UndirectedGraph.java

```
2 //
3 // File: UndirectedGraph.java
4 // Package: ---
 5// Unit:
            Class UndirectedGraph
9 import java.util.ArrayList;
10 import java.util.LinkedList;
11 import edu.rit.pj2.vbl.DoubleVbl;
12 import edu.rit.util.Random;
13 import edu.rit.util.Searching;
14
15 /**
16 * Class UndirectedGraph represents an undirected graph meaning that if
17 * there exists an edge connecting some vertex A to some vertex B, then
18 * that same edge connects vertex B to vertex A.
19 *
20 * @author Jimi Ford
21 * @version 2-15-2015
22 */
23 public class UndirectedGraph {
24
25
     // private data members
26
     private ArrayList<UndirectedEdge> edges;
27
     public ArrayList<Cricket> vertices;
28
     private int v;
29
30
     /**
31
32
      * Private constructor used internally by the static random graph
33
34
      * @param v the number of vertices in the graph
35
36
     private UndirectedGraph(int v, CricketObserver o) {
37
         this.v = v;
         vertices = new ArrayList<Cricket>(v);
38
39
         edges = new ArrayList<UndirectedEdge>();
40
         for(int i = 0; i < v; i++) {</pre>
41
             vertices.add(new Cricket(i,o));
42
43
     }
44
45
46
      * Perform a BFS to get the distance from one vertex to another
47
      * @param start the id of the start vertex
48
49
      * @param goal the id of the goal vertex
      * @return the minimum distance between the two vertices
50
51
52
     private int BFS(int start, int goal) {
53
         return BFS(vertices.get(start), vertices.get(goal));
54
     }
55
56
57
      * Perform a BFS to get the distance from one vertex to another
58
```

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

```
117
        * @param tick the current time tick
118
119
       public void tick(int tick) {
120
           Cricket c;
121
           for(int i = 0; i < v; i++) {</pre>
122
                c = vertices.get(i);
123
                c.timeTick(tick);
124
125
           for(int i = 0; i < v; i++) {</pre>
126
                c = vertices.get(i);
127
                c.emitChirp();
128
           }
129
       }
130
131
132
        * Generate a random graph with a PRNG, a specified vertex count and
133
        * an edge probability
134
        * @param prng Prof. Alan Kaminsky's Perfect Random Number Generator
135
136
        * @param v number of vertices to use
137
        * @param p edge probability between vertices
138
        * @return the randomly generated graph
139
140
       public static UndirectedGraph randomGraph(Random prng, int v, double p,
141
                CricketObserver o) {
142
           UndirectedGraph g = new UndirectedGraph(\lor, o);
143
           UndirectedEdge edge;
144
           Cricket a, b;
145
           int edgeCount = 0;
146
           for (int i = 0; i < v; i++) {
147
                for (int j = i + 1; j < v; j++) {
148
                    // connect edges
                    // always order it `i` then `j`
149
                    if(prng.nextDouble() <= p) {</pre>
150
151
                        a = q.vertices.get(i);
                        b = g.vertices.get(j);
152
153
                        edge = new UndirectedEdge(edgeCount++, a, b);
154
                        g.edges.add(edge);
155
                    }
156
                }
157
           }
158
           return g;
159
       }
160
161
162
        * create a cycle graph
        * @param v number of vertices
163
164
        * @param o cricket observer crickets should report to
165
        * @return constructed cycle graph
166
167
       public static UndirectedGraph cycleGraph(int v, CricketObserver o) {
168
           return kregularGraph(v, 1, 0);
169
       }
170
171
        * create a k-regular graph
172
        * @param v number of vertices
173
174
        * @param k number of adjacent vertices left and right of given vertex to
```

```
175
        * connect to
        * @param o cricket observer the crickets should report to
176
177
        * @return the constructed k-regular graph
178
179
       public static UndirectedGraph kregularGraph(int v, int k,
180
                CricketObserver o) {
181
           return smallWorldGraph(null, v, k, 0, o);
182
       }
183
       /**
184
185
        * create a small-world graph
186
        * @param prng pseudorandom number generator
187
        * @param v number of vertices
188
        * @param k the initial k-regular graph to modify
189
        * @param p edge rewire probability
190
        * @param o cricket observer the crickets should report to
191
        * @return the constructed small-world graph
192
193
       public static UndirectedGraph smallWorldGraph(Random prng, final int v,
194
                int k, double p, CricketObserver o) {
195
           UndirectedGraph g = new UndirectedGraph(v, o);
196
           UndirectedEdge edge;
197
           Cricket a, b, c;
198
           int edgeCount = 0;
           for(int i = 0; i < v; i++) {</pre>
199
200
                a = g.vertices.get(i);
201
                for(int j = 1; j <= k; j++) {</pre>
202
                    b = g.vertices.get((i + j) % v);
203
                    if(prng != null && prng.nextDouble() < p) {</pre>
204
                        do {
205
                            c = q.vertices.get(prng.nextInt(v));
206
                        } while(c.n == a.n || c.n == b.n || a.directFlight(c));
207
                        b = c;
208
                    }
209
                    edge = new UndirectedEdge(edgeCount++, a, b);
210
                    q.edges.add(edge);
211
                }
212
           }
213
           return g;
214
       }
215
216
217
        * create a scale-free graph
218
        * @param prng psuedorandom number generator to use
219
        * @param v number of vertices
220
        * @param dE number of edges to add to each additional vertex
        * @param o cricket observer the crickets should report to
221
222
        * @return the scale-free graph generated
223
224
       public static UndirectedGraph scaleFreeGraph(Random prng, final int v,
225
                final int dE, CricketObserver o) {
226
           UndirectedGraph g = new UndirectedGraph(\lor, o);
227 //
           boolean[]
228
           int edgeCount = 0;
229
           int c0 = prnq.nextInt(v);
230
           int c1 = (c0 + 1) \% \vee;
231
           int c2 = (c1 + 1) \% v;
232
           Cricket a = g.vertices.get(c0), b = g.vertices.get(c1),
```

```
233
                    c = q.vertices.get(c2);
234
           UndirectedEdge edge = new UndirectedEdge(edgeCount++, a, b);
235
            g.edges.add(edge);
236
           edge = new UndirectedEdge(edgeCount++, b, c);
237
            q.edges.add(edge);
238
           edge = new UndirectedEdge(edgeCount++, a, c);
239
           g.edges.add(edge);
240
           // we have 3 fully connected vertices now
241
           Cricket[] others = new Cricket[v-3];
           for(int other = 0, i = 0; i < v; i++) {
242
243
                if(i != c0 && i != c1 && i != c2) {
244
                    others[other++] = q.vertices.get(i);
245
                }
246
           }
247
           // the rest are contained in others
248
           double[] cum = new double[v];
249
           double[] deg = new double[v];
250
           Cricket next, temp;
251
           ArrayList<Cricket> existing = new ArrayList<Cricket>();
252
            existing.add(a); existing.add(b); existing.add(c);
253
           Searching.Double h = new Searching.Double();
254
           for(int i = 0; i < others.length; i++) {</pre>
255
                next = others[i];
256
                existing.add(next);
257
                if(existing.size() <= dE) {</pre>
                    for(int e = 0; e < existing.size(); e++) {</pre>
258
259
                        temp = existing.get(e);
260
                        if(next.equals(temp)) continue;
261
                        edge = new UndirectedEdge(edgeCount++, temp, next);
262
                        g.edges.add(edge);
263
264
                } else {
265
                    setDegreeDistribution(g, deg);
266
                    for(int e = 0; e < dE; e++) {</pre>
267
                        setProbabilityDistribution(deg, cum);
268
                        double nr = prng.nextDouble();
269
                        double ch = cum[cum.length-1]*nr;
                        int vertex = Searching.searchInterval(cum, ch, h);
270
271
                        deg[vertex] = 0;
272
                        temp = g.vertices.get(vertex);
273
                        edge = new UndirectedEdge(edgeCount++, next, temp);
274
                        g.edges.add(edge);
275
                    }
276
                }
277
           }
278
279
           return g;
280
       }
281
282
283
        * set the degree distribution of a given graph
284
        * @param g the given graph
285
        * @param deg the degree distribution of the graph
286
287
       private static void setDegreeDistribution(UndirectedGraph q, double[] deg) {
288
           for(int i = 0; i < g.v; i++) {</pre>
289
                deg[i] = g.vertices.get(i).degree();
290
           }
```

UndirectedGraph.java

```
291
       }
292
293
       * set the cumulative sum of the degree array
294
       * @param deg degrees of a graph
295
       * @param cum cumulative sum of the degree
296
297
       private static void setProbabilityDistribution(double deg[], double[] cum) {
298
299
           double cumulative = 0;
           for(int i = 0; i < deg.length; i++) {</pre>
300
301
               cum[i] = cumulative += deg[i];
302
303
       }
304 }
305
```