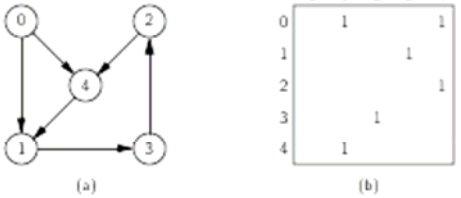
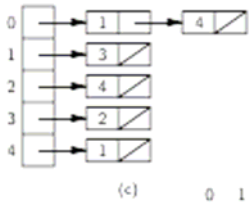
// Source code example for "A Practical Introduction to Data Structures and
// Algorithm Analysis" by Clifford A. Shaffer, Prentice Hall, 1998.
// Copyright 1998 by Clifford A. Shaffer

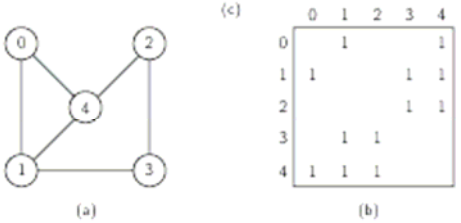
Graph Representations

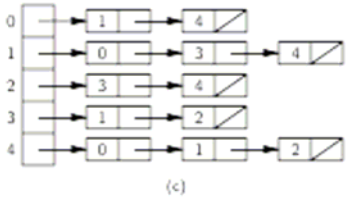
Adjacency Matrix: $\Theta(|V|^2)$.

Adjacency List: $\Theta(|V| + |E|)$.









```
// graph.java
interface Graph {                    // Graph class ADT
  public int n();                    // Number of vertices
  public int e();                    // Number of edges
  public Edge first(int v);          // Get first edge for vertex
  public Edge next(Edge w);          // Get next edge for a vertex
  public boolean isEdge(Edge w);       // True if this is an edge
  public boolean isEdge(int i, int j); // True if this is an edge
  public int v1(Edge w);               // Where edge came from
  public int v2(Edge w);               // Where edge goes to
  public void setEdge(int i, int j, int weight);// Set edge weight
  public void setEdge(Edge w, int weight);      // Set edge weight
  public void delEdge(Edge w);         // Delete edge w
  public void delEdge(int i, int j);   // Delete edge (i, j)
  public int weight(int i, int j);     // Return weight of edge
  public int weight(Edge w);           // Return weight of edge
  public void setMark(int v, int val); // Set Mark for v
  public int getMark(int v);           // Get Mark for v
} // interface Graph
```

Vertices identified by an integer i (0 <= i <= G.n())
Edges have a double nature:

Seen as pairs of vertices or
as an aggregate objects.

```
//Edge.java
interface Edge {   // Interface for graph edges
  public int v1(); // Return the vertex it comes from
  public int v2(); // Return the vertex it goes to
} // interface Edge

// Edgem.java
// Edge class for Adjacency Matrix graph representation
class Edgem implements Edge {
  private int vert1, vert2; // The vertex indices

  public Edgem(int vt1, int vt2) { vert1 = vt1; vert2 = vt2; }
  public int v1() { return vert1; }
  public int v2() { return vert2; }
} // class Edgem
```

```
//Graphm.java
class Graphm implements Graph { // Graph: Adjacency matrix
  private int[][] matrix;                // The edge matrix
  private int numEdge;                   // Number of edges
  public int[] Mark;                     // The mark array

  public Graphm(int n) {                 // Constructor
    Mark = new int[n];
    matrix = new int[n][n];
    numEdge = 0;
  }

  public int n() { return Mark.length; } // Number of vertices

  public int e() { return numEdge; }     // Number of edges

  public Edge first(int v) {         // Get the first edge
    for(int i=0; i<Mark.length; i++)// (i.e. with the lowest v2
      if (matrix[v][i] != 0)         // value) for a vertex v
        return new Edgem(v, i);
    return null;                     // No edge for this vertex
  }

  public Edge next(Edge w) { // Get next edge for a vertex
    if (w == null) return null;
    for (int i=w.v2()+1; i<Mark.length; i++)
      if (matrix[w.v1()][i] != 0)
        return new Edgem(w.v1(), i);
    return null;             // No next edge;
  }

  public boolean isEdge(Edge w) { // True if this is an edge
    if (w == null) return false;
    else return matrix[w.v1()][w.v2()] != 0;
  }

  public boolean isEdge(int i, int j) // True if this is an edge
    { return matrix[i][j] != 0; }

  public int v1(Edge w){ return w.v1(); } // Where edge comes from

  public int v2(Edge w){ return w.v2(); } // Where edge goes to
```

```
  public void setEdge(int i, int j, int wt) { // Set edge weight
    matrix[i][j] = wt;
    numEdge++;
  }
  public void setEdge(Edge w, int weight) // Set edge weight
  { if (w != null) setEdge(w.v1(), w.v2(), weight); }



  public void delEdge(Edge w) {       // Delete edge w
    if (w != null)
      if (matrix[w.v1()][w.v2()] != 0) {
        matrix[w.v1()][w.v2()] = 0;
        numEdge--;
      }
  }
  public void delEdge(int i, int j) { // Delete edge (i, j)
    if (matrix[i][j] != 0) {
      matrix[i][j] = 0;
      numEdge--;
    }
  }

  public int weight(int i, int j) {  // Return weight of edge
    if (matrix[i][j] == 0) return Integer.MAX_VALUE;
    else return matrix[i][j];
  }

  public int weight(Edge w) {        // Return weight of edge
    if (matrix[w.v1()][w.v2()] == 0) return Integer.MAX_VALUE;
    else return matrix[w.v1()][w.v2()];
  }

  public void setMark(int v, int val){ Mark[v]=val; } // Set Mark

  public int getMark(int v) { return Mark[v]; }       // Get Mark

} // class Graphm
```

Graph Traversals

Some applications require visiting every vertex in the graph exactly once. Application may require that vertices be visited in some special order based on graph topology.

To insure visiting all the vertices of the connected components of the graph:

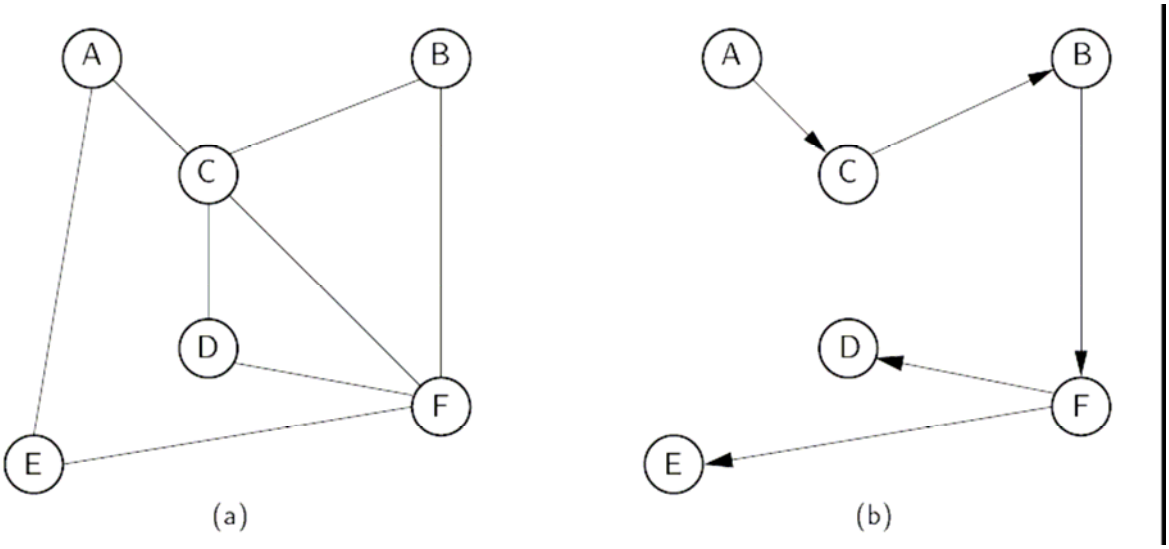
```
void graphTraverse(Graph G) {
    for (v=0; v<G.n(); v++)
       G.setMark(v, UNVISITED); // Initialize mark bits
    for (v=0; v<G.n(); v++)
       if (G.getMark(v) == UNVISITED)
         doTraverse(G, v);    // see below … DFS or BFS
}
```

Depth First Search (DFS)

Whenever a vertex v is visited during the search, recursively visit all of its unvisited neighbours.

The process creates a **depth-first search tree**. This tree is composed of edges that were followed to any new, unvisited, vertex during the traversal; and leaves out the edges that lead to already visited vertices.

```
// Depth first search (starting from vertex v)
static void DFS(Graph G, int v) {
           PreVisit(G, v);             // Take appropriate action
           G.setMark(v, VISITED);
           for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
              if (G.getMark(G.v2(w)) == UNVISITED)
                DFS(G, G.v2(w));
           PostVisit(G, v);            // Take appropriate action
}
```

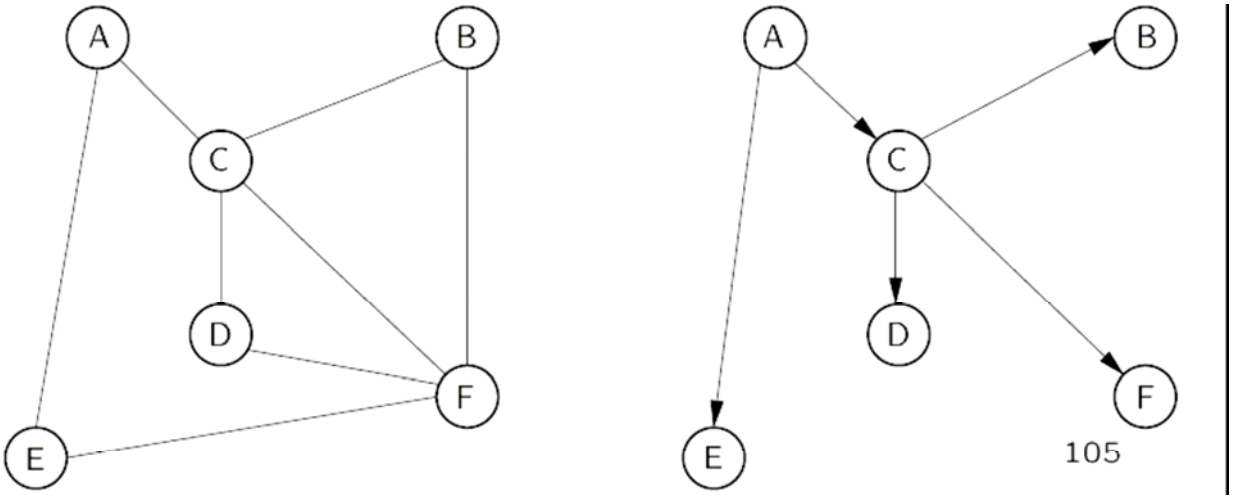


Breadth First Search (BFS)

Like DFS, but replace stack with a queue.
Visit the vertex's neighbours before continuing deeper in the tree.
BFS is implemented similarly to DFS, except that a queue replaces the recursion stack.

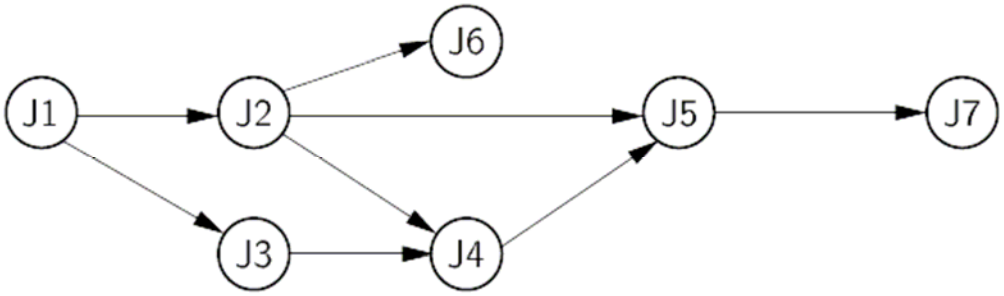
```
static void BFS(Graph G, int start) {
      Queue Q = new AQueue(G.n());    // Use a Queue
      Q.enqueue(new Integer(start));
      G.setMark(start, VISITED);
      while (!Q.isEmpty()) {          // Process each vertex on Q
          int v = ((Integer)Q.dequeue()).intValue();

          PreVisit(G, v);             // Take appropriate action
          for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
             if (G.getMark(G.v2(w)) == UNVISITED) {
               G.setMark(G.v2(w), VISITED);
               Q.enqueue(new Integer(G.v2(w)));
             }
          PostVisit(G, v);            // Take appropriate action
      }
}
```

Starting at A and visiting adjacent neighbours in lexicographic order:



Topological Sort

Problem: Given a set of jobs, courses, etc...with prerequisite constraints, output the jobs in a linear order that allow us to complete them one at time without violating any prerequisites.
The problem is modelled using a Directed Acyclic Graph (DAG)



Example of acceptable sort: J1, J2, J3, J4, J5, J6, J7

One kind of Topological sort is obtained visiting the graph by means of a DFS with a post order manipulation and subsequently reversing the list obtained.

Starting at J1 and visiting adjacent neighbours in lexicographic order:
Reverse(J7, J5, J4, J6, J2, J3, J1) = J1, J3, J2, J6, J4, J5, J7

```
static void topsort(Graph G) {  // Topological sort: recursive
    for (int i=0; i<G.n(); i++) // Initialize Mark array
        G.setMark(i, UNVISITED);
    for (int i=0; i<G.n(); i++)//Process all vertices for all
                               //connected components of the graph
       if (G.getMark(i) == UNVISITED)
          tophelp(G, i);        // Call helper function
}

static void tophelp(Graph G, int v) { // Topsort helper
     G.setMark(v, VISITED);
     for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (G.getMark(G.v2(w)) == UNVISITED)
          tophelp(G, G.v2(w));
     printout(v); // PostVisit for Vertex v
}
```

A Queue-based Topological Sort:

- first visit all the edges counting the number of edges that leads to each vertex.
- All vertices without incoming edges are placed on the queue.
- While the queue is not empty
  1. Take off a vertex from the queue and print it
  2. Decrease the counts of all its neighbours
     - Each vertex with zero incoming edge is placed on the queue

```
static void topsort(Graph G) {          // Topological sort: Queue

     Queue Q = new AQueue(G.n());
     int[] Count = new int[G.n()];
     int v;

     for (v=0; v<G.n(); v++)              // Initialize
        Count[v] = 0;
     for (v=0; v<G.n(); v++)              // Process every edge
        for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
          Count[G.v2(w)]++;               // Add to v2's count

     for (v=0; v<G.n(); v++)              // Initialize Queue
        if (Count[v] == 0)                // Vertex has no prereqs
          Q.enqueue(new Integer(v));

     while (!Q.isEmpty()) {               // Process the vertices
          v = ((Integer)Q.dequeue()).intValue();
          printout(v);                    // PreVisit for Vertex V
          for (Edge w=G.first(v); G.isEdge(w); w=G.next(w)) {
             Count[G.v2(w)]--;            // One less prerequisite
             if (Count[G.v2(w)] == 0)     // This vertex now free
               Q.enqueue(new Integer(G.v2(w)));
          }
     }
}
```

Shortest Paths Problems

Input: A graph with weights or costs associated with each edge.
Output: The list of edges forming the shortest path.
Problems:

- Find the shortest path between two specified vertices.
- Find the shortest path from vertex S to all other vertices (is the worst case of the previous problem)
- Find the shortest path between all pairs of vertices.

(For sake of simplicity, our algorithms will actually calculate only distance rather than recording the actual path.)

d(A, B) is the shortest distance from vertex A to B.
w(A, B) is the weight of the edge connecting A to B.
-- If there is no such edge, then w(A, B) = INFINITY.

Single Source Shortest Paths

Given start vertex s, find the shortest path from s to all other vertices.

Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex x.
Problem: The true Shortest path to a vertex already processed might go through x.
Solution: Process vertices in order of distance from s.

Dijkstra Algorithm

Maintain a distance estimate for all vertices in the graph: D[v].
Assume that we have processed in order of distance the first i-1 vertices closest to ; call it S. When processing the i-th closest vertex x, the shortest path from s to x must have its next-to-last vertex in S; Thus:

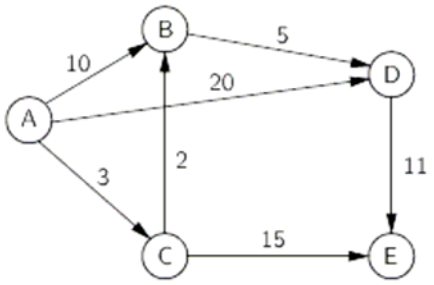
$$d(s,x) = \min(d(s,u)+w(u,x)) \text{ for all } u \text{ in } S$$

The shortest path from s to x is the minimum over all paths that go from s to u, then have an edge from u to x, where u belongs to S.
Whenever a vertex x is processed, D[v] is updated for every neighbour v of x.

```
// Dijkstra's Algorithm: Array
// Compute shortest path distances from s, store in D
static void Dijkstra(Graph G, int s, int[] D) {
    for (int i=0; i<G.n(); i++)          // Initialize
       D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i=0; i<G.n(); i++) {      // Process vertices
       int v = minVertex(G, D);        // Get next-closest vertex
       G.setMark(v, VISITED);
       if (D[v] == Integer.MAX_VALUE) return;
       for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
          if (D[G.v2(w)] > (D[v] + G.weight(w)))
            D[G.v2(w)] = D[v] + G.weight(w);
    }
}
```

Dijkstra's Algorithm Example

	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18



Issue: How to determine the next-closest vertex? (I.e., implement minVertex)
Approach 1: Scan through the table of current distances.

- Cost: Q($|\mathbf{V}|^2 + |\mathbf{E}|$) = Q($|\mathbf{V}|^2$)

```
static int minVertex(Graph G, int[] D) {
     int v = 0;          // Initialize v to any unvisited vertex;
     for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED) { v = i; break; }
     for (int i=0; i<G.n(); i++)           // Find smallest value
        if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
           v = i;
     return v;
}
```

Approach 2:
Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value.
Must update priority queue for each edge.
– Cost: Q((|**V**| + |**E**|) log|**V**|)

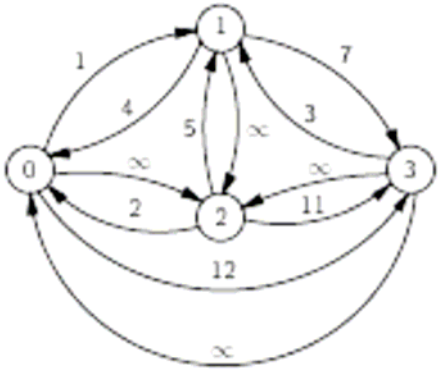
We need to store on the heap some object that implements Elem interface (i.e. with a key method) and stores a vertex and its current distance from the start vertex. The key method returns the distance from the start vertex.

```
interface Elem {             // Interface for generic element type
  public abstract int key(); // Key used for search and ordering
} // interface Elem

class DijkElem implements Elem {
  private int vertex;
  private int distance;

  public DijkElem(int v, int d) { vertex = v; distance = d; }
  public DijkElem() {vertex = 0; distance = 0; }

  public int key() { return distance; }
  public int vertex() { return vertex; }
} // class DikjElem

// Dijkstra's shortest-paths algorithm:
// priority queue version
static void Dijkstra(Graph G, int s, int[] D) {
      int v;                                 // The current vertex
       DijkElem[] E = new DijkElem[G.e()];   // Heap: array of edges
       E[0] = new DijkElem(s, 0);            // Initialize array
       MinHeap H = new MinHeap(E, 1, G.e()); // Create heap from array E
                                             // with currently 1 element
                                             // out of G.e()
       for (int i=0; i<G.n(); i++)           // Initialize distances
          D[i] = Integer.MAX_VALUE;

       D[s] = 0;
       for (int i=0; i<G.n(); i++) {         // For each vertex
           do {
                v = ((DijkElem)H.removemin()).vertex();
           } while (G.getMark(v) == VISITED);
           /// Now we have Extracted the minimum UNVISITED vertex v
           G.setMark(v, VISITED);
           if (D[v] == Integer.MAX_VALUE) return;
           for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
               if   (D[G.v2(w)] > (D[v] + G.weight(w))) { // Update D
                   D[G.v2(w)] = D[v] + G.weight(w);
                   H.insert(new DijkElem(G.v2(w), D[G.v2(w)]));
               }
       }
}
```

All Pairs Shortest Paths
For every vertex (u; v), calculate d(u, v).
Could run Dijkstra's Algorithm V times.

Better is Floyd's Algorithm.
[The issue is how to efficiently check all the paths without computing any path more than once.]
Define a k-path from u to v to be any path whose intermediate vertices all have indices less than k (aside u and v).

The shortest path between u, v will be an n-path.
0-path between u,v is weight of edge[u, v]



Path 3-0-2 is a 1-path; Path 1-3-2 is a 4-path

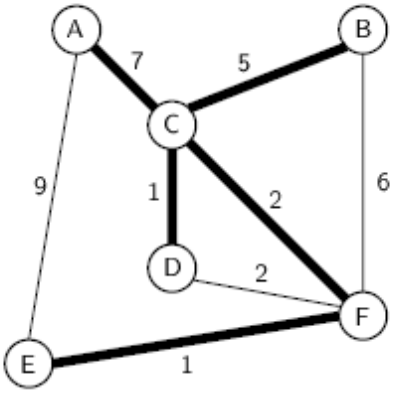
If $D_k[u,v]$ is the shortest k-path between u and v then
The shortest k+1 path between u and v
either goes through vertex k or it does not.
If it goes through vertex k, then such a path is constructed as the shortest k-path between u and k followed by the shortest k-path between k and v. If it does not go through vertex k, then $D_{k+1}[u,v] = D_k[u,v]$.

```
// Floyd's Algorithm: Compute all-pairs shortest paths in matrix D
static void Floyd(Graph G, int[][] D) {
     for (int i=0; i<G.n(); i++)   // Initialize D with 0-paths
        for (int j=0; j<G.n(); j++)// amid all couples of vertices
           D[i][j] = G.weight(i, j);

     for (int k=0; k<G.n(); k++) // Compute all k paths
        for (int i=0; i<G.n(); i++)
           for (int j=0; j<G.n(); j++)
              if ( (D[i][k] != Integer.MAX_VALUE) &&
                   (D[k][j] != Integer.MAX_VALUE) &&
                   (D[i][j] > (D[i][k] + D[k][j])) )
                 D[i][j] = D[i][k] + D[k][j];
}
```

Minimum Cost Spanning Trees

Minimum Cost Spanning Tree (MST) Problem:

- Input: An undirected, connected graph G.
- Output: The subgraph of G that
  - 1) has minimum total cost as measured by summing the values for all of the edges in the subset, and
  - 2) keeps the vertices connected.



Prim's MST Algorithm

This is an example of a greedy algorithm.

Starting from any vertex N in the graph; Add it to the MST;

Search for the minimum edge that comes from N to a vertex out of the MST, call it M
Insert M and the selected edge e_1 to the MST

Search for the minimum edge that comes from N or M to a vertex out of the MST,call it P
Insert P and the selected edge e_2 to the MST

Repeat the same steps until all the vertex in the graph have been visited.

```
// Compute a minimal-cost spanning tree
// D[i], respect to vertex i, stores the distance from the closest
// vertex in the MST
static void Prim(Graph G, int s, int[] D) {
       int[] V = new int[G.n()];  // V[i] closest MST vertex to i
       for (int i=0; i<G.n(); i++) // Initialize
          D[i] = Integer.MAX_VALUE;
       D[s] = 0;
       for (int i=0; i<G.n(); i++) {      // Process vertices
          int v = minVertex(G, D);
          G.setMark(v, VISITED);
          if (v != s)
             System.out.println("Add Edge " + V[v] + ", " + v);
          if (D[v] == Integer.MAX_VALUE) return;
          for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
             if (D[G.v2(w)] > G.weight(w)) {
               D[G.v2(w)] = G.weight(w);
               V[G.v2(w)] = v;
             }
          }
}
```