, we keep a list of all adjacent vertices.
- Because each edge appears in a list node, the number of list nodes equals the number of edges.
- Consequently, $O(|E|)$ space is used to store the list nodes.
- *Consequently, we say that the space requirement is $O(|E|)$, or linear in the size of the graph.*

Adjacency lists

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.

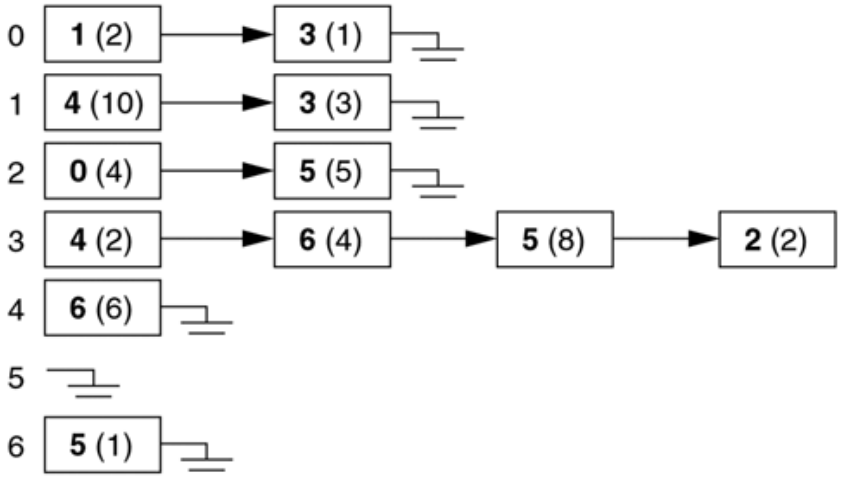
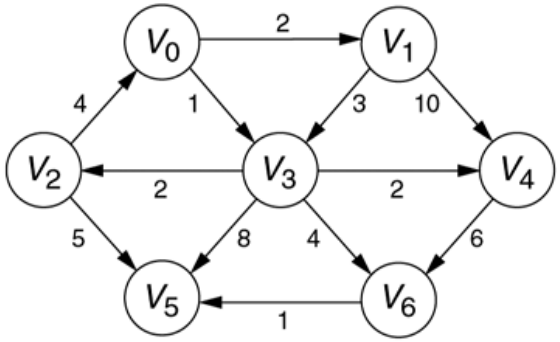


figure 14.1

A directed graph



Linear time construction

- The adjacency list can be constructed in linear time from a list of edges.
- We begin by making all the lists empty.
- When we encounter an edge (v, w, $c_{v,w}$), we add an entry consisting of w and the cost $c_{v,w}$ to v's adjacency list.
- The insertion can be anywhere; inserting it at the front can be done in constant time.
- Each edge can be inserted in constant time, so the entire adjacency list structure can be constructed in linear time.

Linear time construction

- When inserting an edge, we do not check whether it is already present.
- That cannot be done in constant time (using a simple linked list), and doing the check would destroy the linear-time bound for construction.
- In most cases, ignoring this check is unimportant.
- If there are two or more edges of different cost connecting a pair of vertices, any shortest-path algorithm will choose the lower cost edge without resorting to any special processing.
- Note also that ArrayLists can be used instead of linked lists, with the constant-time add operation replacing insertions at the front.

Mapping of vertex names to numbers

- In most real-life applications the vertices have names, which are unknown at compile time, instead of numbers.
- Consequently, we must provide a way to transform names to numbers.
- The easiest way to do so is to provide a map by which we map a vertex name to an internal number ranging from 0 to |V| - 1 (the number of vertices is determined as the program runs).
- The internal numbers are assigned as the graph is read. The first number assigned is 0.
- As each edge is input, we check whether each of the two vertices has been assigned a number, by looking in the map.
- If it has been assigned an internal number, we use it.
- Otherwise, we assign to the vertex the next available number and insert the vertex name and number in the map.

Mapping of vertex names to numbers

- With this transformation, all the graph algorithms use only the internal numbers.
- Eventually, we have to output the real vertex names, not the internal numbers, so for each internal number we must also record the corresponding vertex name.
- One way to do so is to keep a string for each vertex.
- We use this technique to implement a Graph class.
- The class and the shortest-path algorithms require several data structures — namely, a list, a queue, a map, and a priority queue.
- The import directives are shown in Figure 14.3. => next slide

Implementation

```
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

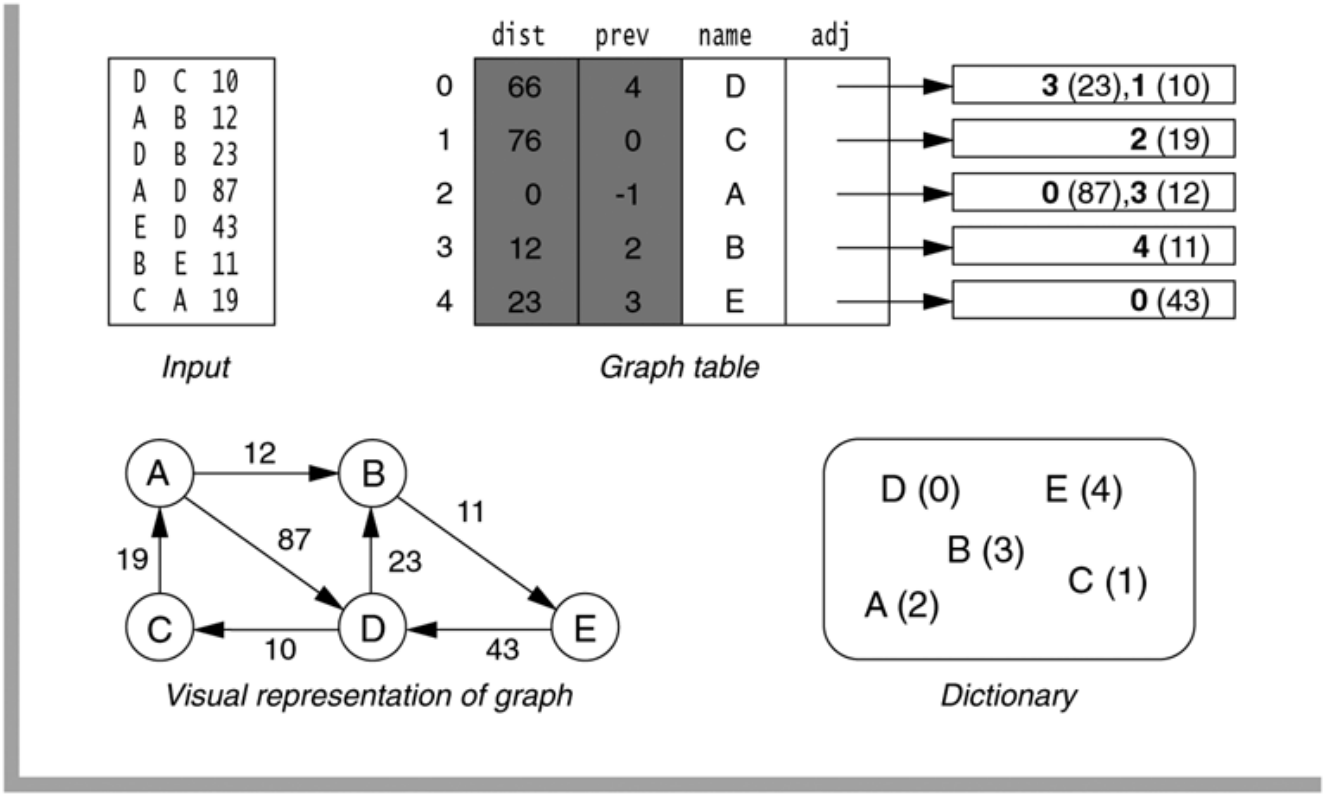
The queue (implemented with a linked list) and priority queue are used in various shortest-path calculations.
The adjacency list is represented with LinkedList.
A HashMap is also used to represent the graph.

Internal numbers

- When we write an actual Java implementation, we do not need internal vertex numbers.
- Instead, each vertex is stored in a Vertex object, and instead of using a number, we can use a reference to the Vertex object as its (uniquely identifying) number.
- However, when describing the algorithms, assuming that vertices are numbered is often convenient, and we occasionally do so.

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).



The Vertex object

- The Vertex object stores information about all the vertices.
- Of particular interest is how it interacts with other Vertex objects.
  - name: The name corresponding to this vertex is established when the vertex is placed in the map and never changes.
    - None of the shortest-path algorithms examines this member.
    - It is used only to print a final path.

The Vertex object

- adj: This list of adjacent vertices is established when the graph is read.
  - None of the shortest-path algorithms changes the list.
  - It is a list of Edge objects that each contain an internal vertex number and edge cost.
  - In reality, Figure 14.5 shows that each Edge object contains a reference to a Vertex and an edge cost and that the list is actually stored by using an ArrayList or LinkedList.

The Vertex object

- dist: The length of the shortest path (either weighted or unweighted, depending on the algorithm) from the starting vertex to this vertex as computed by the shortest-path algorithm.

- prev: The previous vertex on the shortest path to this vertex, which in the abstract (Figure 14.4) is an int but in reality (the code and Figure 14.5) is a reference to a Vertex.

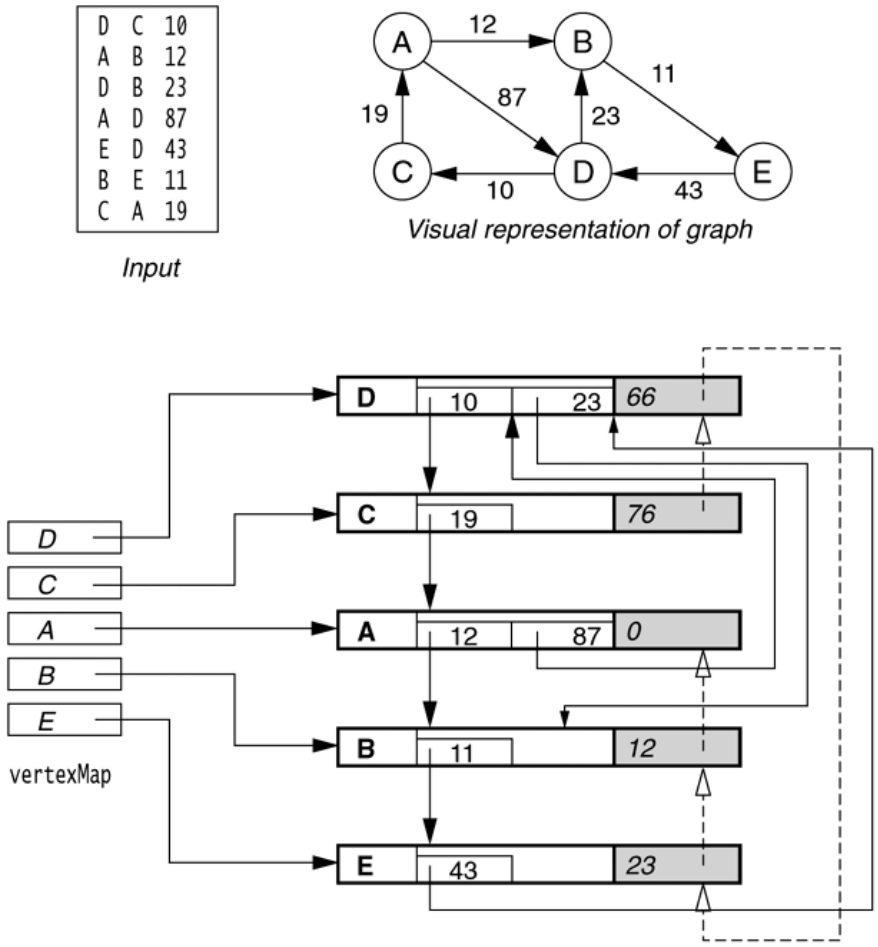


figure 14.5

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.

Single-source algorithms

- The shortest-path algorithms are all single-source algorithms, which begin at some starting point and compute the shortest paths from it to all vertices.
- In this example the starting point is A, and by consulting the map we can find its Vertex object.
- Note that the shortest-path algorithm declares that the shortest path to A is 0.

Edge class

```
1  // Represents an edge in the graph.
2  class Edge
3  {
4      public Vertex dest;        // Second vertex in Edge
5      public double cost;        // Edge cost
6
7      public Edge( Vertex d, double c )
8      {
9          dest = d;
10         cost = c;
11     }
12 }
```

figure 14.6

The basic item stored in an adjacency list

Vertex class

```
1  // Represents a vertex in the graph.
2  class Vertex
3  {
4      public String     name;   // Vertex name
5      public List<Edge> adj;    // Adjacent vertices
6      public double     dist;   // Cost
7      public Vertex     prev;   // Previous vertex on shortest path
8      public int        scratch;// Extra variable used in algorithm
9
10     public Vertex( String nm )
11       { name = nm; adj = new LinkedList<Edge>( ); reset( ); }
12
13     public void reset( )
14      { dist = Graph.INFINITY; prev = null; pos = null; scratch = 0; }
15 }
```

figure 14.7

The `Vertex` class stores information for each vertex

Cost of shortest path

Will explain this later, during algorithm illustration.

```
1  // Graph class: evaluate shortest paths.
2  //
3  // CONSTRUCTION: with no parameters.
4  //
5  // ******************PUBLIC OPERATIONS*********************
6  // void addEdge( String v, String w, double cvw )
7  //                              --> Add additional edge
8  // void printPath( String w )   --> 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 }
```

All vertices in a hash

figure 14.8

The Graph class skeleton

```
1       /**
2        * If vertexName is not present, add it to vertexMap.
3        * In either case, return the Vertex.
4        */
5       private Vertex getVertex( String vertexName )
6       {
7           Vertex v = vertexMap.get( vertexName );
8           if( v == null )
9           {
10              v = new Vertex( vertexName );
11              vertexMap.put( vertexName, v );
12          }
13          return v;
14      }
```

figure 14.9

The `getVertex` routine returns the `Vertex` object that represents `vertexName`, creating the object if it needs to do so

```
1       /**
2        * Add a new edge to the graph.
3        */
4       public void addEdge( String sourceName, String destName, double cost )
5       {
6           Vertex v = getVertex( sourceName );
7           Vertex w = getVertex( destName );
8           v.adj.add( new Edge( w, cost ) );
9       }
```

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

```
1       /**
2        * Initializes the vertex output info prior to running
3        * any shortest path algorithm.
4        */
5       private void clearAll( )
6       {
7           for( Vertex v : vertexMap.values( ) )
8               v.reset( );
9       }
```

figure 14.12

A recursive routine for printing the shortest path

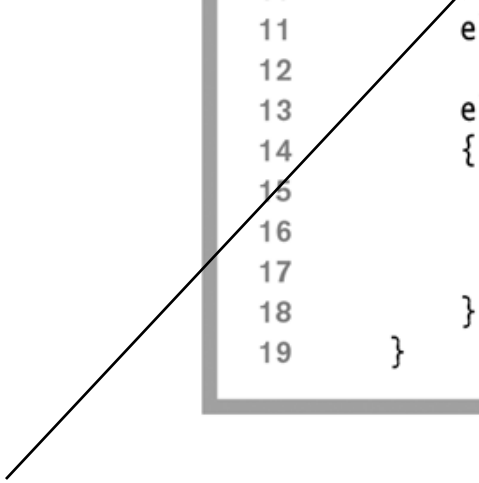
```
1       /**
2        * Recursive routine to print shortest path to dest
3        * after running shortest path algorithm. The path
4        * is known to exist.
5        */
6       private void printPath( Vertex dest )
7       {
8           if( dest.prev != null )
9           {
10              printPath( dest.prev );
11              System.out.print( " to " );
12          }
13          System.out.print( dest.name );
14      }
```

Recursively reconstructs in reverse-order the path to the starting vertex

figure 14.13

A routine for printing the shortest path by consulting the graph table (see Figure 14.5)

```
1       /**
2        * Driver routine to handle unreachables and print total cost.
3        * It calls recursive routine to print shortest path to
4        * destNode after a shortest path algorithm has run.
5        */
6       public void printPath( String destName )
7       {
8           Vertex w = vertexMap.get( destName );
9           if( w == null )
10              throw new NoSuchElementException( );
11          else if( w.dist == INFINITY )
12              System.out.println( destName + " is unreachable" );
13          else
14          {
15              System.out.print( "(Cost is: " + w.dist + ") " );
16              printPath( w );
17              System.out.println( );
18          }
19      }
```



Checks whether a path exists or not

```
1     /**
2      * A main routine that
3      * 1. Reads a file (supplied as a command-line parameter)
4      *    containing edges.
5      * 2. Forms the graph.
6      * 3. Repeatedly prompts for two vertices and
7      *    runs the shortest path algorithm.
8      * The data file is a sequence of lines of the format
9      *    source destination.
10     */
11    public static void main( String [ ] args )
12    {
13        Graph g = new Graph( );
14        try
15        {
16            FileReader fin = new FileReader( args[0] );
17            BufferedReader graphFile = new BufferedReader( fin );
18
19            // Read the edges and insert
20            String line;
21            while( ( line = graphFile.readLine( ) ) != null )
22            {
23                StringTokenizer st = new StringTokenizer( line );
24
25                try
26                {
27                    if( st.countTokens( ) != 3 )
28                    {
29                        System.err.println( "Skipping bad line " + line );
30                        continue;
31                    }
32                    String source  = st.nextToken( );
33                    String dest    = st.nextToken( );
34                    int    cost    = Integer.parseInt( st.nextToken( ) );
35                    g.addEdge( source, dest, cost );
36                }
37                catch( NumberFormatException e )
38                  { System.err.println( "Skipping bad line " + line ); }
39            }
40        }
41        catch( IOException e )
42          { System.err.println( e ); }
43
44        System.out.println( "File read..." );
45        System.out.println( g.vertexMap.size( ) + " vertices" );
46
47        BufferedReader in = new BufferedReader(
48                            new InputStreamReader( System.in ) );
49        while( processRequest( in, g ) )
50            ;
51    }
```

Constructing the graph

Find shortest path

figure 14.14

A simple `main`

```
1     /**
2      * Process a request; return false if end of file.
3      */
4     public static boolean processRequest( BufferedReader in, Graph g )
5     {
6         String startName = null;
7         String destName = null;
8         String alg = null;
9
10        try
11        {
12            System.out.print( "Enter start node:" );
13            if( ( startName = in.readLine( ) ) == null )
14                return false;
15            System.out.print( "Enter destination node:" );
16            if( ( destName = in.readLine( ) ) == null )
17                return false;
18            System.out.print( " Enter algorithm (u, d, n, a ): " );
19            if( ( alg = in.readLine( ) ) == null )
20                return false;
21
22            if( alg.equals( "u" ) )
23                g.unweighted( startName );
24            else if( alg.equals( "d" ) )
25                g.dijkstra( startName );
26            else if( alg.equals( "n" ) )
27                g.negative( startName );
28            else if( alg.equals( "a" ) )
29                g.acyclic( startName );
30
31            g.printPath( destName );
32        }
33        catch( IOException e )
34          { System.err.println( e ); }
35        catch( NoSuchElementException e )
36          { System.err.println( e ); }
37        catch( GraphException e )
38          { System.err.println( e ); }
39        return true;
40    }
```

figure 14.15

For testing purposes, `processRequest` calls one of the shortest-path algorithms

Shortest path algorithms

Unweighted shortest path problem

- Unweighted single-source, shortest-path problem

- *Find the shortest path (measured by number of edges) from a designated vertex S to every vertex.*

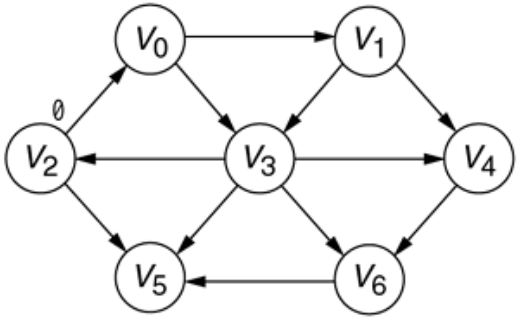


figure 14.16

The graph after the starting vertex has been marked as reachable in zero edges

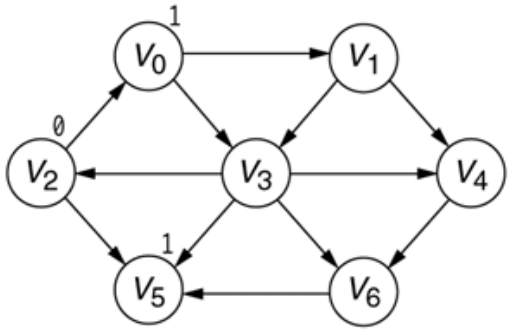
V2 is the starting vertex S. The shortest path from S to V2 is a path of length 0.

Now we can start looking for all vertices that are distance 1 from S.

We can find them by looking at the vertices adjacent to S. If we do so, we see that V0 and V5 are one edge away from S => next slide

figure 14.17

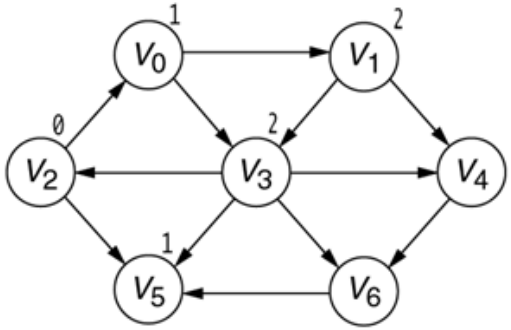
The graph after all the vertices whose path length from the starting vertex is 1 have been found



Next, we find each vertex whose shortest path from S is exactly 2. We do so by finding all the vertices adjacent to V0 or V5 (the vertices at distance 1) whose shortest paths are not already known. This search tells us that the shortest path to V1 and V3 is 2. => next slide

figure 14.18

The graph after all the vertices whose shortest path from the starting vertex is 2 have been found



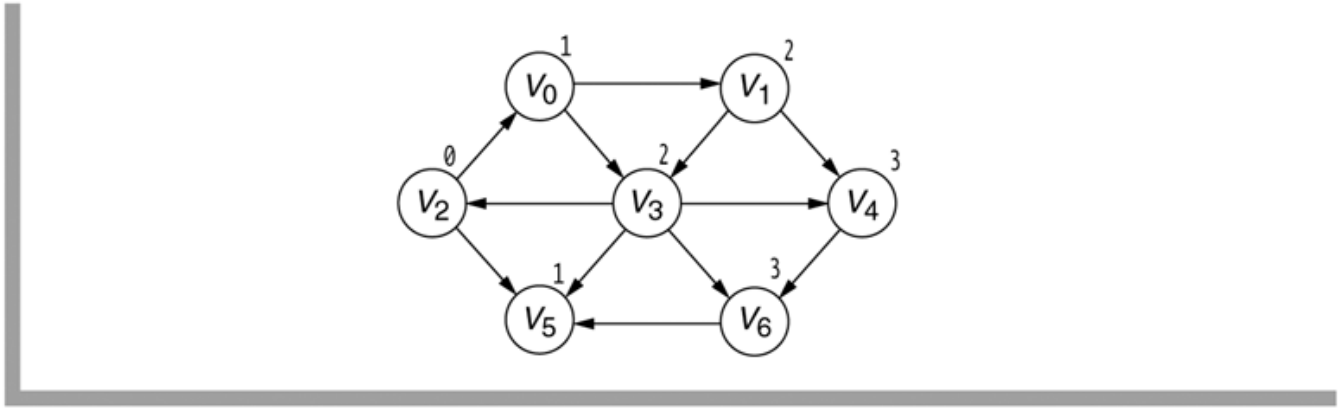
Finally, by examining the vertices adjacent to the recently evaluated V1 and V3, we find that V4 and V6 have a shortest path of 3 edges.

All vertices have now been calculated.

Figure 14.19 shows the final result of the algorithm. => next slide

figure 14.19

The final shortest paths



This strategy for searching a graph is called breadth-first search, which operates by processing vertices in layers:

Those closest to the start are evaluated first, and those most distant are evaluated last.

Fundamental principle

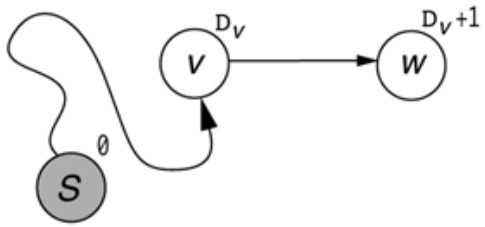


figure 14.20

If *w* is adjacent to *v* and there is a path to *v*, there also is a path to *w*.

If a path to vertex v has cost Dv and w is adjacent to v, then there exists a path to w of cost Dw = Dv + 1.

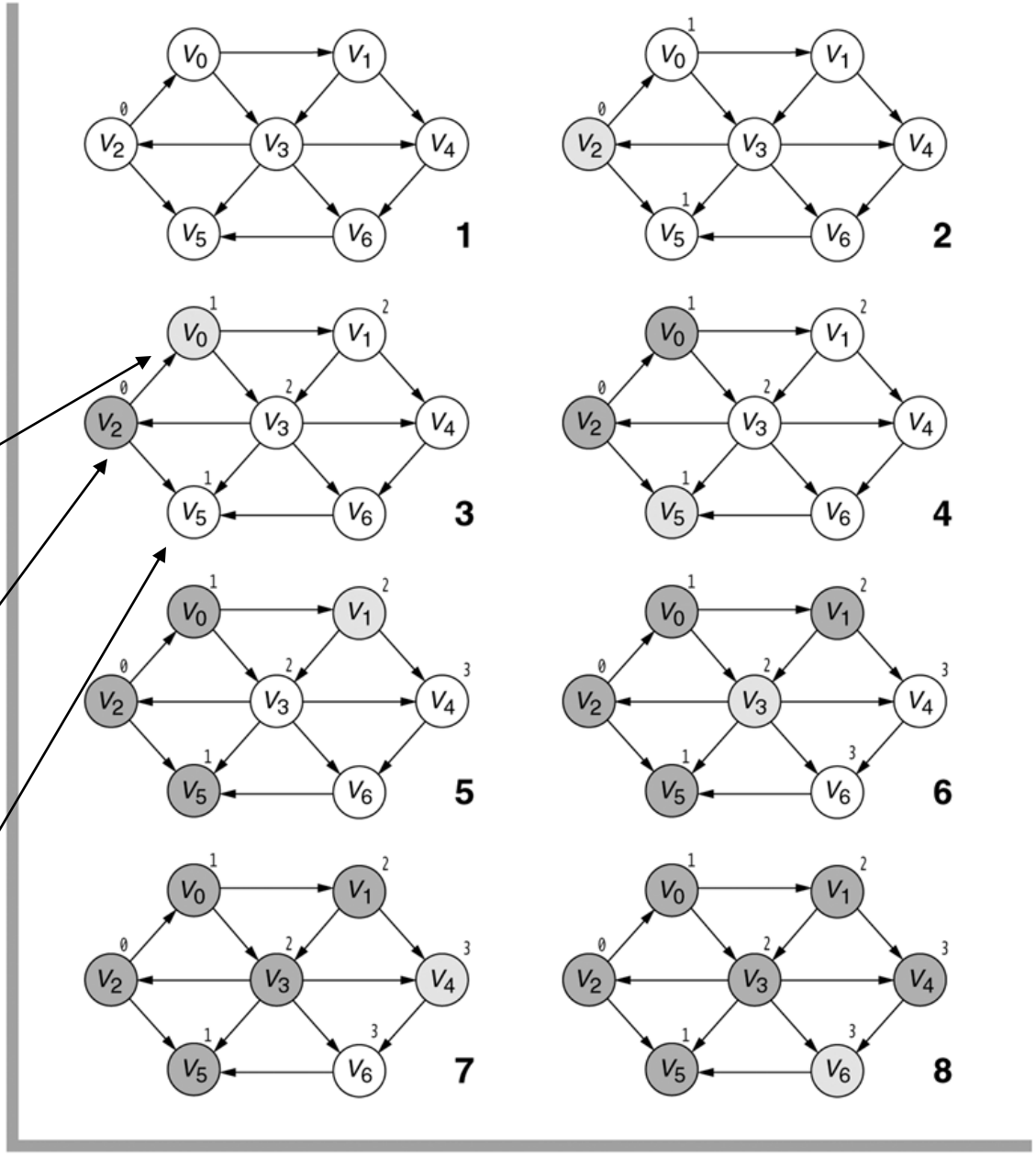
All the shortest-path algorithms work by starting with Dw = ∞ and reducing its value when an appropriate v is scanned.

To do this task efficiently, we must scan vertices v systematically.

When a given v is scanned, we update the vertices w adjacent to v by scanning through v's adjacency list.

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.



Algorithm

- An algorithm for solving the unweighted shortest-path problem is as follows.
- Let D, be the length of the shortest path from S to i. We know that Ds = 0 and that D = ∞ initially for all i ≠ S.
- We maintain a moving eyeball that hops from vertex to vertex and is initially at S.
- If v is the vertex that the eyeball is currently on, then, for all w that are adjacent to v, we set Dw = Dv + 1 if Dw = ∞.
- This reflects the fact that we can get to w by following a path to v and extending the path by the edge (v, w).

Algorithm

- So we update vertices w as they are seen from the eyeball.

- Because the eyeball processes each vertex in order of its distance from the starting vertex and the edge adds exactly 1 to the length of the path to w, we are guaranteed that the *first time Dw is lowered from ∞, it is lowered to the value of the length of the shortest path to w.*

Algorithm

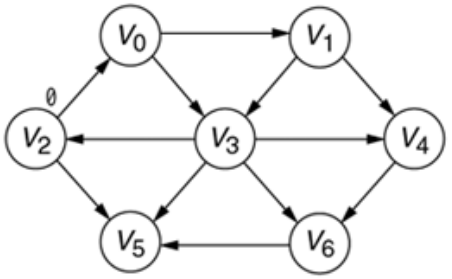
- After we have processed all of v's adjacent vertices, we move the eyeball to another vertex u (that has not been visited by the eyeball) such that Du = Dv.

- If that is not possible, we move to a vertex u that satisfies Du = Dv + 1. If that is not possible, we are done.

Data structures: queue

- When a vertex w has its Dw lowered from ∞, it becomes a candidate for an eyeball visitation at some point in the future.
- That is, after the eyeball visits vertices in the current distance group Dv, it visits the next distance group Dv + 1, which is the group containing w.
- To select a vertex v for the eyeball, we merely choose the front vertex from the queue.
- We start with an empty queue and then we enqueue the starting vertex S.
- A vertex is enqueued and dequeued at most once per shortest-path calculation, and queue operations are constant time.

```
1       /**
2        * Single-source unweighted shortest-path algorithm.
3        */
4       public void unweighted( String startName )
5       {
6           clearAll( );
7
8           Vertex start = vertexMap.get( startName );
9           if( start == null )
10              throw new NoSuchElementException( "Start vertex not found" );
11
12          Queue<Vertex> q = new LinkedList<Vertex>( );
13          q.add( start ); start.dist = 0;
14
15          while( !q.isEmpty( ) )
16          {
17              Vertex v = q.remove( );
18
19              for( Edge e : v.adj )
20              {
21                  Vertex w = e.dest;
22
23                  if( w.dist == INFINITY )
24                  {
25                      w.dist = v.dist + 1;
26                      w.prev = v;
27                      q.add( w );
28                  }
29              }
30          }
31      }
```



Queue

				2(0)
			5(1)	0(1)
		3(2)	1(2)	5(1)
			3(2)	1(2)
			4(3)	3(2)
			6(3)	4(3)
				6(3)

Add candidates in the queue

figure 14.22

The unweighted shortest-path algorithm, using breadth-first search

Other algorithms

- Djikstra Algorithm is used to solve the positive-weighted, shortest-path problem,
  - Algorithms course next year

Readings

- Chapter 14