CS 3410 – Ch 14 – Graphs and Paths

Sections	Pages	Exercises
14.1-14.3	527-552	1,2,5,7-9

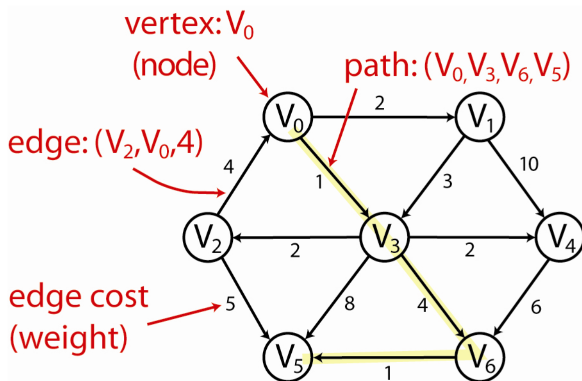
14.1 – Definitions

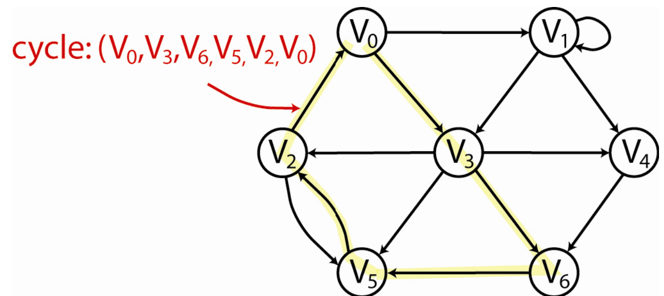
1. A *vertex* (*node*) and an *edge* are the basic building blocks of a *graph*. Two *vertices,* (v, w) may be connected by an *edge,* $e = (v, w)$.

2. A graph is a set of vertices, V and a set of edges, E that connect the vertices: $G = (V, E)$. For example:

   $V = \{V_0, V_1, V_2, V_3, V_4, V_5, V_5\},\ |V| = 7$

   $E = \{(V_0, V_1, 2), (V_0, V_3, 1), (V_1, V_3, 3), \ldots, (V_6, V_5, 1)\},\ |E| = 12$



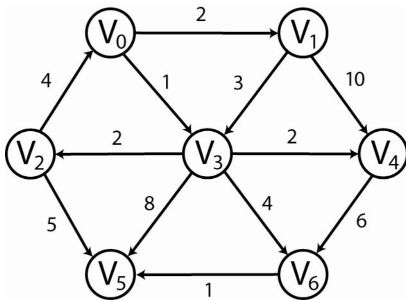


3. From the perspective of graph theory a vertex has no attributes; however, in the context of real problems, we may add structure to vertices. For instance, the vertices may represent airports or molecules.
4. If the edge pair is ordered, we say that they are *directed edges* and the graph is a *directed graph*.
5. In a directed graph, a vertex w is said to be *adjacent* to vertex v if and only if $(v, w) \in E$. In other words if there is a directed edge between the two vertices, that begins at v and ends at w. In the graph above, V_3 and V_4 are adjacent to V_1.
6. An edge may also contain an *edge cost* (or *weight*) that measures the cost of traversing the edge. For example, the "cost" to travel from V_1 to V_4 is 10. Thus, we modify the notation for an edge to include this cost, $e = (v, w, c)$.
7. A *path* is a sequence of vertices connected by edges.
8. The *path length* is the number of edges on the path (which is the number of vertices – 1).
9. The *weighted path length* is the sum of the weights of the edges on the path.
10. A path may exist from a vertex to itself. If this path contains no edges, it has length 0.
11. A *simple path* is a path where all the vertices are distinct, except possibly the first and last.
12. A *cycle* in a directed graph is a path that begins and ends at the same vertex and contains at least one edge.
13. A *simple cycle* is a cycle that is a simple path.'
14. A *directed acyclic graph (DAG)* is a directed graph with no cycles.

15. Examples where graphs are useful:
    a. Airport system: nodes are airports, edges are flights between airports
    b. Email/Packet routing on the internet: nodes are routers, edges are network links
    c. Printed circuit board design: Integrated circuits are placed on a board. *Traces* (edges) connect *pins* (nodes) which are anchored in the integrated circuits.
    d. Social networks: nodes are people, edges are friends
    e. Chemistry-molecules: nodes are atoms, edges are bonds
    f. Biology (disease spread, breeding patterns, etc.): nodes are regions where organism lives, edges are migration paths
16. We say that $|S|$ represents the size of the set S, e.g. the number of elements in the set.
17. The maximum number of edges in a directed graph is $|V|^2$, thus, $|E| \leq |V|^2$.
18. If most of the edges are present, we say the graph is *dense*. If the number of edges is $O(V)$, then we say the graph is *sparse*.

14.1.1 – Graph Representation

1. We can represent a graph with a two-dimensional array called an *adjacency matrix*. For a dense graph, this is OK, for a sparse graph, there is a lot of wasted space.

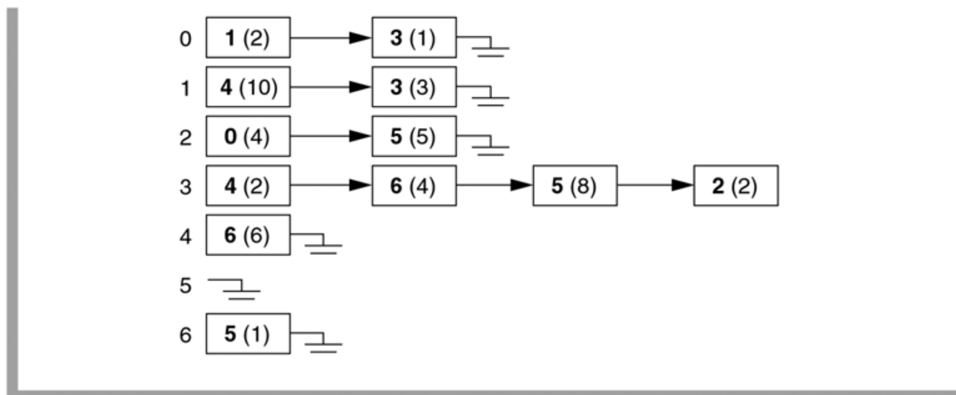


	0	1	2	3	4	5	6
0		2		1			
1				3	10		
2	4					5	
3			2		2	8	4
4							6
5							
6						1	

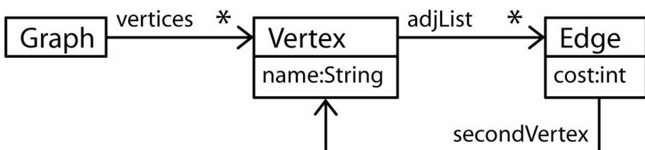
2. For a sparse graph, a better approach is an *adjacency list*. For each vertex, we keep a list of all adjacent vertices. Thus, the space requirement is approximately $O(|E|)$.

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.



3. In the figure above, vertices are duplicated (*e.g.* Vertex 3 is in both Vertex 0 and Vertex 1's adjacency lists). Of course we wouldn't want to implement the adjacency list in exactly this way; we would want to represent a vertex only once. A reasonable way to do this is to represent a Graph as a list of Vertex objects and each Vertex with a list of Edge objects, where an Edge has a cost/weight and a link to the destination (second) vertex. This is an *analysis* view of the domain:



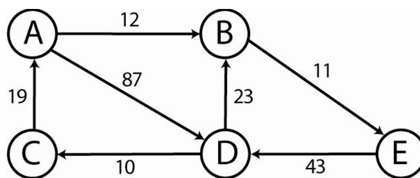
4. An efficient way to specify a graph for input is to list the edges: begin vertex, end vertex, and weight.

5. Example:

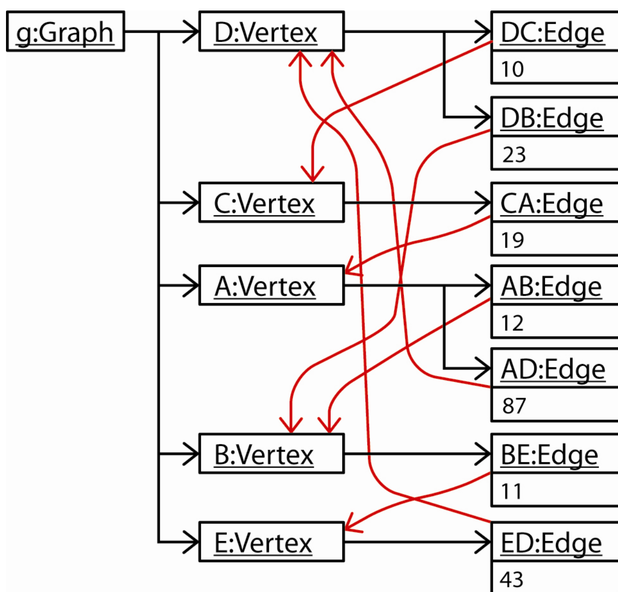
Input:

Vertex		
Begin	**End**	**Weight**
D	C	10
A	B	12
D	B	23
A	D	87
E	D	43
B	E	11
C	A	19

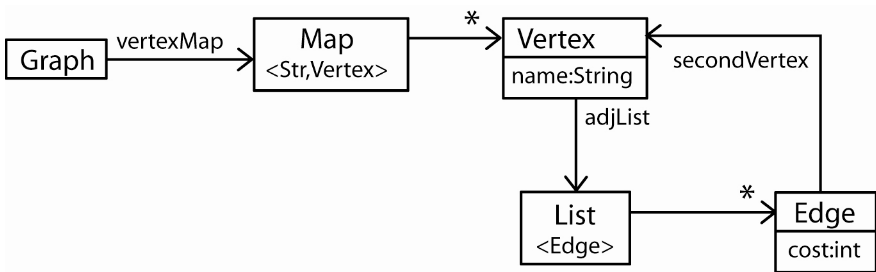
Graph:



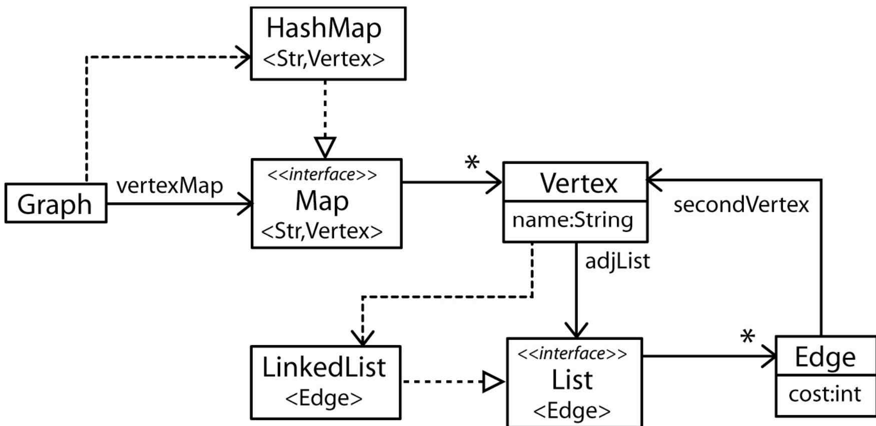
Internal Representation:



6. As we start to *design* the Graph class, we see that it will be useful to represent the collection of Vertex objects as a Map using the name of the vertex as the key and each Vertex will have a List of Edge objects.



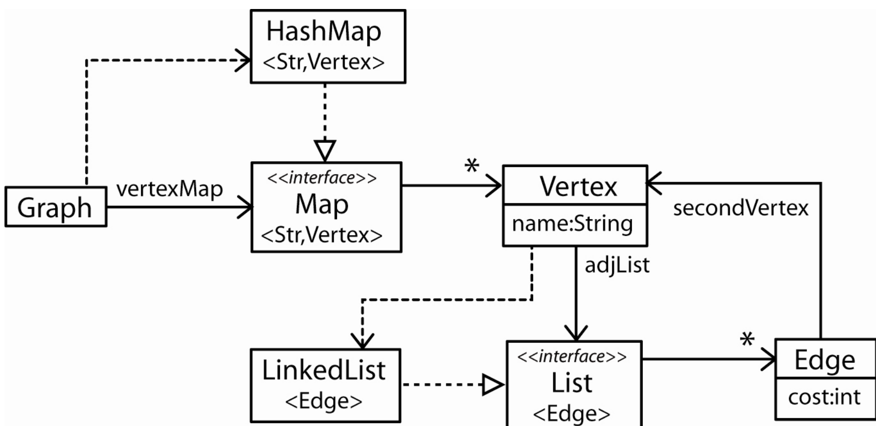
7. Next, we show the Map and List implementations.



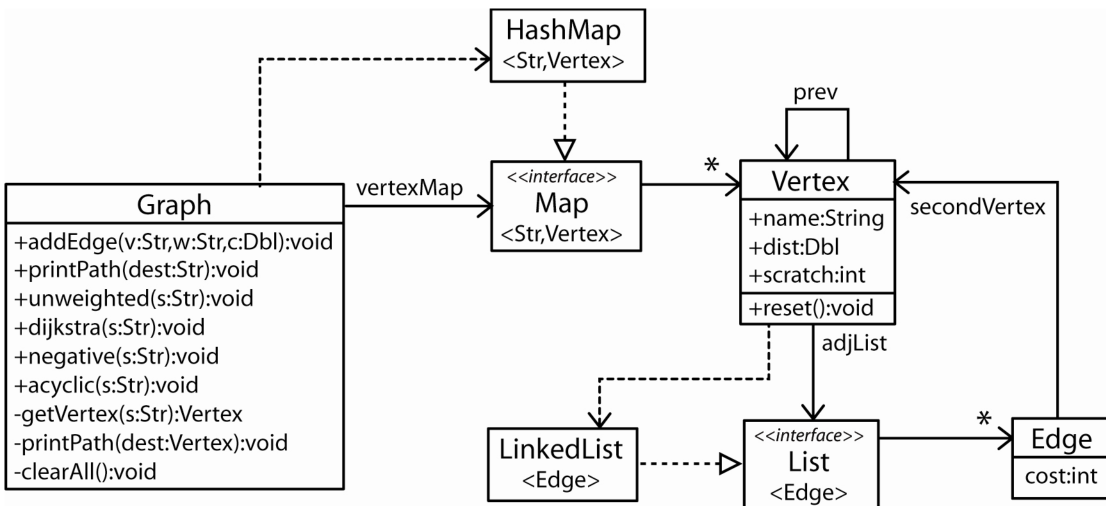

```
Map<String,Vertex> vMap = HashMap<String,Vertex>();

List<Edge> adjList = LinkedList<Edge>();
```

8. What happens when the Map and List implementations change? Notice that we have programmed to an interface, not an implementation.



9. Flesh out the design some more by adding methods and fields.



10. 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

11. Vertex class – Some of the fields will be used in the shortest path algorithms that we consider later.

```
 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

12. Graph class

```
 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 }
```

figure 14.8

The Graph class skeleton

13. getVertex Method – This method gets the Vertex with *vertexName*. If it doesn't exist, it puts it in the map. This makes the process of creating a Graph simpler, as we will see next.

```
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

14. addEdge Method – As we read the edge list we simply call the *addEdge* method, which in turn calls *getVertex* for the two vertices that form the Edge. *getVertex* automatically puts the Vertex in the map if it not already there.

```
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

15. clearAll Method

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    }
```

16. Private printPath Method

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   }
```

17. Public printPath Method

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   }
```

18. Sample Driver – Reads an edge list from a file specified from the command line. Then, calls *processRequest* to run one of four shortest path algorithms.

```
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     }
```

figure 14.14

A simple `main`

19. The *processRequest* method prompts for the start and end nodes, and the algorithm to use.

```
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

14.2 – Unweighted Shortest-Path Problem

1. Unweighted shortest-path problem – Find the shortest path (measured by number of edges) from a designated vertex S (the *source*) to every other vertex. You may only want the shortest path to one particular node (the destination); however, the algorithm we consider automatically finds the shortest path to *all* nodes.

2. Idea – Use a *roving eyeball* to visit each node. Initially, start the eyeball at S. If v is the vertex that the eyeball is on then find each vertex w that is adjacent to v and update its distance to $D_w = D_E + 1$ if its distance has not been updated before. When this is complete, move the eyeball to the next unprocessed vertex. 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 D_w is updated (the only time), it is set to the value of the length of the shortest path to w.

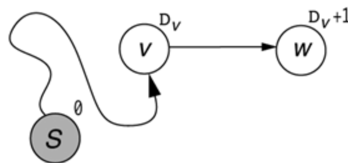
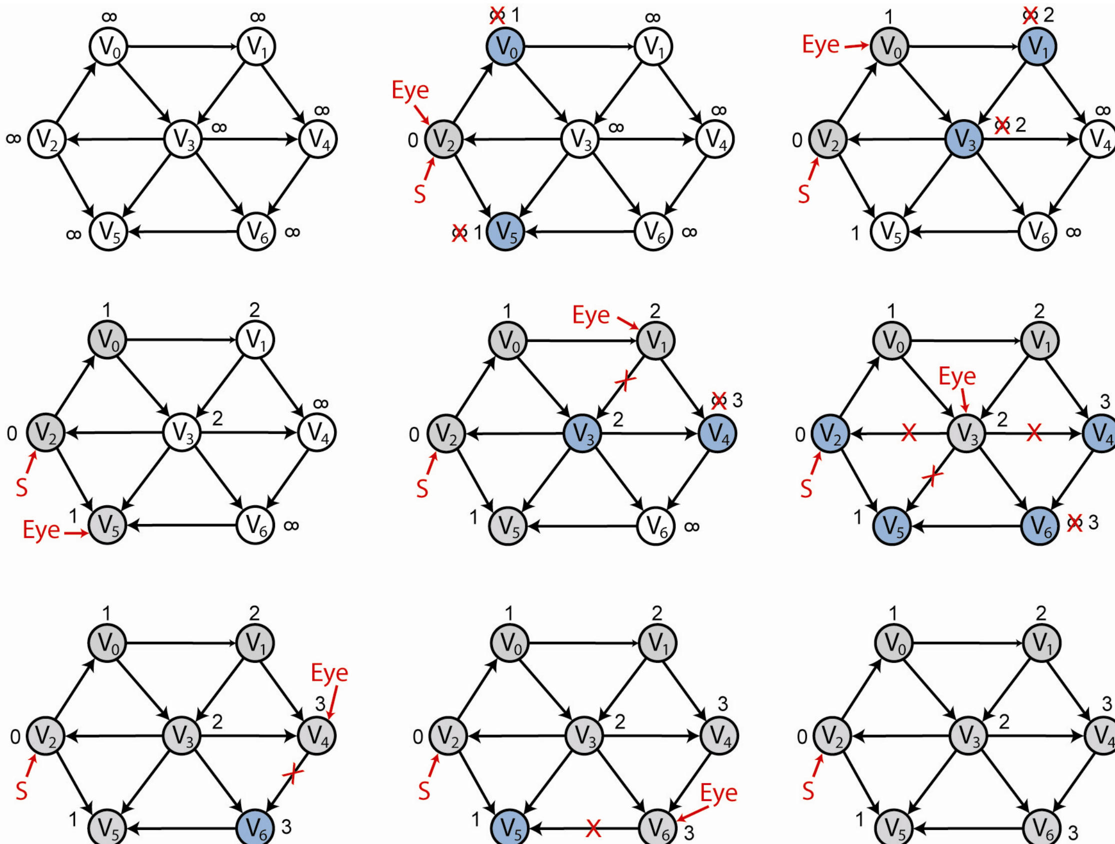


figure 14.20

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

3. Example:



4. Algorithm:

    Loop over all vertices, *u,* starting with *S*

        For each neighbor, *w* of *u*

            Update w's distance if it hasn't been updated before

        Get next u (closest to S and unvisited)

5. Algorithm

    - Let D_i be the length of the shortest path from S to i.

    Initialize $D_S = 0$, and all other nodes with, $D_i = \infty$.

    Let E be the *eyeball* that moves from node to node.

    Initialize E to S.

    Loop until all vertices visited by E:

        For each vertex w that is adjacent to E

            If $D_w = \infty$

                $D_w = D_E + 1$

        Set E to next vertex, u with $D_u = D_E$ if possible

        Otherwise, set E to next vertex, u with $D_u = D_E + 1$ if possible

        Otherwise done

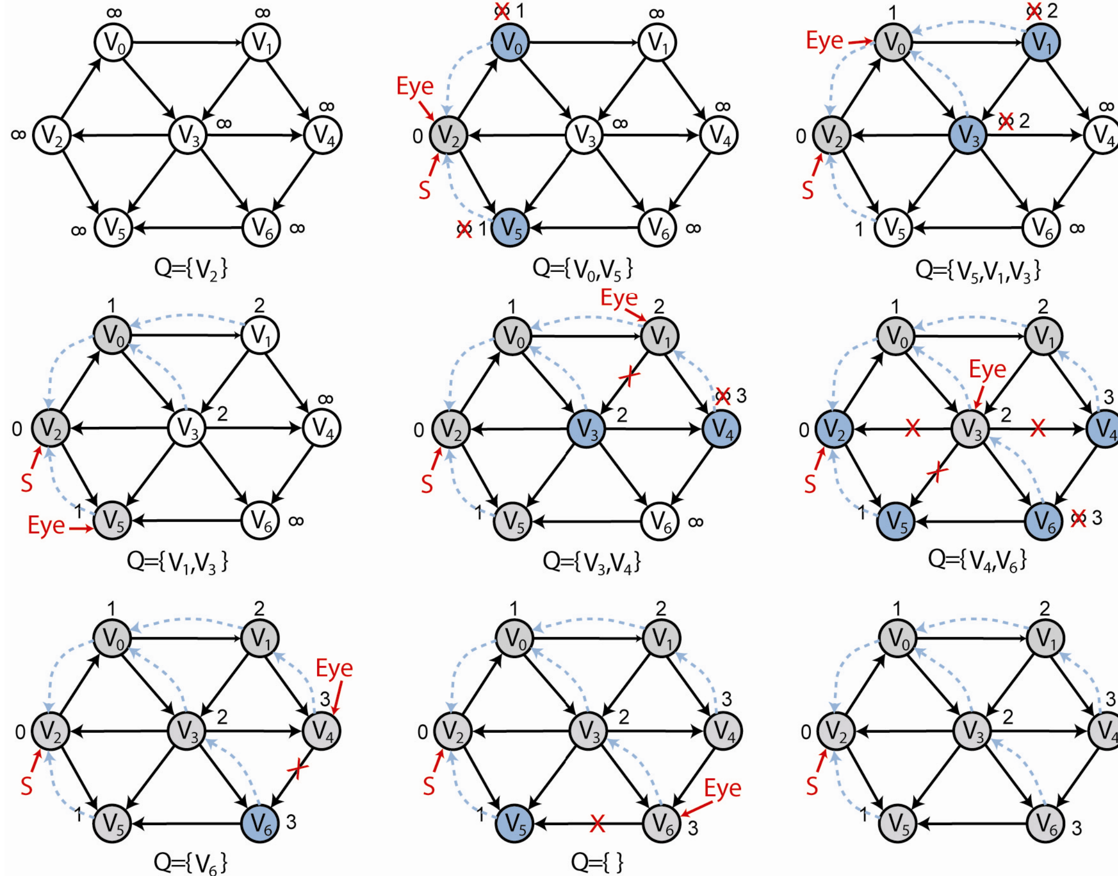
    Note: this is a *breadth-first* search. It starts at a node (the Source) and searches all its neighbors (adjacent nodes). Then, in turn it searches each neighbor's neighbors. Each search advances the *frontier* until all nodes have been processed (in which case all edges have been traversed/processed exactly once).

6. At each Eyeball position, we have to check all the w's that are adjacent to the eyeball. This is easy since all we have to do is iterate over the eyeball's adjacency list. Since each edge is only processed once, this incurs a total cost of $O(|E|)$.

    However, we must move the Eyeball to another node. We could scan through the vertex map each time which could take $O(|V|)$ time and we need to do it $|V|$ times. Thus, the total cost of this step would be $O(|V|^2)$, for a total complexity of $O(|V|^2 + |E|) = O(|V|^2)$. Fortunately, there is a better technique.

7. When a vertex w has its distance set for the first (and only) time, it becomes a candidate for an Eyeball visitation at some later time. Thus, w just has to wait its turn. We can use a queue to model this situation so that when w's distance is set, we put w at the end of the queue. To select a new vertex for the eyeball, we simply take the next vertex in the queue. Since a vertex is enqueued and dequeued at most once, and queue operations are constant, the cost of choosing the eyeball vertex is $\mathrm{O}(|\mathrm{V}|)$ for the entire algorithm. Thus, the total work for the algorithm is $\mathrm{O}(|\mathrm{V}| + |\mathrm{E}|) = \mathrm{O}(|\mathrm{E}|)$ which is dominated by $\mathrm{O}(|\mathrm{E}|)$, the processing of the adjacency lists. We say that this algorithm is linear in the size of the graph.

8. Notice that this procedure only gives us the minimum distance to each node. With a small modification, we can obtain the shortest path. We remember that the Vertex class had a *prev* field. When we update a distance on an adjacent node, we will set *prev* to the Eyeball. This is shown in the example below.



9. Java Implementation of unweighted shortest-path problem:

```
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   }
```

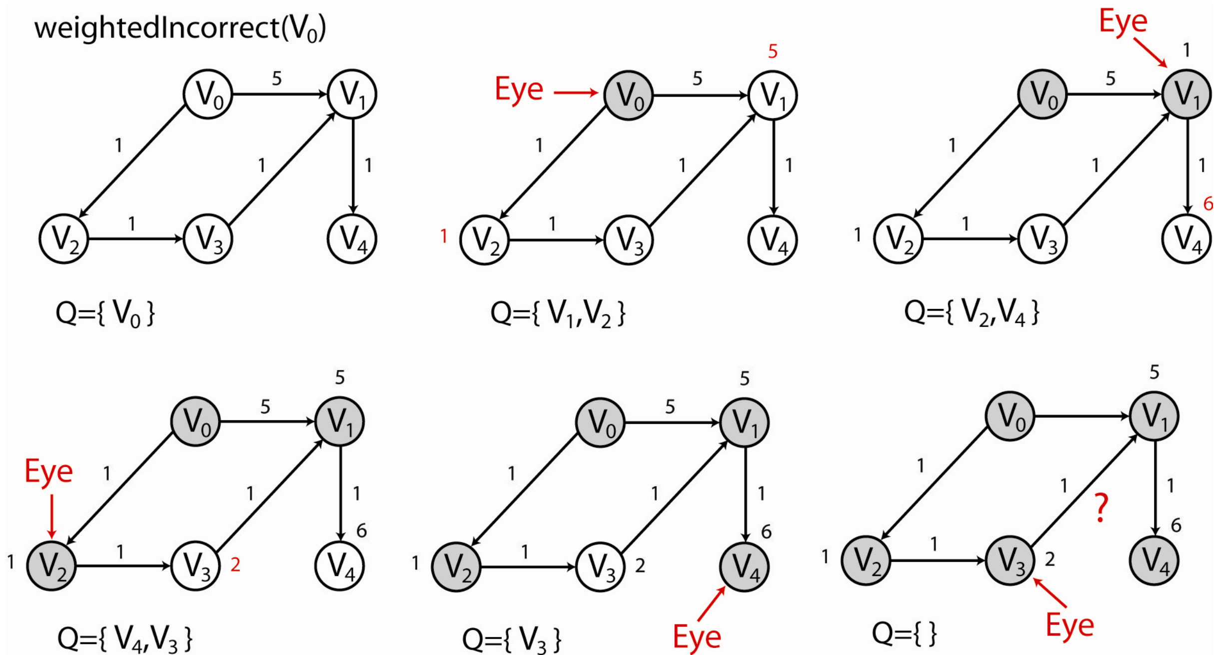
figure 14.22

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

14.3 – Positive-Weighted Shortest-Path Problem: Dijkstra's Algorithm

1. Positive-weighted shortest-path problem – Find the shortest path (measured by total cost) from a designated vertex S to every vertex. All edge costs are nonnegative.

2. Suppose we try the un-weighted algorithm:



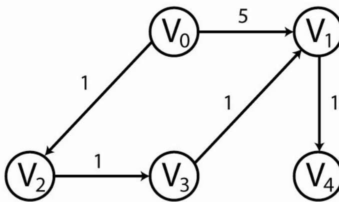
   As we see, we run into a problem. The un-weighted algorithm says that when the eye is at V_3, we should not consider updating the cost to V_1, but clearly we must. However, if we modify the algorithm and allow consideration of V_1, and thus update its cost from 5 to 3, then what happens to the cost at V_4? Its value stays at 6. It seems like this approach would require moving the eye back to vertices already visited.

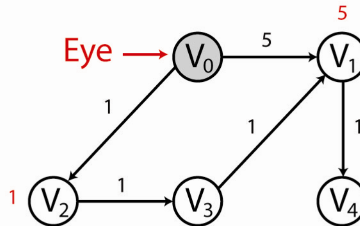
3. Fortunately, there is a better solution, often called Dijkstra's Algorithm. There are two central ideas which we will consider as modifications to the un-weighted case:

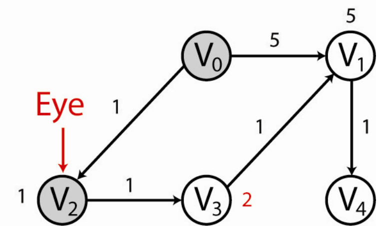
   a. Allow a node's weight to be updated more than once
   b. Move the eyeball to the next, unvisited node with the *smallest cost*. For this, we will use a priority queue. Since a node's weight can be updated, this means that the priority queue must support a *decreaseKey* operation.

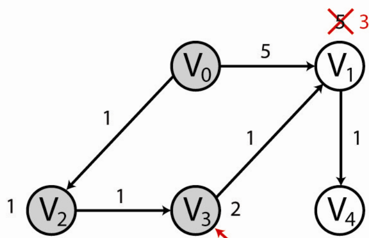
4. Example - Dijkstra's Algorithm:

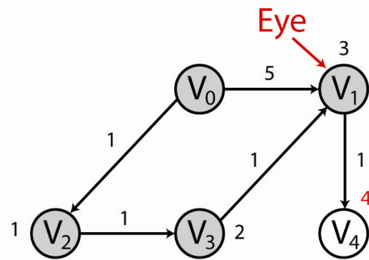
weighted(V_0)

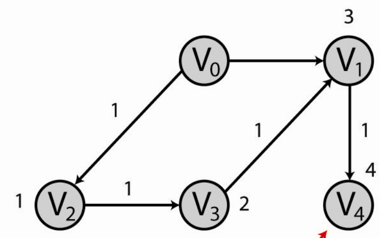


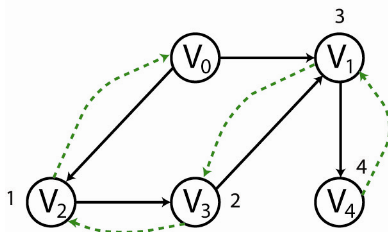












10. Let's develop an algorithm. First, let:

   1. D_i be the length of the shortest path from S to i.
   2. S be the *source* node.
   3. E be the *eyeball* that moves from node to node.

   Algorithm

   1. Initialize $D_S = 0$, and all other nodes with, $D_i = \infty$.
   2. Put S into priority queue.
   3. Loop until all priority queue is empty:
      a. E = remove min from priority queue
      b. For each vertex w that is adjacent to E

         new cost = $= D_E + c(E, w)$

         If $D_w = \infty$

            $D_w = new\ cost$

            put w in priority queue

         else if $D_w > new\ cost$

            $D_w = new\ cost$

            update priority of w

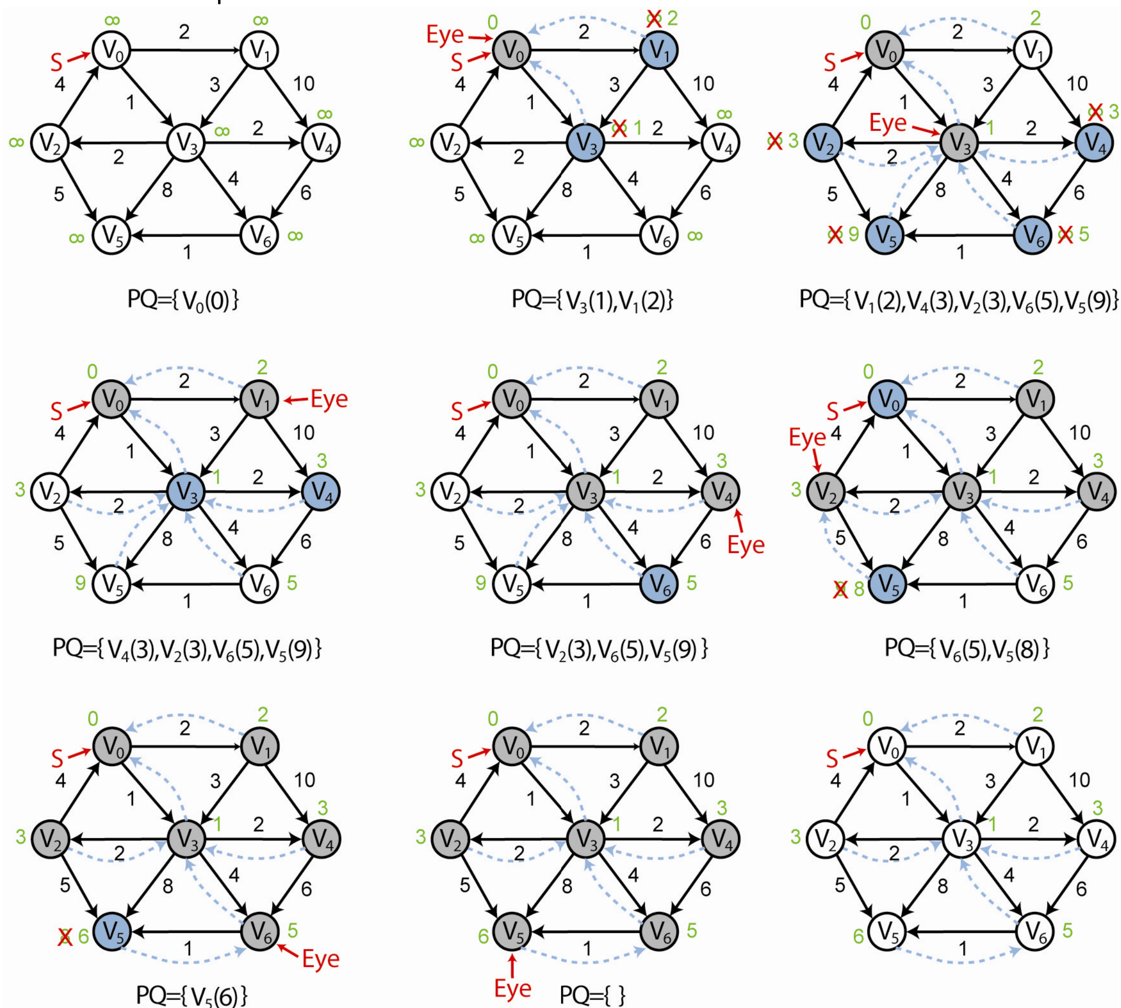
5. Now, let's look at the complexity. First, each node is added to and removed from the priority queue exactly 1 time. The dominant operation here is remove which is $O(\log|V|)$. Thus, step 3a contributes towards the total work $O(|V|\log|V|)$.

   Next, the updating of a priority (decrease key) is also $O(\log|V|)$. An upper bound on the number of times this can be called is $O(|E|)$. Thus, the total work for step 3b is $O(|E|\log|V|)$.

   Finally, the total work performed is $O(|E|\log|V|)+O(|V|\log|V|)$ which is typically dominated by $O(|E|\log|V|)$.

6. The author implements Dijkstra's algorithm with a priority queue, but not one that supports *descreaseKey*. Instead, he uses an additional field in the Vertex class, *scratch* which keeps track of whether a Vertex has been processed or not. He shows that this is also $O(|E|\log|V|)$.

7. Another example:

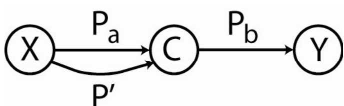


8. The crux of the proof that Dijkstra's Algorithm works is the following hypothesis: "A least-cost path from X to Y contains least-cost paths from X to every node on the path to Y."

   For example, if $X \to A \to B \to C \to Y$ is the least-cost path from X to Y, then:

   - $X \to A \to B \to C$ is the least-cost path from X to C
   - $X \to A \to B$ is the least-cost path from X to B
   - $X \to A$ is the least-cost path from X to A

   Let P be the shortest path from X to Y, which is composed of two sub-paths: P_a and P_b:



   Proof by contradiction:

   Assume hypothesis is false. Thus, given a least-cost path P from X to Y that goes through C, then there is a better path P' from X to C than P_a, the one in P.

   Show a contradiction – If P' were better than P_a, then we could replace the sub-path from X to C in P with this lesser-cost path P'. Now, since P_b doesn't change, we now have a better path from X to Y, P' and P_b. But this violates the assumption that P is the least-cost path from X to Y. Therefore, the original hypothesis must be true

9. Java Implementation

```
 1 // Represents an entry in the priority queue for Dijkstra's algorithm.
 2 class Path implements Comparable<Path>
 3 {
 4     public Vertex     dest;   // w
 5     public double     cost;   // d(w)
 6
 7     public Path( Vertex d, double c )
 8     {
 9         dest = d;
10         cost = c;
11     }
12
13     public int compareTo( Path rhs )
14     {
15         double otherCost = rhs.cost;
16
17         return cost < otherCost ? -1 : cost > otherCost ? 1 : 0;
18     }
19 }
```

figure 14.26

Basic item stored in the priority queue

```
1  /**
2   * Single-source weighted shortest-path algorithm.
3   */
4  public void dijkstra( String startName )
5  {
6      PriorityQueue<Path> pq = new PriorityQueue<Path>( );
7
8      Vertex start = vertexMap.get( startName );
9      if( start == null )
10         throw new NoSuchElementException( "Start vertex not found" );
11
12     clearAll( );
13     pq.add( new Path( start, 0 ) ); start.dist = 0;
14
15     int nodesSeen = 0;
16     while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
17     {
18         Path vrec = pq.remove( );
19         Vertex v = vrec.dest;
20         if( v.scratch != 0 )  // already processed v
21             continue;
22
23         v.scratch = 1;
24         nodesSeen++;
25
26         for( Edge e : v.adj )
27         {
28             Vertex w = e.dest;
29             double cvw = e.cost;
30
31             if( cvw < 0 )
32                 throw new GraphException( "Graph has negative edges" );
33
34             if( w.dist > v.dist + cvw )
35             {
36                 w.dist = v.dist + cvw;
37                 w.prev = v;
38                 pq.add( new Path( w, w.dist ) );
39             }
40         }
41     }
42 }
```

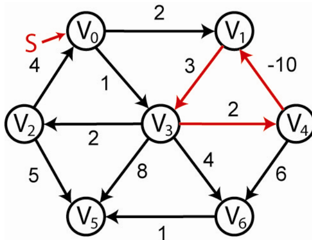
figure 14.27

A positive-weighted, shortest-path algorithm: Dijkstra's algorithm

14.3 – Negative-Weighted Shortest-Path Problem

1. As stated earlier, Dijkstra's algortihm does not work when a graph has negative weights.

2. With negative weights on a graph, we have an additional problem, that of a *negative-weight cycle*. For example, the path from V_0 to V_6 can be made arbitrarily short by continuing to cycle through the negative weight cycle, V_3, V_4, V_1. In such cases, we say the shortest-path is undefined.



3. The Bellman-Ford algorithm will solve the case with negative weights. If there is a negative-weight cycle, the algorithm will detect it and terminate. Otherwise, it will find the shortest-path. First, let:

    1. D_i be the length of the shortest path from S to i.
    2. S be the *source* node.

    Algorithm

    1. Initialize $D_S = 0$, and all other nodes with, $D_i = \infty$.
    2. For i=1 to $|V| - 1$
        a. For each edge (u, v)

            if $D_v > D_u + c(u, v)$

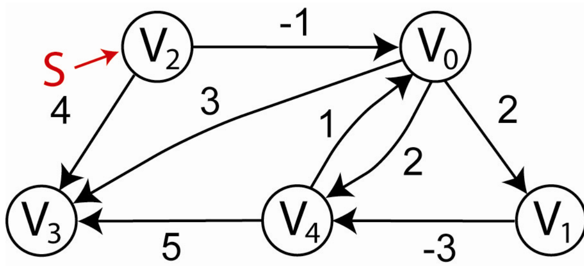
                $D_v = D_u + c(u, v)$
    3. For each edge (u, v)

        if $D_v > D_u + c(u, v)$

            negative weight cycle exists

4. Now, let's consider the complexity. We can see that step 2 dominates and that the outer loop occurs $|V| - 1$ times and the inner loop (step 2a) occurs $|E|$ times. Thus, the complexity of the algorithm is $O(|E||V|)$. Thus, it is at least quadratic in the number of vertices, which is considerably slower than Dijkstra's algorithm.

5. Example:



Distance table:

	V_0	V_1	V_2	V_3	V_4
$i = 0$	∞	∞	0	∞	∞
$i = 1$					
$i = 2$					
$i = 3$					
$i = 4$					

Summary

Type of Graph	Weights	Cycles	Running Time	Comments	Section
Unweighted	none	Yes	$O(\lvert E\rvert)$	Breadth-first search	14.2
Weighted, no negative edges	positive	Yes	$O(\lvert E\rvert \log \lvert V\rvert)$	Dijkstra's algorithm	14.3
Weighted, negative edges	any	Yes	$O(\lvert E\rvert \cdot \lvert V\rvert)$	Bellman-Ford algorithm	14.4
Weighted, acyclic	any	No	$O(\lvert E\rvert)$	Topological sort	14.5