

Automator.java

```
1 //*****
2 //
3 // File:    Automator.java
4 // Package: ---
5 // Unit:    Class Automator
6 //
7 //*****
8
9 import java.io.IOException;
10 import java.nio.charset.Charset;
11 import java.nio.file.Files;
12 import java.nio.file.Paths;
13 import java.util.List;
14
15 /**
16  * This class automates many calls to the Chirp main method
17  * by using command line arguments from an automation file.
18  *
19  * Each line in the file must either be commented out with
20  * a '#', or be a valid command for Chirp.java.
21  *
22  * @author Jimi Ford (jhf3617)
23  * @version 3-31-2015
24  */
25 public class Automator {
26
27     /**
28      *
29      * @param args command line arguments
30      * @param args[0] = automation file
31      */
32     public static void main(String[] args) {
33         if(args.length != 1) {
34             usage();
35         }
36         try {
37             List<String> lines = Files.readAllLines(Paths.get(args[0]),
38                 Charset.defaultCharset());
39             String[] lineArr;
40             int lineCount = 0;
41             boolean skip, comment;
42             for (String line : lines) {
43                 ++lineCount;
44                 line = line.trim();
45                 lineArr = line.split(" ");
46                 skip = lineArr[0].equals(line);
47                 comment = lineArr[0].startsWith("#");
48                 if(skip || comment) {
49                     if(comment) {
50                         if(line.equals("#")) {
51                             System.out.println();
52                         } else {
53                             System.out.println(line);
54                         }
55                     }
56                     continue;
57                 }
58                 Chirp.main(lineArr);
```

```
59     }
60     } catch (IOException e) {
61         error("Error reading automation file");
62     }
63 }
64
65 /**
66  * display usage message and exit
67  */
68 private static void usage() {
69     System.err.println("usage: java Automator <automation file>");
70     System.exit(1);
71 }
72
73 /**
74  * print error message and call usage()
75  * @param msg
76  */
77 private static void error(String msg) {
78     System.err.println(msg);
79     usage();
80 }
81 }
82
```

CricketObserver.java

```
1 //*****
2 //
3 // File:    CricketObserver.java
4 // Package: ---
5 // Unit:    Class CricketObserver
6 //
7 //*****
8
9 /**
10  * Class observes a group of crickets for a given number of time ticks and
11  * keeps track of whether or not they have chirped or not.
12  *
13  * @author Jini Ford (jhf3617)
14  * @version 3-31-2015
15  */
16 public class CricketObserver {
17
18     /**
19      * the number of crickets being observed
20      */
21     public final int crickets;
22
23     /**
24      * the number of time ticks observing for
25      */
26     public final int ticks;
27
28     // private data members
29     private boolean[][] chirps;
30
31     /**
32      * Construct a cricket observer
33      * @param crickets the number of crickets to observe
34      * @param ticks the number of time ticks observing for
35      */
36     public CricketObserver(int crickets, int ticks) {
37         this.crickets = crickets;
38         this.ticks = ticks;
39         chirps = new boolean[ticks][crickets];
40     }
41
42     /**
43      * called by a cricket to inform the observer that he has chirped
44      * @param tick the time tick at which the cricket is chirping
45      * @param n the unique identifier of the cricket
46      */
47     public void reportChirp(int tick, int n) {
48         chirps[tick][n] = true;
49     }
50
51     /**
52      * lookup a given time and cricket to see if it chirped at that moment
53      * @param tick the moment in time to lookup
54      * @param cricket the unique identifier of the cricket to check
55      * @return true if it chirped
56      */
57     public boolean chirped(int tick, int cricket) {
58         return chirps[tick][cricket];
59     }
60 }
```

```

59     }
60
61     /**
62     * get the time tick at which all the crickets being observed synchronized
63     * @return a number >= 0 if they synchronized, -1 if they didn't
64     */
65     public int sync() {
66         int row = 0;
67         while(row < ticks) {
68             if(sync(row)) return row;
69             row++;
70         }
71         return -1;
72     }
73
74     /**
75     * determine whether the crickets were synchronized at a given time tick or
76     * not
77     * @param tick the time tick to test
78     * @return true if every cricket at this time tick chirped
79     */
80     private boolean sync(int tick) {
81         boolean retval = true;
82         for(int i = 0; i < crickets && retval; i++) {
83             retval = chirps[tick][i];
84         }
85         return retval;
86     }
87 }
88

```

Ticker.java

```
1 //*****
2 //
3 // File:    Ticker.java
4 // Package: ---
5 // Unit:    Class Ticker
6 //
7 //*****
8
9 /**
10  * Class simulates a number of time ticks on a given network of crickets
11  * @author Jimi Ford (jhf3617)
12  * @version 3-31-2015
13  */
14 public class Ticker {
15
16     /**
17      * tick a number of time ticks on a given network of crickets
18      * @param g the network of crickets to tick
19      * @param ticks the number of ticks to simulate
20      */
21     public static void tick(UndirectedGraph g, int ticks) {
22         for(int i = 0; i < ticks; i++) {
23             g.tick(i);
24         }
25     }
26 }
27
```

UndirectedEdge.java

```
1 //*****
2 //
3 // File:    UndirectedEdge.java
4 // Package: ---
5 // Unit:    Class UndirectedEdge
6 //
7 //*****
8
9 /**
10  * Class UndirectedEdge represents an edge in a graph that connects two
11  * vertices. It's important to note that the edge does not have a direction nor
12  * weight.
13  *
14  * @author Jimi Ford
15  * @version 2-15-2015
16  */
17 public class UndirectedEdge {
18
19     // private data members
20     private Cricket a, b;
21
22     // future projects may rely on a unique identifier for an edge
23     private final int id;
24
25     /**
26      * Construct an undirected edge
27      * @param id a unique identifier to distinguish between other edges
28      * @param a one vertex in the graph
29      * @param b another vertex in the graph not equal to <I>a</I>
30      */
31     public UndirectedEdge(int id, Cricket a, Cricket b) {
32         this.id = id;
33         // enforce that a.n is always less than b.n
34         if(a.n < b.n) {
35             this.a = a;
36             this.b = b;
37         } else if(b.n < a.n) {
38             this.a = b;
39             this.b = a;
40         } else {
41             throw new IllegalArgumentException("Cannot have self loop");
42         }
43         this.a.addEdge(this);
44         this.b.addEdge(this);
45     }
46
47     /**
48      * Get the <I>other</I> vertex given a certain vertex connected to
49      * this edge
50      *
51      * @param current the current vertex
52      * @return the other vertex connected to this edge
53      */
54     public Cricket other(Cricket current) {
55         if(current == null) return null;
56         return current.n == a.n ? b : a;
57     }
58 }
```

Chirp.java

```
1 //*****
2 //
3 // File:    Chirp.java
4 // Package: ---
5 // Unit:    Class Chirp
6 //
7 //*****
8
9 import java.io.IOException;
10
11
12 /**
13  * Chirp runs a simulation of crickets chirping at night. The phenomenon we are
14  * interested in studying is that some types of networks synchronize in how they
15  * chirp. Based on the command line parameters, chirp tests the type of network
16  * and determines what time the crickets synchronize.
17  *
18  * @author Jimi Ford (jhf3617)
19  * @version 3-31-2015
20  */
21 public class Chirp {
22
23     private static final int GRAPH_TYPE_INDEX = 0,
24                             NUM_VERTICES_INDEX = 1,
25                             NUM_TICKS_INDEX = 2,
26                             OUTPUT_IMAGE_INDEX = 3,
27                             SEED_INDEX = 4,
28                             K_INDEX = 4,
29                             DE_INDEX = 4,
30                             DE_SEED_INDEX = 5,
31                             EDGE_PROBABILITY_INDEX = 5,
32                             K_SEED_INDEX = 5,
33                             REWIRE_PROBABILITY_INDEX = 6;
34
35     /**
36      * main method
37      * @param args command line arguments
38      */
39     public static void main(String[] args) {
40         if(args.length != 4 && args.length != 5 &&
41            args.length != 6 && args.length != 7) usage();
42         int crickets = 0, ticks = 0, k = 0, dE = 0;
43         long seed = 0;
44         double prob = 0;
45         char mode;
46         String outputImage = args[OUTPUT_IMAGE_INDEX];
47
48         try {
49             crickets = Integer.parseInt(args[NUM_VERTICES_INDEX]);
50         } catch (NumberFormatException e) {
51             error("<num vertices> must be a number");
52         }
53         try {
54             ticks = Integer.parseInt(args[NUM_TICKS_INDEX]) + 1;
55         } catch (NumberFormatException e) {
56             error("<num ticks> must be numeric");
57         }
58         mode = args[GRAPH_TYPE_INDEX].toLowerCase().charAt(0);
59         if(!(mode == 'c' || mode == 'r' || mode == 'k' ||
```

```

60         mode == 's' || mode == 'f')) {
61         error("<graph type> must be either 'c' for cycle, "
62             + "'r' for random, "
63             + "'k' for k-regular, "
64             + "'s' for small-world, "
65             + "'f' for scale-free");
66     }
67     UndirectedGraph g = null;
68     CricketObserver o = new CricketObserver(crickets, ticks);
69     switch(mode) {
70     case 'r': // RANDOM GRAPH
71         try {
72             seed = Long.parseLong(args[SEED_INDEX]);
73             prob = Double.parseDouble(args[EDGE_PROBABILITY_INDEX]);
74             g = UndirectedGraph.randomGraph(
75                 new Random(seed), crickets, prob, o);
76         } catch (NumberFormatException e) {
77             error("<seed> and <edge probability> must be numeric");
78         } catch (IndexOutOfBoundsException e) {
79             error("<seed> and <edge probability> must be included with "
80                 + "random graph mode");
81         }
82         break;
83     case 'c': // CYCLE GRAPH
84         g = UndirectedGraph.cycleGraph(crickets, o);
85         break;
86     case 'k': // K-REGULAR GRAPH
87         try {
88             k = Integer.parseInt(args[K_INDEX]);
89             g = UndirectedGraph.kregularGraph(crickets, k, o);
90         } catch (NumberFormatException e) {
91             error("<k> must be an integer");
92         } catch (IllegalArgumentException e) {
93             error("<k> must be < the number of crickets");
94         }
95         break;
96     case 's': // SMALL WORLD GRAPH
97         try {
98             k = Integer.parseInt(args[K_INDEX]);
99             prob = Double.parseDouble(args[REWIRE_PROBABILITY_INDEX]);
100             seed = Long.parseLong(args[K_SEED_INDEX]);
101             g = UndirectedGraph.smallWorldGraph(
102                 new Random(seed), crickets, k, prob, o);
103         } catch (NumberFormatException e) {
104             error("<k> must be an integer < V, <rewire probability> "
105                 + "must be a number "
106                 + "between 0 and 1, and <seed> must be numeric");
107         } catch (IllegalArgumentException e) {
108             error("<k> must be < the number of crickets");
109         }
110         break;
111     case 'f': // SCALE-FREE GRAPH
112         try {
113             dE = Integer.parseInt(args[DE_INDEX]);
114             seed = Long.parseLong(args[DE_SEED_INDEX]);
115             g = UndirectedGraph.scaleFreeGraph(
116                 new Random(seed), crickets, dE, o);
117         } catch (NumberFormatException e) {

```



```

118         error("<dE> and <seed> must be numeric");
119     } catch (IndexOutOfBoundsException e) {
120         error("<dE> and <seed> must be supplied");
121     }
122 }
123
124 g.vertices.get(0).forceChirp();
125 Ticker.tick(g, ticks);
126
127
128
129 try {
130     ImageHandler.handle(o, outputImage);
131 } catch (IOException e) {
132     error("Problem writing image");
133 }
134 int sync = o.sync();
135 String description;
136 switch(mode) {
137     case 'c': // CYCLE GRAPH
138         description = "Cycle V = " + crickets + ":";
139         handleOutput(description, sync);
140         break;
141     case 'r': // RANDOM GRAPH
142         description = "Random V = " + crickets + ", p = " + prob + ":";
143         handleOutput(description, sync);
144         break;
145     case 'k': // K-REGULAR GRAPH
146         description = "K-regular V = " + crickets + ", k = " + k + ":";
147         handleOutput(description, sync);
148         break;
149     case 's': // SMALL-WORLD GRAPH
150         description = "Small-world V = " + crickets + ", k = " + k +
151             ", p = " + prob + ":";
152         handleOutput(description, sync);
153         break;
154     case 'f': // SCALE-FREE GRAPH
155         description = "Scale-free V = " + crickets + ", dE = " + dE + ":";
156         handleOutput(description, sync);
157         break;
158 }
159
160 }
161
162 /**
163  * handle printing the results of the simulation
164  * @param description the description of what kind of graph is being printed
165  * @param sync time at which the network synchronized
166  * (-1 for not synchronized)
167  */
168 private static void handleOutput(String description, int sync) {
169     System.out.print(description);
170     if(sync >= 0) {
171         System.out.println("\t" + "synchronized at t=" + sync + ".");
172     } else {
173         System.out.println("\t" + (char)27 + "[31m" + "did not synchronize." +
174             (char)27 + "[0m");
175     }

```

```

176     }
177
178     /**
179     * print an error message and call usage()
180     * @param msg
181     */
182     private static void error(String msg) {
183         System.err.println(msg);
184         usage();
185     }
186
187     /**
188     * usage message called when program improperly used
189     */
190     private static void usage() {
191         System.err.println(
192             "usage: java Chirp <graph type> <num vertices> <num ticks> "
193             + "<output image> {(<seed> <edge probability>), or "
194             + "(<k>), or "
195             + "(<k> <seed> <rewire probability>), or "
196             + "(<dE> <seed>)}" );
197         System.exit(1);
198     }
199 }
200

```

Cricket.java

```
1 //*****
2 //
3 // File:    Cricket.java
4 // Package: ---
5 // Unit:    Class Cricket
6 //
7 //*****
8
9 /**
10  * This class models a cricket that will chirp at time t + 2 if it hears a chirp
11  * at time t. It inherits from vertex so that it can be connected to other
12  * crickets through undirected edges.
13  *
14  * @author Jimi Ford (jhf3617)
15  * @version 3-31-2015
16  */
17 public class Cricket extends Vertex {
18
19     private boolean[] chirp = new boolean[2];
20     private boolean willChirp;
21     private int currentTick = 0;
22     private final CricketObserver observer;
23
24     /**
25      * Construct a cricket
26      * @param n the unique integer identifier
27      * @param o the cricket observer this cricket should report to
28      */
29     public Cricket(int n, CricketObserver o) {
30         super(n);
31         this.observer = o;
32     }
33
34     /**
35      * force a cricket to chirp at the next time tick
36      */
37     public void forceChirp() {
38         willChirp = chirp[0] = true;
39     }
40
41     /**
42      * will chirp only if it is being forced to, or if it has heard a chirp
43      * 2 time ticks ago
44      */
45     public void emitChirp() {
46         if(willChirp) {
47             willChirp = false;
48             int n = super.degree();
49             for(int i = 0; i < n; i++) {
50                 edges.get(i).other(this).hearChirp();
51             }
52             observer.reportChirp(currentTick, super.n);
53         }
54     }
55
56     /**
57      * hear another chirp from an adjacent cricket
58      */
59 }
```

```

59 private void hearChirp() {
60     chirp[1] = true;
61 }
62
63 /**
64  * simulate time passing by letting the cricket know what time it is
65  *
66  * @param tick the current time tick for this cricket
67  */
68 public void timeTick(int tick) {
69     currentTick = tick;
70     willChirp = chirp[0];
71     chirp[0] = chirp[1];
72     chirp[1] = false;
73 }
74
75 /**
76  * determine if a given cricket is directly connected to this cricket
77  * @param other the given cricket to check
78  * @return true if this cricket as a single edge that connects the two
79  */
80 public boolean directFlight(Cricket other) {
81     boolean retval = false;
82     if(equals(other)) return true;
83     int e = super.degree();
84     Cricket o;
85     for(int i = 0; i < e && !retval; i++) {
86         o = super.edges.get(i).other(this);
87         retval = o.equals(other);
88     }
89     return retval;
90 }
91
92 /**
93  * determine if another object is equal to this cricket
94  * @param o the other object
95  * @return true if the other object is equal to this cricket
96  */
97 public boolean equals(Object o) {
98     if( !(o instanceof Cricket)) {
99         return false;
100     }
101     if(o == this) {
102         return true;
103     }
104     Cricket casted = (Cricket) o;
105
106     return casted.n == this.n;
107 }
108 }
109

```

ImageHandler.java

```
1 //*****
2 //
3 // File:    ImageHandler.java
4 // Package: ---
5 // Unit:    Class ImageHandler
6 //
7 //*****
8
9 import java.io.BufferedOutputStream;
10
11
12 /**
13  * Class takes care of saving the results of the simulation as an image
14  *
15  * @author Jimi Ford (jhf3617)
16  * @version 3-31-2015
17  */
18 public class ImageHandler {
19
20     // private data members
21     private static final byte SILENT = 0,
22                             CHIRPED = 1,
23                             SYNC = 2;
24
25     /**
26      * @param o the cricket observer that holds the results of the simulation
27      * @param out the name of the image file to save
28      * @throws FileNotFoundException if there was an error writing to the given
29      * file
30      */
31     public static void handle(CricketObserver o, String out)
32         throws FileNotFoundException {
33         AList<Color> palette = new AList<Color>();
34         Color green = new Color().rgb(0, 255, 0); // green
35         Color red = new Color().rgb(255, 0, 0); // red
36         Color blue = new Color().rgb(0,0,255); // blue
37         palette.addLast (green);
38         palette.addLast (red);
39         palette.addLast (blue);
40
41         OutputStream imageout =
42             new BufferedOutputStream (new FileOutputStream (new File(out)));
43         IndexPngWriter imageWriter = new IndexPngWriter
44             (o.ticks, o.crickets, imageout, palette);
45         ByteImageQueue imageQueue = imageWriter.getImageQueue();
46         byte[] bytes;
47         boolean chirped;
48         int sync = o.sync();
49         for(int i = 0; i < o.ticks; i++) {
50             bytes = new byte[o.crickets];
51             for(int j = 0, cricket = 0; j < bytes.length; j++, cricket++) {
52                 if(i != sync) {
53                     chirped = o.chirped(i, cricket);
54                     bytes[j] = chirped ? CHIRPED : SILENT;
55                 } else {
56                     bytes[j] = SYNC;
57                 }
58             }
59         }
60     }
61 }
```

```
68         }
69     }
70     try {
71         imageQueue.put(i, bytes);
72     } catch (InterruptedException e) {
73         // TODO Auto-generated catch block
74         e.printStackTrace();
75     }
76 }
77 try {
78     imageWriter.write();
79 } catch (IOException e) {
80     // TODO Auto-generated catch block
81     e.printStackTrace();
82 } catch (InterruptedException e) {
83     // TODO Auto-generated catch block
84     e.printStackTrace();
85 }
86 }
87 }
88
```

UndirectedGraph.java

```
1 //*****
2 //
3 // File:    UndirectedGraph.java
4 // Package: ---
5 // Unit:    Class UndirectedGraph
6 //
7 //*****
8
9 import java.util.ArrayList;
10 import java.util.LinkedList;
11 import edu.rit.pj2.vbl.DoubleVbl;
12 import edu.rit.util.Random;
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
30     /**
31      * Private constructor used internally by the static random graph
32      * method
33      * @param v the number of vertices in the graph
34      */
35     private UndirectedGraph(int v, CricketObserver o) {
36         this.v = v;
37         vertices = new ArrayList<Cricket>(v);
38         edges = new ArrayList<UndirectedEdge>();
39         for(int i = 0; i < v; i++) {
40             vertices.add(new Cricket(i,o));
41         }
42     }
43
44     /**
45      * Perform a BFS to get the distance from one vertex to another
46      *
47      * @param start the id of the start vertex
48      * @param goal the id of the goal vertex
49      * @return the minimum distance between the two vertices
50      */
51     private int BFS(int start, int goal) {
52         return BFS(vertices.get(start), vertices.get(goal));
53     }
54
55     /**
56      * Perform a BFS to get the distance from one vertex to another
57      *
58      * @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      return 0;
90  }
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++) {

```



```

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

```

Vertex.java

```
1 //*****
2 //
3 // File:    Vertex.java
4 // Package: ---
5 // Unit:    Class Vertex
6 //
7 //*****
8
9 import java.util.ArrayList;
10
11 /**
12  * Class Vertex represents a single vertex in a graph. Vertices can be connected
13  * to other vertices through undirected edges.
14  *
15  * @author Jimi Ford
16  * @version 2-15-2015
17  */
18 public class Vertex {
19
20     // private data members
21     protected ArrayList<UndirectedEdge> edges = new ArrayList<UndirectedEdge>();
22
23     /**
24      * The unique identifier for this vertex
25      */
26     public final int n;
27
28     /**
29      * Construct a vertex with a unique identifier <I>n</I>
30      *
31      * @param n the unique identifier to distinguish this vertex from
32      *         all other vertices in the graph
33      */
34     public Vertex(int n) {
35         this.n = n;
36     }
37
38     /**
39      * Get the number of edges connected to this vertex
40      *
41      * @return the number of edges connected to this vertex
42      */
43     public int degree() {
44         return edges.size();
45     }
46
47     /**
48      * Get the reference to the collection of edges connected to
49      * this vertex.
50      *
51      * @return the reference to the collection of edges
52      */
53     public ArrayList<UndirectedEdge> getEdges() {
54         return this.edges;
55     }
56
57     /**
58      * Add an edge to this vertex
```

```
59      *
60      * @param e the edge to add
61      */
62      public void addEdge(UndirectedEdge e) {
63          this.edges.add(e);
64      }
65
66      /**
67       * Compare another object to this one
68       *
69       * @param o the other object to compare to this one
70       * @return true if the other object is equivalent to this one
71       */
72      public boolean equals(Object o) {
73          if( !(o instanceof Vertex)) {
74              return false;
75          }
76          if(o == this) {
77              return true;
78          }
79          Vertex casted = (Vertex) o;
80
81          return casted.n == this.n;
82      }
83 }
84
```