```java
//*************************************************************************
//
// File:    UndirectedGraph.java
// Package: ---
// Unit:    Class UndirectedGraph
//
//*************************************************************************

import java.util.ArrayList;
import java.util.LinkedList;
import edu.rit.pj2.vbl.DoubleVbl;
import edu.rit.util.Random;

/**
 * Class UndirectedGraph represents an undirected graph meaning that if
 * there exists an edge connecting some vertex A to some vertex B, then
 * that same edge connects vertex B to vertex A.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class UndirectedGraph {

    // private data members
    private ArrayList<UndirectedEdge> edges;
    public ArrayList<Cricket> vertices;
    private int v;


    /**
     * Private constructor used internally by the static random graph
     * method
     * @param v the number of vertices in the graph
     */
    private UndirectedGraph(int v, CricketObserver o) {
        this.v = v;
        vertices = new ArrayList<Cricket>(v);
        edges = new ArrayList<UndirectedEdge>();
        for(int i = 0; i < v; i++) {
            vertices.add(new Cricket(i,o));
        }
    }

    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the id of the start vertex
     * @param goal the id of the goal vertex
     * @return the minimum distance between the two vertices
     */
    private int BFS(int start, int goal) {
        return BFS(vertices.get(start), vertices.get(goal));
    }

    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the reference to the start vertex
```

```
59      * @param goal the reference to the goal vertex
60      * @return the minimum distance between the two vertices
61      */
62     private int BFS(Cricket start, Cricket goal) {
63         int distance = 0, verticesToProcess = 1, uniqueNeighbors = 0;
64         LinkedList<Cricket> queue = new LinkedList<Cricket>();
65         boolean[] visited = new boolean[v];
66         visited[start.n] = true;
67         Cricket current, t2;
68         queue.add(start);
69         while(!queue.isEmpty()) {
70             current = queue.removeFirst();
71             if(current.equals(goal)) {
72                 return distance;
73             }
74             for(int i = 0; i < current.degree(); i++) {
75                 t2 = current.getEdges().get(i).other(current);
76                 if(!visited[t2.n]) {
77                     visited[t2.n] = true;
78                     queue.add(t2);
79                     uniqueNeighbors++;
80                 }
81             }
82             verticesToProcess--;
83             if(verticesToProcess <= 0) {
84                 verticesToProcess = uniqueNeighbors;
85                 uniqueNeighbors = 0;
86                 distance++;
87             }
88 
89         }
90         return 0;
91     }
92 
93     /**
94      * Accumulate the distances of each pair of vertices into
95      * a "running total" to be averaged
96      *
97      * @param thrLocal the reference to the "running total"
98      * Prof. Alan Kaminsky's library handles averaging this
99      * accumulated value.
100     */
101    public void accumulateDistances(DoubleVbl.Mean thrLocal) {
102        for(int i = 0; i < v; i++) {
103            for(int j = i + 1; j < v; j++) {
104                int distance = BFS(i, j);
105                // only accumulate the distance if the two vertices
106                // are actually connected
107                if(distance > 0) {
108                    thrLocal.accumulate(distance);
109                }
110            }
111        }
112    }
113 
114    public void tick(int tick) {
115        Cricket c;
116        for(int i = 0; i < v; i++) {
```

```
            c = vertices.get(i);
            c.timeTick(tick);
        }
        for(int i = 0; i < v; i++) {
            c = vertices.get(i);
            c.emitChirp();
        }
    }

    /**
     * Generate a random graph with a PRNG, a specified vertex count and
     * an edge probability
     *
     * @param prng Prof. Alan Kaminsky's Perfect Random Number Generator
     * @param v number of vertices to use
     * @param p edge probability between vertices
     * @return the randomly generated graph
     */
    public static UndirectedGraph randomGraph(Random prng, int v, double p,
            CricketObserver o) {
        UndirectedGraph g = new UndirectedGraph(v, o);
        UndirectedEdge edge;
        Cricket a, b;
        int edgeCount = 0;
        for (int i = 0; i < v; i++) {
            for (int j = i + 1; j < v; j++) {
                // connect edges
                // always order it `i` then `j`
                if(prng.nextDouble() <= p) {
                    a = g.vertices.get(i);
                    b = g.vertices.get(j);
                    edge = new UndirectedEdge(edgeCount++, a, b);
                    g.edges.add(edge);
                }
            }
        }
        return g;
    }

    public static UndirectedGraph cycleGraph(int v, CricketObserver o) {
        return kregularGraph(v, 1, o);
    }

    public static UndirectedGraph kregularGraph(int v, int k,
            CricketObserver o) {
        return smallWorldGraph(null, v, k, 0, o);
    }

    public static UndirectedGraph smallWorldGraph(Random prng, final int v,
            int k, double p, CricketObserver o) {
        UndirectedGraph g = new UndirectedGraph(v, o);
        UndirectedEdge edge;
        Cricket a, b, c;
        int edgeCount = 0;
        for(int i = 0; i < v; i++) {
            a = g.vertices.get(i);
            for(int j = 1; j <= k; j++) {
                b = g.vertices.get((i + j) % v);
```

```
175                 if(prng != null && prng.nextDouble() < p) {
176                     do {
177                         c = g.vertices.get(prng.nextInt(v));
178                     } while(c.n == a.n || c.n == b.n || a.directFlight(c));
179                     b = c;
180                 }
181                 edge = new UndirectedEdge(edgeCount++, a, b);
182                 g.edges.add(edge);
183             }
184         }
185         return g;
186     }
187 
188     public static UndirectedGraph scaleFreeGraph(Random prng, final int v,
189             final int dE, CricketObserver o) {
190         UndirectedGraph g = new UndirectedGraph(v, o);
191 //      boolean[]
192         int edgeCount = 0;
193         int c0 = prng.nextInt(v);
194         int c1 = (c0 + 1) % v;
195         int c2 = (c1 + 1) % v;
196         Cricket a = g.vertices.get(c0), b = g.vertices.get(c1),
197                 c = g.vertices.get(c2);
198         UndirectedEdge edge = new UndirectedEdge(edgeCount++, a, b);
199         g.edges.add(edge);
200         edge = new UndirectedEdge(edgeCount++, b, c);
201         g.edges.add(edge);
202         edge = new UndirectedEdge(edgeCount++, a, c);
203         g.edges.add(edge);
204         // we have 3 fully connected vertices now
205         Cricket[] others = new Cricket[v-3];
206         for(int other = 0, i = 0; i < v; i++) {
207             if(i != c0 && i != c1 && i != c2) {
208                 others[other++] = g.vertices.get(i);
209             }
210         }
211         // the rest are contained in others
212         int[] prob;
213         Cricket next, temp;
214         ArrayList<Cricket> existing = new ArrayList<Cricket>();
215         existing.add(a); existing.add(b); existing.add(c);
216         for(int i = 0; i < others.length; i++) {
217             next = others[i];
218             existing.add(next);
219             if(existing.size() <= dE) {
220                 for(int e = 0; e < existing.size(); e++) {
221                     temp = existing.get(e);
222                     if(next.equals(temp)) continue;
223                     edge = new UndirectedEdge(edgeCount++, temp, next);
224                     g.edges.add(edge);
225                 }
226             } else {
227                 // potential bug - when do i add in the current vertex to the
228                 // probability distribution?
229                 int sumD = sumDeg(g);
230                 prob = new int[sumD];
231                 setProbabilityDistribution(g, prob);
232                 for(int e = 0; e < dE; e++) {
```

```
233                     do {
234                         int chosen = (int) Math.floor(prng.nextDouble() *
235                                 prob.length);
236                         temp = g.vertices.get(prob[chosen]);
237                     } while(next.directFlight(temp));
238                     edge = new UndirectedEdge(edgeCount++, next, temp);
239                     g.edges.add(edge);
240                 }
241             }
242         }
243 
244         return g;
245     }
246 
247     private static void setProbabilityDistribution(UndirectedGraph g,
248             int[] prob) {
249         Vertex v;
250         int degree = 0;
251         int counter = 0;
252         for(int i = 0; i < g.v; i++) {
253             v = g.vertices.get(i);
254             degree = v.degree();
255             for(int j = counter; j < degree + counter; j++) {
256                 prob[j] = v.n;
257             }
258             counter += degree;
259         }
260     }
261 
262     private static int sumDeg(UndirectedGraph g) {
263         int retval = 0;
264         Vertex v;
265         for(int i = 0; i < g.v; i++) {
266             v = g.vertices.get(i);
267             retval += v.degree();
268         }
269         return retval;
270     }
271 }
272 
```