```
package graphAlg;

import graphAlg.Indexed;

import java.util.List;
import java.util.ArrayList;

public class Partition<X extends Indexed> {

    /* Abstractly a partition is a set of disjoint sets
     *   P : { PS : SET OF SET OF X |
     *         for all s,t : PS @ s <> t => s intersect t = {} }
     * The partition is represented by a forest of trees
     * which are represented using parent links.
     *   P = { extractSet(entry.element) | entry : entries && entry != null }
     * where
     *   extractSet(x) = { e.element | e : entries && e != null &&
     *                     findSet(e.element) == findSet(x) }
     */
    private List<Entry> entries;

    private class Entry {
        X element;
        Entry parent;
        int rank;

        private Entry(X element) {
            this.element = element;
            // Parent is self loop to indicate it is the root
            this.parent = this;
            // Maximum height of the tree rooted at this node
            this.rank = 0;
        }
    }


    /* P = {} */
    public Partition( int maxSize ) {
        entries = new ArrayList<Entry>( maxSize );
        for( int i = 0; i < maxSize; i++ ) {
            entries.add( null );
        }
    }

    /** @requires !(x in Union(P))
     * @ensures P' = P U {{x}}
     * @param x to make into a single element set in partition
     */
    public void makeSet( X x ) {
        entries.set( x.getIndex(), new Entry(x) );
    }

    /** @requires x IN union(P)
     * @ensures (exists S: P . x IN S && FindSet(x) IN S &&
     *              (forall y: S . FindSet(y) = FindSet(x) ) ) */
    private Entry findSet( X x ) {
        Entry ex = entries.get( x.getIndex() );
        assert ex != null;
        /* Find root of tree containing x */
        Entry s = ex.parent;
        while( s != s.parent ) {
            s = s.parent;
        }
        /* Compress path from x to the root s.
         * This does not change the partition, just its representation. */
        while( ex != s ) {
            Entry y = ex.parent;
            ex.parent = s;
            ex = y;
        }
        return s;
    }

    /* @requires (exists S,T: P . (x IN S) && (y IN T)
     * @returns true if and only if
     *      (exists S: P . (x IN S) && (y IN S))
     */
    public boolean equiv( X x, X y ) {
        return findSet(x) == findSet(y);
    }

    /* @requires (exists S,T: P . (x IN S) && (y IN T) && (S != T)
     * @ensures (exists S,T: P . (x IN S) && (y IN T) &&
     *                           P' = P - {S, T} U {S U T})
     */
    public void union( X x, X y) {
        Entry s = findSet( x );
        Entry t = findSet( y );
        if( s.rank > t.rank ) {
            t.parent = s;
        } else { /* s.rank <= t.rank */
            s.parent = t;
            if( s.rank == t.rank ) {
                t.rank++;
            }
        }
    }
}
```

```
package graphAlg;

import graphs.Graph.AdjacentEdge;
import graphs.UGraph;
import graphs.Vertex;

import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.ArrayList;

public class SpanningKruskal {

    public static class V extends Vertex implements Indexed {

        @Override
        public int getIndex() {
            return index;
        }
        @Override
        public void setIndex( int index ) {
            this.index = index;
        }
    }

    public class E {
        public V source;
        public V target;
        public int weight;

        public E( V from, V to, int weight ) {
            super( );
            this.source = from;
            this.target = to;
            this.weight = weight;
        }

        public int compareTo( E other ) {
            return (weight < other.weight ? -1 :
                weight > other.weight ? 1 : 0 );
        }

        public String toString() {
            return "Edge from " + source + " to " + target + " weight " + weight;
        }
    }

    private class EdgeComparator implements Comparator<E> {

        public int compare( E first, E second ) {
            return ( first.compareTo(second) );
        }
    }

    public List<E> minimalSpanningTree( UGraph<V,E> G ) {
        List<E> edges = new ArrayList<E>(G.size());
        for( V u : G ) {
            for( AdjacentEdge<V, E> e : G.adjacent(u) ) {
                edges.add( e.edgeInfo );
            }
        }
        Collections.sort(edges, new EdgeComparator() );
```



```
        /* set up partition of singleton sets */
        Partition<V> partition = new Partition<V>( G.size() );
        for( V u : G ) {
            partition.makeSet(u);
        }
        List<E> spanningTree = new ArrayList<E>(G.size()-1);
        for( E e : edges ) {
            if( !partition.equiv( e.source, e.target ) ) {
                // add edge to minimal spanning tree
                spanningTree.add(e);
                // union the components joined by the edge
                partition.union( e.source, e.target );
            }
        }
        return spanningTree;
    }
}
```

```
package graphAlg;

import graphAlg.SpanningKruskal.E;
import graphAlg.SpanningKruskal.V;
import graphs.UGraph;
import graphs.UGraphAdj;

import java.util.List;

public class SpanningKruskalTest {

    private static void addVertices( UGraph<V,E> G,
            SpanningKruskal mst, V... vertices ) {
        for( V v : vertices ) {
            G.addVertex( v );
        }
    }
    private static void addEdge( UGraph<V,E> G,
            SpanningKruskal mst, V i, V j, int w ) {
        G.addEdge( i, j, mst.new E(i,j,w) );
    }

    public static void runTest( UGraph<V,E> G, SpanningKruskal mst ) {
        List<E> edges = mst.minimalSpanningTree( G );
        System.out.println( "Minimal spanning tree with " +
                edges.size() + " edges" );
        int weight = 0;
        for( E e : edges ) {
            System.out.println( e );
            weight += e.weight;
        }
        System.out.println( "Weight " + weight );
    }
    public static void main(String[] args) {
        SpanningKruskal mst = new SpanningKruskal();
        UGraph<V,E> G = new UGraphAdj<V,E>();
        V v0 = new V();
        V v1 = new V();
        V v2 = new V();
        V v3 = new V();
        V v4 = new V();
        V v5 = new V();
        V v6 = new V();
        V v7 = new V();
        V v8 = new V();
        addVertices( G, mst, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, mst, v0, v1, 1 );
        addEdge( G, mst, v1, v2, 6 );
        addEdge( G, mst, v1, v6, 4 );
        addEdge( G, mst, v2, v3, 14 );
        addEdge( G, mst, v2, v6, 5 );
        addEdge( G, mst, v2, v4, 10 );
        addEdge( G, mst, v3, v4, 3 );
        addEdge( G, mst, v4, v5, 8 );
        addEdge( G, mst, v5, v6, 2 );
        addEdge( G, mst, v5, v8, 15 );
        addEdge( G, mst, v6, v7, 9 );

        runTest( G, mst );

        mst = new SpanningKruskal();
        G = new UGraphAdj<V,E>();
```



```
        v0 = new V();
        v1 = new V();
        v2 = new V();
        v3 = new V();
        v4 = new V();
        v5 = new V();
        v6 = new V();
        v7 = new V();
        v8 = new V();
        addVertices( G, mst, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, mst, v0, v1, 1 );
        addEdge( G, mst, v1, v2, 6 );
        addEdge( G, mst, v1, v6, 4 );
        addEdge( G, mst, v2, v3, 14 );
        addEdge( G, mst, v2, v6, 5 );
        addEdge( G, mst, v2, v4, 10 );
        addEdge( G, mst, v3, v4, 3 );
        addEdge( G, mst, v4, v5, 8 );
        addEdge( G, mst, v5, v6, 2 );
        addEdge( G, mst, v5, v8, 15 );

        runTest( G, mst );

        mst = new SpanningKruskal();
        G = new UGraphAdj<V,E>();
        v0 = new V();
        v1 = new V();
        v2 = new V();
        v3 = new V();
        v4 = new V();
        v5 = new V();
        v6 = new V();
        v7 = new V();
        v8 = new V();
        addVertices( G, mst, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, mst, v0, v1, 1 );
        addEdge( G, mst, v1, v2, 6 );
        addEdge( G, mst, v1, v6, 4 );
        addEdge( G, mst, v2, v3, 14 );
        addEdge( G, mst, v2, v6, 5 );
        addEdge( G, mst, v2, v4, 10 );
        addEdge( G, mst, v3, v4, 3 );
        addEdge( G, mst, v4, v5, 8 );
        addEdge( G, mst, v5, v6, 2 );
        addEdge( G, mst, v7, v8, 15 );

        runTest( G, mst );

    }
}
```

Sep 04, 14 16:39 **Indexed.java** Page 1/1

```
package graphAlg;

public interface Indexed {

    public int getIndex();

    public void setIndex( int index );

}
```

Sep 08, 14 10:04 **PriorityQueue.java** Page 1/2

```
package graphAlg;

import java.util.List;
import java.util.ArrayList;

/** Priority queue implementation as a binary heap in an array */
public class PriorityQueue<X extends HeapElement<X>> {

    /* Data type invariant:
     * for all i: 2..heap.size() . heap[i DIV 2] <= heap[i]
     */
    private List<X> heap;

    public PriorityQueue( int size ) {
        super();
        heap = new ArrayList<X>(size+1);
        /* Add dummy element at position 0 as heap indices start at 1 */
        heap.add(null);
    }
    /** The heap is empty if it only contains the dummy element at 0 */
    public boolean isEmpty() {
        return heap.size() == 1;
    }
    /** Update the heap at position i to contain element u and
     * update u's heap index to be i */
    private void heapUpdate( int i, X u ) {
        heap.set(i, u);
        u.setHeapIndex(i);
    }
    /** The element u has had it value decreased and needs to be sifted
     * down the heap to re-establish the heap data type invariant.
     */
    private void siftDown( X u ) {
        int i = u.getHeapIndex();
        while( i != 1 && u.compareTo(heap.get(i/2) ) < 0 ) {
            // System.out.println( "heap(" + i/2 + ") = " + heap.get(i/2) );
            heapUpdate(i, heap.get(i/2));
            i = i/2;
        }
        heapUpdate(i, u);
    }
    /** Add a new element to the end of the heap and sift it down to
     * re-establish the data type invariant.
     */
    public void add( X u ) {
        u.setHeapIndex(heap.size());
        heap.add( u );
        siftDown( u );
    }
    /** The minimum value in the heap is always at position 1 */
    public X findMin() {
        return heap.get(1);
    }
    /** Decrease the key of element u to newKey and sift it down to
     * re-establish the data type invariant.
     * @requires newKey <= u.getKey()
     */
    public void decreaseKey( X u, int newKey ) {
        System.out.println( "DecreaseKey " + u + " to " + newKey );
        u.setKey(newKey);
        siftDown( u );
        // System.out.println( "decreaseKey Min " + heap.get(1) );
```

```
    }

    private void siftUp( int i ) {
        while( 2*i < heap.size() ) {
            int j = 2*i;
            if( j+1 < heap.size() &&
                    heap.get(j+1).compareTo(heap.get(j)) < 0 ) {
                j = j+1;
            }
            if( heap.get(i).compareTo(heap.get(j)) > 0 ) {
                X temp = heap.get(i);
                heapUpdate(i, heap.get(j) );
                heapUpdate(j,  temp);
                i = j;
            } else {
                break;
            }
        }
    }

    public void deleteMin() {
        X last = heap.remove(heap.size()-1);
        if( isEmpty() ) {
            return;
        }
        heapUpdate(1, last);
        siftUp( 1 );
        // System.out.println( "deleteMin Min " + heap.get(1) );
    }


    public boolean checkInvariant() {
        boolean isHeap = true;
        for( int i = 2; i < heap.size(); i++ ) {
            if( heap.get(i/2).compareTo(heap.get(i)) > 0 ) {
                isHeap = false;
                System.out.println( "Not heap at " +
                        i/2 + " value " + heap.get(i/2) +
                        i + " value " + heap.get(i) );
            }
        }
        return isHeap;
    }
}
```



```
package graphAlg;

import graphs.Graph.AdjacentEdge;
import graphs.UGraph;
import graphs.Vertex;

public class SpanningPrim {

    public class V extends Vertex implements HeapElement<V> {
        public V parent;
        public int minWeight;
        public boolean inTree;
        int heapIndex;

        public V() {
            super();
            init();
        }
        public void init() {
            parent = null;
            minWeight = Integer.MAX_VALUE;
            inTree = false;
        }

        public int getHeapIndex() {
            return heapIndex;
        }

        public void setHeapIndex( int i ) {
            heapIndex = i;
        }

        public void setKey( int key ) {
            minWeight = key;
        }

        public int compareTo( V w ) {
            if( minWeight < w.minWeight ) {
                return -1;
            } else if( minWeight == w.minWeight ) {
                return 0;
            } else
                return 1;
        }

        public String toString() {
            return super.toString() + " weight " + minWeight;
        }

    }

    public class E {
        public int weight;

        public E( int weight ) {
            this.weight = weight;
        }
    }

    private PriorityQueue<V> PQ;
```

```
    public void minimalSpanningForest( UGraph<V,E> G ) {
        PQ = new PriorityQueue<V>( G.size() );
        for( V u : G ) {
            u.minWeight = Integer.MAX_VALUE;
            u.inTree = false;
            PQ.add( u );
        }
        for( V u : G ) {
            if( !u.inTree ) {
                minimalSpanningTree( G, u );
            }
        }
    }

    private void minimalSpanningTree( UGraph<V,E> G, V u ) {
        /* Treat u as the root of a spanning tree */
        u.parent = null;
        PQ.decreaseKey(u, 0);
        do {
            u = PQ.findMin();
            PQ.deleteMin();
            u.inTree = true;
            for( AdjacentEdge<V,E> e : G.adjacent(u) ) {
                V v = e.target;
                if( !v.inTree && (e.edgeInfo.weight < v.minWeight) ) {
                    v.parent = u;
                    PQ.decreaseKey( v, e.edgeInfo.weight );
                }
            }
        } while( !PQ.isEmpty() );
    }
}
```



```
package graphAlg;

import graphAlg.SpanningPrim.E;
import graphAlg.SpanningPrim.V;
import graphs.UGraph;
import graphs.UGraphAdj;

public class SpanningPrimTest {

    static SpanningPrim forest = new SpanningPrim();

    private static void addVertices( UGraph<V,E> G,
            SpanningPrim forest, V... vertices ) {
        for( V v : vertices ) {
            v.init();
            G.addVertex( v );
        }
    }
    private static void addEdge( UGraph<V,E> G,
            SpanningPrim forest, V i, V j, int w ) {
        G.addEdge( i, j, forest.new E(w) );
    }

    public static void runTest( UGraph<V,E> G, SpanningPrim forest ) {
        forest.minimalSpanningForest( G );
        for( V u : G ) {
            System.out.println( "Parent of " + u + " is " +
                    (u.parent == null ? "none" : u.parent) );
        }
        System.out.println();
    }
    public static void main(String[] args) {
        UGraph<V,E> G = new UGraphAdj<V,E>();
        V v0 = forest.new V();
        V v1 = forest.new V();
        V v2 = forest.new V();
        V v3 = forest.new V();
        V v4 = forest.new V();
        V v5 = forest.new V();
        V v6 = forest.new V();
        V v7 = forest.new V();
        V v8 = forest.new V();
        addVertices( G, forest, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v5, v6, 2 );
        addEdge( G, forest, v5, v8, 15 );
        addEdge( G, forest, v6, v7, 9 );

        runTest( G, forest );

        forest = new SpanningPrim();
        G = new UGraphAdj<V,E>();
        v0 = forest.new V();
        v1 = forest.new V();
        v2 = forest.new V();
```

```
        v3 = forest.new V();
        v4 = forest.new V();
        v5 = forest.new V();
        v6 = forest.new V();
        v7 = forest.new V();
        v8 = forest.new V();
        addVertices( G, forest, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v5, v6, 2 );
        addEdge( G, forest, v5, v8, 15 );

        runTest( G, forest );

        forest = new SpanningPrim();
        G = new UGraphAdj<V,E>();
        v0 = forest.new V();
        v1 = forest.new V();
        v2 = forest.new V();
        v3 = forest.new V();
        v4 = forest.new V();
        v5 = forest.new V();
        v6 = forest.new V();
        v7 = forest.new V();
        v8 = forest.new V();
        addVertices( G, forest, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v5, v6, 2 );
        addEdge( G, forest, v7, v8, 15 );

        runTest( G, forest );

    }
}
```



```
package graphAlg;

import graphs.DGraph;
import graphs.Graph.AdjacentEdge;
import graphs.Vertex;

public class ShortestPathDijkstra {

    public class V extends Vertex implements HeapElement<V> {
        public V parent;
        public int minDistance;
        public int heapIndex;
        public boolean isFinal;

        public V() {
            super();
            init();
        }

        public void init() {
            parent = null;
            minDistance = Integer.MAX_VALUE;
            isFinal = false;
        }

        public int getHeapIndex(){
            return heapIndex;
        }

        public void setHeapIndex( int index ) {
            heapIndex = index;
        }

        public void setKey( int distance ) {
            minDistance = distance;
        }

        public int compareTo( V w ) {
            if( minDistance < w.minDistance ) {
                return -1;
            } else if( minDistance == w.minDistance ) {
                return 0;
            } else
                return 1;
        }

        public String toString() {
            return super.toString() +
                    " distance " + minDistance + " parent " +
                    (parent == null ? " null " : parent );
        }
    }

    public class E {
        public V source;
        public V target;
        public int weight;

        public E( V source, V target, int weight ) {
            this.source = source;
            this.target = target;
            this.weight = weight;
```

```
        }

        public void relax() {
            if( source.minDistance + weight < target.minDistance) {
                target.parent = source;
                PQ.decreaseKey(target, source.minDistance + weight );
                //System.out.println( "  relaxed " + this );
            }
        }
    }

    public PriorityQueue<V> PQ;

    public void shortestPath( DGraph<V,E> G, V s ) {
        PQ = new PriorityQueue<V>( G.size() );
        for( V u : G ) {
            u.minDistance = Integer.MAX_VALUE;
            u.parent = null;
            PQ.add( u );
        }
        PQ.decreaseKey(s,0);
        while( !PQ.isEmpty() ) {
            V u = PQ.findMin();
            u.isFinal = true;
            PQ.deleteMin();
            for( AdjacentEdge<V,E> e : G.adjacent(u) ) {
                if( !e.edgeInfo.target.isFinal ) {
                    e.edgeInfo.relax();
                }
            }
        }
    }
}
```



```
package graphAlg;

import graphAlg.ShortestPathDijkstra.E;
import graphAlg.ShortestPathDijkstra.V;
import graphs.DGraph;
import graphs.DGraphAdj;

public class ShortestPathDijkstraTest {

    private static void addVertices( DGraph<V,E> G,
            ShortestPathDijkstra forest, V... vertices ) {
        for( V v : vertices ) {
            v.init();
            G.addVertex( v );
        }
    }
    private static void addEdge( DGraph<V,E> G,
            ShortestPathDijkstra forest, V i, V j, int w ) {
        G.addEdge( i, j, forest.new E(i,j,w) );
    }

    public static void runTest( DGraph<V,E> G,
            ShortestPathDijkstra forest, V s ) {
        forest.shortestPath( G, s );
        for( V u : G ) {
            System.out.println( u );
        }
    }
    public static void main(String[] args) {
        ShortestPathDijkstra forest = new ShortestPathDijkstra();
        DGraph<V,E> G = new DGraphAdj<V,E>();
        V v0 = forest.new V();
        V v1 = forest.new V();
        V v2 = forest.new V();
        V v3 = forest.new V();
        V v4 = forest.new V();
        V v5 = forest.new V();
        V v6 = forest.new V();
        V v7 = forest.new V();
        V v8 = forest.new V();
        addVertices( G, forest, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v6, v5, 2 );
        addEdge( G, forest, v5, v8, 15 );
        addEdge( G, forest, v6, v7, 9 );

        runTest( G, forest, v0 );

        forest = new ShortestPathDijkstra();
        G = new DGraphAdj<V,E>();
        v0 = forest.new V();
        v1 = forest.new V();
        v2 = forest.new V();
        v3 = forest.new V();
        v4 = forest.new V();
```

```
        v5 = forest.new V();
        v6 = forest.new V();
        v7 = forest.new V();
        v8 = forest.new V();
        addVertices( G, forest, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v5, v6, 2 );
        addEdge( G, forest, v5, v8, 15 );

        runTest( G, forest, v0 );

        forest = new ShortestPathDijkstra();
        G = new DGraphAdj<V,E>();
        v0 = forest.new V();
        v1 = forest.new V();
        v2 = forest.new V();
        v3 = forest.new V();
        v4 = forest.new V();
        v5 = forest.new V();
        v6 = forest.new V();
        v7 = forest.new V();
        v8 = forest.new V();
        addVertices( G, forest, v0, v1, v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v5, v6, 2 );
        addEdge( G, forest, v5, v7, 1 );
        addEdge( G, forest, v8, v7, 15 );

        runTest( G, forest, v0 );

    }
}
```



```
package graphAlg;

import graphs.DGraph;
import graphs.Graph.AdjacentEdge;
import graphs.Vertex;

public class ShortestPathBellmanFord {

    public class V extends Vertex {
        public V parent;
        public int minDistance;

        public V() {
            super();
            init();
        }

        public void init() {
            parent = null;
            minDistance = Integer.MAX_VALUE;
        }

        public int compareTo( V w ) {
            if( minDistance < w.minDistance ) {
                return -1;
            } else if( minDistance == w.minDistance ) {
                return 0;
            } else
                return 1;
        }

        public String toString() {
            return super.toString() +
                    " distance " + minDistance + " parent " +
                    (parent == null ? " null " : parent );
        }
    }

    public class E {
        public V source;
        public V target;
        public int weight;

        public E( V source, V target, int weight ) {
            super();
            this.source = source;
            this.target = target;
            this.weight = weight;
        }

        public void relax() {
            if( source.minDistance + weight < target.minDistance) {
                target.parent = source;
                target.minDistance = source.minDistance + weight;
                //System.out.println( "  relaxed " + this );
            }
        }
    }

    public boolean shortestPath( DGraph<V,E> G, V s ) {
        for( V u : G ) {
            u.minDistance = Integer.MAX_VALUE;
```

```
            u.parent = null;
        }
        s.minDistance = 0;
        for( int i = 1; i < G.size(); i++ ) {
            for( V u : G ) {
                for( AdjacentEdge<V, E> e : G.adjacent( u ) ) {
                    e.edgeInfo.relax();
                }
            }
        }
        for( V u : G ) {
            for( AdjacentEdge<V,E> e : G.adjacent( u ) ) {
                V es = e.edgeInfo.source;
                V et = e.edgeInfo.target;
                if( et.minDistance > es.minDistance + e.edgeInfo.weight ) {
                    return false;
                }
            }
        }
        return true;
    }
}
```



```
package graphAlg;

import graphAlg.ShortestPathBellmanFord.E;
import graphAlg.ShortestPathBellmanFord.V;
import graphs.DGraph;
import graphs.DGraphAdj;

public class ShortestPathBellmanFordTest {

    private static void addVertices( DGraph<V,E> G,
            ShortestPathBellmanFord forest, V... vertices ) {
        for( V v : vertices ) {
            v.init();
            G.addVertex( v );
        }
    }
    private static void addEdge( DGraph<V,E> G,
            ShortestPathBellmanFord forest, V i, V j, int w ) {
        G.addEdge( i, j, forest.new E(i,j,w) );
    }

    public static void runTest( DGraph<V,E> G,
            ShortestPathBellmanFord forest, V s ) {
        boolean ok = forest.shortestPath( G, s );
        if( ok ) {
            for( V u : G ) {
                System.out.println( u );
            }
        } else {
            System.out.println( "Graph has a negative weight cycle" );
        }
        System.out.println();
    }
    public static void main(String[] args) {
        ShortestPathBellmanFord forest = new ShortestPathBellmanFord();
        DGraph<V,E> G = new DGraphAdj<V,E>();
        V v0 = forest.new V();
        V v1 = forest.new V();
        V v2 = forest.new V();
        V v3 = forest.new V();
        V v4 = forest.new V();
        V v5 = forest.new V();
        V v6 = forest.new V();
        V v7 = forest.new V();
        V v8 = forest.new V();
        addVertices( G, forest, v0, v1,v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v6, v5, 2 );
        addEdge( G, forest, v5, v8, 15 );
        addEdge( G, forest, v6, v7, 9 );

        runTest( G, forest, v0 );

        forest = new ShortestPathBellmanFord();
        G = new DGraphAdj<V,E>();
```

```
        v0 = forest.new V();
        v1 = forest.new V();
        v2 = forest.new V();
        v3 = forest.new V();
        v4 = forest.new V();
        v5 = forest.new V();
        v6 = forest.new V();
        v7 = forest.new V();
        v8 = forest.new V();
        addVertices( G, forest, v0, v1,v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v5, v6, 2 );
        addEdge( G, forest, v5, v8, 15 );

        runTest( G, forest, v0 );

        forest = new ShortestPathBellmanFord();
        G = new DGraphAdj<V,E>();
        v0 = forest.new V();
        v1 = forest.new V();
        v2 = forest.new V();
        v3 = forest.new V();
        v4 = forest.new V();
        v5 = forest.new V();
        v6 = forest.new V();
        v7 = forest.new V();
        v8 = forest.new V();
        addVertices( G, forest, v0, v1,v2, v3, v4, v5, v6, v7, v8 );
        addEdge( G, forest, v0, v1, 1 );
        addEdge( G, forest, v1, v2, 6 );
        addEdge( G, forest, v1, v6, 4 );
        addEdge( G, forest, v2, v3, 14 );
        addEdge( G, forest, v2, v6, 5 );
        addEdge( G, forest, v2, v4, 10 );
        addEdge( G, forest, v3, v4, 3 );
        addEdge( G, forest, v4, v5, 8 );
        addEdge( G, forest, v5, v6, 2 );
        addEdge( G, forest, v7, v8, 15 );

        runTest( G, forest, v0 );

    }
}
```