Da http://rosettacode.org/wiki/Dijkstra%27s_algorithm#Java pagina aggiornata 26 dicembre 2018

An example, starting with (Matrice o lista di adiacenze con un nodo di partenza)

```
          a→b, cost=7,  lastNode=a
                                        a→c, cost=9,  lastNode=a
                                        a→d, cost=NA, lastNode=a
                                        a→e, cost=NA, lastNode=a
                                        a→f, cost=14, lastNode=a

    The lowest cost is    a→b    so    a→b    is added to the output.

    There is a connection from   b→d   so the input is updated to:
                                        a→c, cost=9,  lastNode=a
                                        a→d, cost=22, lastNode=b
                                        a→e, cost=NA, lastNode=a
                                        a→f, cost=14, lastNode=a

    The lowest cost is    a→c    so    a→c    is added to the output.

    Paths to    d    and    f    are cheaper via    c    so the input is updated
to:
                                        a→d, cost=20, lastNode=c
                                        a→e, cost=NA, lastNode=a
                                        a→f, cost=11, lastNode=c

    The lowest cost is    a→f    so    c→f    is added to the output.

    The input is updated to:
                                        a→d, cost=20, lastNode=c
                                        a→e, cost=NA, lastNode=a

    The lowest cost is    a→d    so    c→d    is added to the output.

    There is a connection from    d→e    so the input is updated to:
                                        a→e, cost=26, lastNode=d

    Which just leaves adding    d→e    to the output.

    The output should now be:
                                      [ d→e
                                        c→d
                                        c→f
                                        a→c
                                        a→b ]
```

Task

1. Implement a version of Dijkstra's algorithm that outputs a set of edges depicting the shortest path to each reachable node from an origin.
2. Run your program with the following directed graph starting at node **a**.
3. Write a program which interprets the output from the above and use it to output the shortest path from node **a** to nodes **e** and **f** .

Vertices

Number	Name
1	a
2	b
3	c
4	d
5	e
6	f

Edges

Start	End	Cost
a	b	7
a	c	9
a	f	14
b	c	10
b	d	15
c	d	11
c	f	2
d	e	6
e	f	9

You can use numbers or names to identify vertices in your program.

See also

- Dijkstra's Algorithm vs. A* Search vs. Concurrent Dijkstra's Algorithm (youtube)

JAVA 2018

Algorithm is derived from Wikipedia section 'Using a priority queue'. This implementation finds the single path from a source to all reachable vertices. Building the graph from a set of edges takes O(E log V) for each pass. Vertices are stored in a **TreeSet** (self-balancing binary search tree) instead of a PriorityQueue (a binary heap) in order to get O(log n) performance for removal of any element, not just the head. Decreasing the distance of a vertex is accomplished by removing it from the tree and later re-inserting it.

```
import java.io.*;
import java.util.*; // uso Map<> e TreeSet<> della piattaforma  JCF

public class Dijkstra {
   private static final Graph.Edge[] GRAPH = {
      new Graph.Edge("a", "b", 7),
      new Graph.Edge("a", "c", 9),
      new Graph.Edge("a", "f", 14),
      new Graph.Edge("b", "c", 10),
      new Graph.Edge("b", "d", 15),
      new Graph.Edge("c", "d", 11),
      new Graph.Edge("c", "f", 2),
      new Graph.Edge("d", "e", 6),
      new Graph.Edge("e", "f", 9),
   };
   private static final String START = "a";
   private static final String END = "e";

   public static void main(String[] args) {
      Graph g = new Graph(GRAPH);
      g.dijkstra(START);
      g.printPath(END);
      //g.printAllPaths();
   }
}

class Graph {
   private final Map<String, Vertex> graph; // mapping of vertex names to Vertex
                                            // objects, built from a set of Edges

   /** One edge of the graph (only used by Graph constructor) */
   public static class Edge {
      public final String v1, v2;
      public final int dist;
      public Edge(String v1, String v2, int dist) {
         this.v1 = v1;
         this.v2 = v2;
         this.dist = dist;
      }
   }

  /** One vertex of the graph, complete with mappings
       to neighbouring vertices */
  public static class Vertex implements Comparable<Vertex>{
       public final String name;
       public int dist = Integer.MAX_VALUE; // MAX_VALUE assumed to be infinity
       public Vertex previous = null;
       public final Map<Vertex, Integer> neighbours = new HashMap<>();
       public Vertex(String name)
        {
                this.name = name;
        }
```

```java
        private void printPath()
        {
                if (this == this.previous)
                {
                        System.out.printf("%s", this.name);
                }

                else if (this.previous == null)
                {
                        System.out.printf("%s(unreached)", this.name);
                }
                else
                {
                        this.previous.printPath();
                        System.out.printf(" -> %s(%d)", this.name, this.dist);
                }
        }

        public int compareTo(Vertex other)
        {
                if (dist == other.dist)
                        return name.compareTo(other.name);

                return Integer.compare(dist, other.dist);
        }

        @Override public String toString()
        {
                return "(" + name + ", " + dist + ")";
        }
    }

    /** Builds a graph from a set of edges */
    public Graph(Edge[] edges) {
       graph = new HashMap<>(edges.length);

       //one pass to find all vertices
       for (Edge e : edges) {
          if (!graph.containsKey(e.v1)) graph.put(e.v1, new Vertex(e.v1));
          if (!graph.containsKey(e.v2)) graph.put(e.v2, new Vertex(e.v2));
       }

       //another pass to set neighbouring vertices
       for (Edge e : edges) {
          graph.get(e.v1).neighbours.put(graph.get(e.v2), e.dist);
          //graph.get(e.v2).neighbours.put(graph.get(e.v1), e.dist);
          // also do this for an undirected graph
       }
    }

    /** Runs dijkstra using a specified source vertex */
    public void dijkstra(String startName) {
       if (!graph.containsKey(startName)) {
          System.err.printf("Graph doesn't contain start vertex \"%s\"\n",
                            startName);
          return;
       }
       final Vertex source = graph.get(startName);
       NavigableSet<Vertex> q = new TreeSet<>();
```

```java
        // set-up vertices
        for (Vertex v : graph.values()) {
            v.previous = v == source ? source : null;
            v.dist = v == source ? 0 : Integer.MAX_VALUE;
            q.add(v);
        }

        dijkstra(q);
    }

    /** Implementation of dijkstra's algorithm using a binary heap. */
    private void dijkstra(final NavigableSet<Vertex> q) {
        Vertex u, v;
        while (!q.isEmpty()) {

            u = q.pollFirst(); // vertex with shortest distance
                               // (first iteration will return source)
            if (u.dist == Integer.MAX_VALUE) break; // we can ignore u
            //(and any other remaining vertices) since they are unreachable

            //look at distances to each neighbour
            for (Map.Entry<Vertex, Integer> a : u.neighbours.entrySet()) {
                v = a.getKey(); //the neighbour in this iteration

                final int alternateDist = u.dist + a.getValue();
                if (alternateDist < v.dist) { // shorter path to neighbour found
                    q.remove(v);
                    v.dist = alternateDist;
                    v.previous = u;
                    q.add(v);
                }
            }
        }
    }

    /** Prints a path from the source to the specified vertex */
    public void printPath(String endName) {
        if (!graph.containsKey(endName)) {
            System.err.printf("Graph doesn't contain end vertex \"%s\"\n",endName);
            return;
        }

        graph.get(endName).printPath();
        System.out.println();
    }
    /** Prints the path from the source to every vertex (
         output order is not guaranteed) */
    public void printAllPaths() {
        for (Vertex v : graph.values()) {
            v.printPath();
            System.out.println();
        }
    }
}
```

Output:

```
a -> c(9) -> d(20) -> e(26)
```

Altro esempio, slides, con animazione, perché studiare questo algoritmo ?