```
 3 // File:    UndirectedGraph.java
 8
 9 import java.util.ArrayList;
13
14 /**
15  * Class UndirectedGraph represents an undirected graph meaning that
   if
16  * there exists an edge connecting some vertex A to some vertex B,
   then
17  * that same edge connects vertex B to vertex A.
18  *
19  * @author Jimi Ford
20  * @version 2-15-2015
21  */
22 public class UndirectedGraph {
23
24     // private data members
25     private ArrayList<UndirectedEdge> edges;
26     public ArrayList<Cricket> vertices;
27     private int v;
28
29     // Prevent construction
30     private UndirectedGraph() {
31
32     }
33
34     /**
35      * Private constructor used internally by the static random
   graph
36      * method
37      * @param v the number of vertices in the graph
38      */
39     private UndirectedGraph(int v, CricketObserver o) {
40         this.v = v;
41         vertices = new ArrayList<Cricket>(v);
42         edges = new ArrayList<UndirectedEdge>();
43         for(int i = 0; i < v; i++) {
44             vertices.add(new Cricket(i,o));
45         }
46     }
47
```

```
48      /**
49       * Perform a BFS to get the distance from one vertex to another
50       *
51       * @param start the id of the start vertex
52       * @param goal the id of the goal vertex
53       * @return the minimum distance between the two vertices
54       */
55      private int BFS(int start, int goal) {
56          return BFS(vertices.get(start), vertices.get(goal));
57      }
58
59      /**
60       * Perform a BFS to get the distance from one vertex to another
61       *
62       * @param start the reference to the start vertex
63       * @param goal the reference to the goal vertex
64       * @return the minimum distance between the two vertices
65       */
66      private int BFS(Cricket start, Cricket goal) {
67          int distance = 0, verticesToProcess = 1, uniqueNeighbors =
    0;
68          LinkedList<Cricket> queue = new LinkedList<Cricket>();
69          boolean[] visited = new boolean[v];
70          visited[start.n] = true;
71          Cricket current, t2;
72          queue.add(start);
73          while(!queue.isEmpty()) {
74              current = queue.removeFirst();
75              if(current.equals(goal)) {
76                  return distance;
77              }
78              for(int i = 0; i < current.degree(); i++) {
79                  t2 = current.getEdges().get(i).other(current);
80                  if(!visited[t2.n]) {
81                      visited[t2.n] = true;
82                      queue.add(t2);
83                      uniqueNeighbors++;
84                  }
85              }
86              verticesToProcess--;
87              if(verticesToProcess <= 0) {
```

```
88                  verticesToProcess = uniqueNeighbors;
89                  uniqueNeighbors = 0;
90                  distance++;
91              }
92
93          }
94          return 0;
95      }
96
97      /**
98       * Accumulate the distances of each pair of vertices into
99       * a "running total" to be averaged
100      *
101      * @param thrLocal the reference to the "running total"
102      * Prof. Alan Kaminsky's library handles averaging this
103      * accumulated value.
104      */
105     public void accumulateDistances(DoubleVbl.Mean thrLocal) {
106         for(int i = 0; i < v; i++) {
107             for(int j = i + 1; j < v; j++) {
108                 int distance = BFS(i, j);
109                 // only accumulate the distance if the two vertices
110                 // are actually connected
111                 if(distance > 0) {
112                     thrLocal.accumulate(distance);
113                 }
114             }
115         }
116     }
117
118     public void tick(int tick) {
119         Cricket c;
120         for(int i = 0; i < v; i++) {
121             c = vertices.get(i);
122             c.timeTick(tick);
123         }
124         for(int i = 0; i < v; i++) {
125             c = vertices.get(i);
126             c.emitChirp();
127         }
128     }
```

```
129
130     /**
131      * Generate a random graph with a PRNG, a specified vertex count
    and
132      * an edge probability
133      *
134      * @param prng Prof. Alan Kaminsky's Perfect Random Number
    Generator
135      * @param v number of vertices to use
136      * @param p edge probability between vertices
137      * @return the randomly generated graph
138      */
139     public static UndirectedGraph randomGraph(Random prng, int v,
    double p, CricketObserver o) {
140         UndirectedGraph g = new UndirectedGraph(v, o);
141         UndirectedEdge edge;
142         Cricket a, b;
143         int edgeCount = 0;
144         for (int i = 0; i < v; i++) {
145             for (int j = i + 1; j < v; j++) {
146                 // connect edges
147                 // always order it `i` then `j`
148                 if(prng.nextDouble() <= p) {
149                     a = g.vertices.get(i);
150                     b = g.vertices.get(j);
151                     edge = new UndirectedEdge(edgeCount++, a, b);
152                     g.edges.add(edge);
153                 }
154             }
155         }
156         return g;
157     }
158
159     public static UndirectedGraph cycleGraph(int v, CricketObserver
    o) {
160         return kregularGraph(v, 1, o);
161     }
162
163     public static UndirectedGraph kregularGraph(int v, int k,
    CricketObserver o) {
164         return smallWorldGraph(null, v, k, 0, o);
```

```
165     }
166
167     public static UndirectedGraph smallWorldGraph(Random prng, final
    int v, int k, double p, CricketObserver o) {
168         UndirectedGraph g = new UndirectedGraph(v, o);
169         UndirectedEdge edge;
170         Cricket a, b, c;
171         int edgeCount = 0;
172         for(int i = 0; i < v; i++) {
173             a = g.vertices.get(i);
174             for(int j = 1; j <= k; j++) {
175                 b = g.vertices.get((i + j) % v);
176                 if(prng != null && prng.nextDouble() < p) {
177                     do {
178                         c = g.vertices.get(prng.nextInt(v));
179                     } while(c.n == a.n || c.n == b.n ||
    a.directFlight(c));
180                     b = c;
181                 }
182                 edge = new UndirectedEdge(edgeCount++, a, b);
183                 g.edges.add(edge);
184             }
185         }
186         return g;
187     }
188
189     public static UndirectedGraph scaleFreeGraph(Random prng, final
    int v,
190             final int dE, CricketObserver o) {
191         UndirectedGraph g = new UndirectedGraph(v, o);
192 //      boolean[]
193         int edgeCount = 0;
194         int c0 = prng.nextInt(v);
195         int c1 = (c0 + 1) % v;
196         int c2 = (c1 + 1) % v;
197         Cricket a = g.vertices.get(c0), b = g.vertices.get(c1), 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() * prob.length);
235                         temp = g.vertices.get(prob[chosen]);
236                     } while(next.directFlight(temp));
237                     edge = new UndirectedEdge(edgeCount++, next,
    temp);
238                     g.edges.add(edge);
```

```
239                 }
240             }
241         }
242 
243         return g;
244     }
245 
246     private static void setProbabilityDistribution(UndirectedGraph
    g, int[] prob) {
247         Vertex v;
248         int degree = 0;
249         int counter = 0;
250         for(int i = 0; i < g.v; i++) {
251             v = g.vertices.get(i);
252             degree = v.degree();
253             for(int j = counter; j < degree + counter; j++) {
254                 prob[j] = v.n;
255             }
256             counter += degree;
257         }
258     }
259 
260     private static int sumDeg(UndirectedGraph g) {
261         int retval = 0;
262         Vertex v;
263         for(int i = 0; i < g.v; i++) {
264             v = g.vertices.get(i);
265             retval += v.degree();
266         }
267         return retval;
268     }
269 }
270 
```