import java.util.\*;

import java.util.Map.Entry;

//--- Edge class ------------------------------------------------------

class Edge<E> implements Comparable< Edge<E> >

{

Vertex<E> source, dest;

double cost;

Edge( Vertex<E> src, Vertex<E> dst, Double cst)

{

source = src;

dest = dst;

cost = cst;

}

Edge( Vertex<E> src, Vertex<E> dst, Integer cst)

{

this (src, dst, cst.doubleValue());

}

Edge()

{

this(null, null, 1.);

}

public String toString(){

return "Edge: " + source.getData()

+ " to " + dest.getData()

+ ", distance: " + cost;

}

public int compareTo( Edge<E> rhs )

{

return

(cost < rhs.cost? -1 : cost > rhs.cost? 1 : 0);

}

}

public class Kruskal<E> extends Graph<E>

{

// will add Edges from largest to smallest cost

private PriorityQueue< Edge<E> > edgeHeap;

public Kruskal ()

{

edgeHeap = new PriorityQueue< Edge<E> >();

}

public void clear()

{

edgeHeap.clear();

}

// algorithms

public ArrayList< Edge<E> > applyKruskal()

{

Iterator<Entry<E, Vertex<E>>> iter;

LinkedList< HashSet<Vertex<E>> > vertexSets

= new LinkedList< HashSet<Vertex<E>> >();

Iterator< HashSet<Vertex<E>> > fIter;

HashMap<E, Vertex<E>> vertsInGraph;

HashSet<Vertex<E>> singleton, vertSet,

vertSetSrc = null, vertSetDst = null;

Edge<E> smallestEdge;

Vertex<E> src, dst, vert;

ArrayList< Edge<E> > newEdges

= new ArrayList< Edge<E> >();

int k, numVertsFound;

// form a forest of sets, initializing each

// with only one vertex from the graph

vertsInGraph = vertexSet;

// refer to Superclass' vertex set

for (k = 0,

iter = vertsInGraph.entrySet().iterator();

iter.hasNext(); k++)

{

vert = iter.next().getValue();

singleton = new HashSet<Vertex<E>>();

singleton.add(vert);

vertexSets.add( singleton );

}

// form a binary min heap of edges

// so we can easily find min costs

if (!buildEdgeHeap())

return null;

// test for empty to avoid inf. loop

// resulting from disconnected graph

while (!edgeHeap.isEmpty() && vertexSets.size() > 1)

{

// pop smallest edge left in heap

smallestEdge = edgeHeap.remove();

src = smallestEdge.source;

dst = smallestEdge.dest;

// see if src and dst are in different sets. // if so, take union

for (fIter = vertexSets.iterator(),

numVertsFound = 0 ;

fIter.hasNext() && (numVertsFound < 2) ; )

{

vertSet = fIter.next();

if ( vertSet.contains(src) )

{

vertSetSrc = vertSet;

numVertsFound++;

}

if ( vertSet.contains(dst) )

{

vertSetDst = vertSet;

numVertsFound++;

}

}

if (vertSetSrc == vertSetDst)

// same sets: reject

continue;

newEdges.add(smallestEdge);

vertSetSrc.addAll(vertSetDst);

vertexSets.remove(vertSetDst);

}

return newEdges;

}

private boolean buildEdgeHeap()

{

HashMap<E, Vertex<E>> vertsInGraph;

Iterator<Entry<E, Vertex<E>>> vertIter;

Iterator<Entry<E, Pair<Vertex<E>, Double>>> edgeIter;

Vertex<E> src, dst;

Pair<Vertex<E>, Double> edge;

double cost;

if (vertexSet.isEmpty())

return false;

vertsInGraph = vertexSet;

for (vertIter = vertsInGraph.entrySet().iterator();

vertIter.hasNext(); )

{

src = vertIter.next().getValue();

for (edgeIter =

src.adjList.entrySet().iterator();

edgeIter.hasNext(); )

{

edge = edgeIter.next().getValue();

dst = edge.first;

cost = edge.second;

edgeHeap.add( new Edge<E>(src, dst, cost) );

}

}

return true;

}

}