

MonteCarlo.java

```
1 //*****
2 //
3 // File:    MonteCarlo.java
4 // Package: ---
5 // Unit:    Class MonteCarlo
6 //
7 //*****
8
9 import java.io.FileNotFoundException;
10 import java.io.IOException;
11 import java.io.PrintWriter;
12 import edu.rit.pj2.Task;
13
14 /**
15  * Class MonteCarlo takes in a seed value for a random number
16  * generator, the upper and lower boundaries for the number of vertices in
17  * each graph as well as a number to increment by, the upper and lower
18  * boundaries for the edge probability as well as a number to increment by,
19  * the number of random graphs to generate for each combination of V (vertices)
20  * and p (edge probability), and finally, a prefix for naming each plot
21  * generated by this program. After checking for valid input, this program
22  * loops through each combination of vertices and edge probabilities, running
23  * the specified number of simulations on each combination. Each random graph
24  * (or simulation) is generated by looking at every possible pair of vertices,
25  * generating a random floating point between 0 and 1, and marking these
26  * vertices with an edge connecting them if the random value is less than or
27  * equal to the specified edge probability (for that unique graph). In each
28  * simulation, the distance values of each graph are calculated with a breadth
29  * first search from vertex A to vertex B using the depth of the search as the
30  * distance from A to B.
31  *
32  * @author Jimi Ford
33  * @version 2-15-2015
34  */
35 public class MonteCarlo extends Task {
36
37     // Private constants
38     private static final String[] arguments = {
39         "<seed>",
40         "<min_v>",
41         "<max_v>",
42         "<v_grain>",
43         "<min_p>",
44         "<max_p>",
45         "<p_grain>",
46         "<num_simulations>",
47         "<optional_plotfile_prefix>"
48     };
49
50     private static final int
51         SEED = 0,
52         MIN_VERTICES = 1,
53         MAX_VERTICES = 2,
54         VERTEX_GRANULARITY = 3,
55         MIN_P = 4,
56         MAX_P = 5,
57         P_GRANULARITY = 6,
58         NUMBER_OF_SIMULATIONS = 7,
```

```

59     PLOT_FILE_PREFIX = 8;
60
61     /**
62     * MonteCarlo's main method to be invoked by Prof. Alan Kaminsky's
63     * Parallel Java 2 library.
64     *
65     * @param args command line arguments
66     *
67     * <P>
68     * usage: java pj2 MonteCarlo <seed> <min_v> <max_v>
69     * <v_grain> <min_p> <max_p> <p_grain>
70     * <num_simulations> <optional plotfile prefix>
71     * <P>
72     */
73     public void main(String[] args) {
74         if(args.length != 8 && args.length != 9) {
75             usage();
76         }
77
78         long seed = 0;
79         int minVertices = 0, maxVertices = 0, vertexGranularity = 0,
80             numSimulations = 0;
81         double pGrain = 0, minP = 0, maxP = 0;
82
83         try {
84             seed = Long.parseLong(args[SEED]);
85         } catch (NumberFormatException e) {
86             displayError(
87                 String.format("Argument %1s must be numeric and between %2d "+
88                     "and %3d inclusive.\n", arguments[SEED],
89                     Long.MIN_VALUE, Long.MAX_VALUE));
90         }
91
92         try {
93             minVertices = Integer.parseInt(args[MIN_VERTICES]);
94             if(minVertices < 1) throw new NumberFormatException();
95         } catch (NumberFormatException e) {
96             displayError(
97                 String.format("Argument %1s must be numeric and between 1 "+
98                     "and %2d inclusive.\n", arguments[MIN_VERTICES],
99                     Integer.MAX_VALUE));
100         }
101
102         try {
103             maxVertices = Integer.parseInt(args[MAX_VERTICES]);
104             if(maxVertices < minVertices)
105                 displayError(String.format(
106                     "Argument %1s must be greater than or equal to %2s.\n",
107                     arguments[MAX_VERTICES], arguments[MIN_VERTICES]));
108         } catch (NumberFormatException e) {
109             displayError(String.format(
110                 "Argument %1s must be numeric and between 1 and %2d inclusive.\n",
111                 arguments[MAX_VERTICES], Integer.MAX_VALUE));
112         }
113
114         try {
115             vertexGranularity = Integer.parseInt(args[VERTEX_GRANULARITY]);
116             if(vertexGranularity < 1) throw new NumberFormatException();

```

```

117     } catch (NumberFormatException e) {
118         displayError(String.format(
119             "Argument %1s must be numeric and between 1 and %2d inclusive.\n",
120             arguments[VERTEX_GRANULARITY], Integer.MAX_VALUE));
121     }
122
123     try {
124         minP = Double.parseDouble(args[MIN_P]);
125         if(minP < 0 || minP > 1) throw new NumberFormatException();
126     } catch (NumberFormatException e) {
127         displayError(String.format(
128             "Argument %1s must be numeric and between "+
129             "0 inclusive and 1 inclusive.\n",
130             arguments[MIN_P]));
131     }
132
133     try {
134         maxP = Double.parseDouble(args[MAX_P]);
135         if(maxP < minP)
136             displayError(String.format(
137                 "Argument %1s must be greater than or equal to %2s.\n",
138                 arguments[MAX_P], arguments[MIN_P]));
139         if(maxP > 1) throw new NumberFormatException();
140     } catch (NumberFormatException e) {
141         displayError(String.format(
142             "Argument %1s must be numeric and between "+
143             "0 inclusive and 1 inclusive.\n",
144             arguments[MAX_P]));
145     }
146
147     try {
148         pGrain = Double.parseDouble(args[P_GRANULARITY]);
149         if(pGrain <= 0 || pGrain > 1)
150             throw new NumberFormatException();
151     } catch (NumberFormatException e) {
152         displayError(String.format(
153             "Argument %1s must be numeric and between "+
154             "0 exclusive and 1 inclusive.\n",
155             arguments[P_GRANULARITY]));
156     }
157
158     try {
159         numSimulations = Integer.parseInt(args[NUMBER_OF_SIMULATIONS]);
160         if(numSimulations < 1) throw new NumberFormatException();
161     } catch (NumberFormatException e) {
162         displayError(String.format(
163             "Argument %1s must be numeric and between 1 and %2d inclusive.\n",
164             arguments[NUMBER_OF_SIMULATIONS], Integer.MAX_VALUE));
165     }
166
167     // store file prefix
168     final String plotFilePrefix = args.length == 9 ?
169         args[PLOT_FILE_PREFIX] : "plot";
170
171     String pMinStr = Double.toString(minP);
172     String pMaxStr = Double.toString(maxP);
173     String pGrainStr = Double.toString(pGrain);
174     final int sigFig =

```

```

175         Math.max(Math.max(
176             pGrainStr.length() - pGrainStr.indexOf('.') - 1,
177             pMaxStr.length() - pMaxStr.indexOf('.') - 1),
178             pMinStr.length() - pMinStr.indexOf('.') - 1);
179     int exp = 1;
180     for(int i = 0; i < sigFig; i++) {
181         exp *= 10;
182     }
183     final int pMax = (int) (Math.round(maxP * exp));
184     final int pInc = (int) (Math.round(pGrain * exp));
185     // if 0 is the lower bound, set pMin to the next "step" of edge probability
186     // which is pInc
187     final int pMin = ((int) (Math.round(minP * exp))) == 0 ?
188         pInc : ((int) (Math.round(minP * exp)));
189     pGrainStr = null;
190
191
192
193     SimulationResultCollection results = new SimulationResultCollection(
194         minVertices, maxVertices, vertexGranularity, pMin, pMax, pInc, exp);
195
196     // loop through number of vertices
197     for(int vCount = minVertices; vCount <= maxVertices;
198         vCount += vertexGranularity) {
199         // loop through edgeProbability
200         for(int p = pMin; p <= pMax; p += pInc) {
201             double prob = p / (double) exp;
202             // loop through each simulation
203             results.add(new Simulation(this, seed, vCount, prob,
204                 numSimulations).simulate());
205         }
206         try {
207             new PlotHandler(plotFilePrefix, results, vCount).write();
208         } catch (IOException e) {
209             System.err.println("Error writing file for v="+vCount);
210         }
211     }
212
213     StringBuilder builder = new StringBuilder();
214     for(int p = 0; p <= pMax; p += pInc) {
215         builder.append(", " + (p / ((double) exp)));
216     }
217     builder.append('\n');
218     for(int v = minVertices; v <= maxVertices; v += vertexGranularity) {
219         builder.append(v + ", ");
220         for(int p = pMin; p <= pMax; p += pInc) {
221             builder.append(results.get(v,p)+" , ");
222         }
223         builder.append('\n');
224     }
225     PrintWriter tableWriter = null;
226     final String tableSuffix = "-table.csv";
227     try {
228         tableWriter = new PrintWriter(plotFilePrefix + tableSuffix);
229         tableWriter.print(builder.toString());
230     } catch (FileNotFoundException e) {
231         System.err.println("Error writing table data to file \""+
232             plotFilePrefix + tableSuffix + "\"");

```

```

233     } finally {
234         if(tableWriter != null) tableWriter.close();
235     }
236     System.out.println("Finished simulations! run \"java PlotHandler\" "+
237         "followed by any number of .dwg files (that were previously generated) "+
238         "to visualize the results.");
239 } // main
240
241
242 /**
243  * Display the proper usage of this program and exit.
244  */
245 private static void usage() {
246     System.err.printf ("Usage: java pj2 MonteCarlo "+
247         "%1s %2s %3s %4s %5s %6s %7s %8s %9s\n",
248         arguments[SEED],
249         arguments[MIN_VERTICES],
250         arguments[MAX_VERTICES],
251         arguments[VERTEX_GRANULARITY],
252         arguments[MIN_P],
253         arguments[MAX_P],
254         arguments[P_GRANULARITY],
255         arguments[NUMBER_OF_SIMULATIONS],
256         arguments[PLOT_FILE_PREFIX]);
257     System.exit(1);
258 }
259
260 /**
261  * Print an error message to System.err and gracefully exit
262  * @param msg the error message to display
263  */
264 private static void displayError(String msg) {
265     System.err.println(msg);
266     usage();
267 }
268 }
269

```

PlotHandler.java

```
1 //*****
2 //
3 // File:    PlotHandler.java
4 // Package: ---
5 // Unit:    Class PlotHandler
6 //
7 //*****
8
9 import java.io.File;
10 import java.io.IOException;
11 import java.text.DecimalFormat;
12 import edu.rit.numeric.ListXYSeries;
13 import edu.rit.numeric.plot.Dots;
14 import edu.rit.numeric.plot.Plot;
15
16 /**
17  * Class PlotHandler is the delegate for dealing with visualizing the data
18  * generated by the "number crunching" program, MonteCarlo. Its purpose is to
19  * be instantiated in MonteCarlo with the data to plot, where the write()
20  * method should then be called. Running this program and specifying in
21  * the command line arguments the plot files previously generated will
22  * open a graphical representation of these plots for each file.
23  *
24  * @author Jimi Ford
25  * @version 2-15-2015
26  *
27  */
28 public class PlotHandler {
29
30     // private data members
31     private final String fileName;
32     private final int v;
33     private final SimulationResultCollection collection;
34
35     /**
36      * Construct a new plot handler that plots average distances for a fixed
37      * vertex count v, while varying the edge probability p
38      *
39      * @param plotFilePrefix prefix to be used in the name of
40      *        the plot file
41      * @param collection collection of results of the finished set of
42      *        simulations.
43      * @param v number of vertices used in each simulation
44      */
45     public PlotHandler(String plotFilePrefix,
46                       SimulationResultCollection collection, int v) {
47         fileName = plotFilePrefix + "-V-" + v + ".dwg";
48         this.v = v;
49         this.collection = collection;
50     }
51
52     /**
53      * Save the plot information into a file to visualize by running
54      * the main method of this class
55      *
56      * @throws IOException if it can't write to the file specified
57      */
58     public void write() throws IOException {
```

```

59     ListXYSeries results = new ListXYSeries();
60     double[] values = collection.getAveragesForV(v);
61     for(int i = 0, p = collection.pMin; i < values.length; i++,
62         p += collection.pInc) {
63         results.add(p / ((double) collection.pExp), values[i]);
64     }
65
66     Plot plot = new Plot()
67         .plotTitle (String.format
68             ("Random Graphs, <I>V</I> = %1s", Integer.toString(v)))
69         .xAxisTitle ("Edge Probability <I>p</I>")
70         .xAxisTickFormat(new DecimalFormat("0.0"))
71         .yAxisTitle ("Average Distance <I>d</I>")
72         .yAxisTickFormat (new DecimalFormat ("0.0"))
73         .seriesDots (Dots.circle (5))
74         .seriesStroke (null)
75         .xySeries (results);
76     Plot.write(plot, new File(fileName));
77 }
78
79 /**
80  * Open a GUI for each plot in order to visualize the results of a
81  * previously run set of simulations.
82  *
83  * @param args each plot file generated that you wish to visualize
84  */
85 public static void main(String args[]) {
86     if(args.length < 1) {
87         System.err.println("Must specify at least 1 plot file.");
88         usage();
89     }
90
91     for(int i = 0; i < args.length; i++) {
92         try {
93             Plot plot = Plot.read(args[i]);
94             plot.getFrame().setVisible(true);
95         } catch (ClassNotFoundException e) {
96             System.err.println("Could not deserialize " + args[i]);
97         } catch (IOException e) {
98             System.err.println("Could not open " + args[i]);
99         } catch (IllegalArgumentException e) {
100             System.err.println("Error in file " + args[i]);
101         }
102     }
103 }
104
105 /**
106  * Print the usage message for this program and gracefully exit.
107  */
108 private static void usage() {
109     System.err.println("usage: java PlotHandler <plot-file-1> "+
110         "<(<plot-file-2> <plot-file-3>... etc.)");
111     System.exit(1);
112 }
113 }
114 }
115

```

Simulation.java

```
1 //*****
2 //
3 // File:    Simulation.java
4 // Package: ---
5 // Unit:    Class Simulation
6 //
7 //*****
8
9 import edu.rit.pj2.Loop;
10
11
12
13
14 /**
15  * Class Simulation takes the necessary input to run a specified number of
16  * simulations generating random graphs and averaging the distance over all
17  * the graphs.
18  *
19  * @author Jimi Ford
20  * @version 2-15-2015
21  */
22 public class Simulation {
23
24     // private data members
25     private int v, n;
26     private double p;
27     private Task ref;
28     private long seed;
29     private DoubleVbl.Mean average;
30
31     /**
32      * Construct a simulation object
33      *
34      * @param ref reference to the Task program in order to utilize its
35      *         parallelFor loop
36      * @param seed the seed value for the PRNG
37      * @param v number of vertices in the graph
38      * @param p edge probability of any two vertices being connected
39      * @param n number of simulations to run (or graphs to generate)
40      */
41     public Simulation(Task ref, long seed, int v, double p, int n) {
42         this.v = v;
43         this.p = p;
44         this.n = n;
45         this.seed = seed;
46         this.ref = ref;
47         this.average = new DoubleVbl.Mean();
48     }
49
50
51     /**
52      * Loop through the <I>n</I> simulations and accumulate the distances
53      * between each pair of vertices. The looping in this method is where
54      * most of the computation takes place, so to combat this, a parallel
55      * loop is used.
56      *
57      * @return the results of the <I>n</I> simulations
58      */
59     public SimulationResult simulate() {
60         // run "n" simulations
61         this.ref.parallelFor(0, n - 1).exec(new Loop() {
```


Simulation.java

```
62     Random prng;
63     DoubleVbl.Mean thrAverage;
64
65     @Override
66     public void start() {
67         prng = new Random(seed + rank());
68         thrAverage = threadLocal(average);
69     }
70
71     @Override
72     public void run(int i) {
73         UndirectedGraph.randomGraph(prng, v, p).
74             accumulateDistances(thrAverage);
75     }
76
77     });
78
79     return new SimulationResult(v, p, average.doubleValue());
80 }
81 }
82 }
```

SimulationResult.java

```
1 //*****
2 //
3 // File:    SimulationResult.java
4 // Package: ---
5 // Unit:    Class SimulationResult
6 //
7 //*****
8
9
10 /**
11  * Class SimulationResult is designed to be just a data container for recording
12  * the results of running <I>n</I> simulations given a number of vertices
13  * <I>v</I> and an edge probability <I>p</I>.
14  *
15  * @author Jimi Ford
16  * @version 2-15-2015
17  */
18 public class SimulationResult {
19
20     /**
21      * The average distance between each pair of vertices in <I>n</I> graphs
22      */
23     public final double averageDistance;
24
25     /**
26      * The edge probability used in these <I>n</I> simulations
27      */
28     public final double p;
29
30     /**
31      * The number of vertices in each graph
32      */
33     public final int v;
34
35     /**
36      * Construct a simulation result in order to store the outcome of
37      * a certain number of graphs generated with the given number of
38      * vertices and edge probability.
39      *
40      * @param v number of vertices
41      * @param p edge probability used
42      * @param averageDistance the resulting average distance measured
43      */
44     public SimulationResult(int v, double p, double averageDistance) {
45         this.averageDistance = averageDistance;
46         this.v = v;
47         this.p = p;
48     }
49
50 }
51
```

SimulationResultCollection.java

```
1 //*****
2 //
3 // File:    SimulationResultCollection.java
4 // Package: ---
5 // Unit:    Class SimulationResultCollection
6 //
7 //*****
8
9 /**
10  * Class SimulationResultcollection keeps track of the average distance measured
11  * for each pair of edge probability values and number of vertices. It also
12  * contains the necessary computation to account for using integers as
13  * probabilities, treating them as floating point values ranging from 0 to 1.
14  *
15  * @author Jimi Ford
16  * @version 2-15-2015
17  */
18 public class SimulationResultCollection {
19
20     // private data members
21     private double[][] averages;
22     private int rows, cols;
23
24     /**
25      * The lower bound on number of vertices
26      */
27     public final int vMin;
28
29     /**
30      * The upper bound on number of vertices
31      */
32     public final int vMax;
33
34     /**
35      * The amount to increment the number of vertices by in each set of trials
36      */
37     public final int vInc;
38
39     /**
40      * The scaled lower bound on edge probability
41      */
42     public final int pMin;
43
44     /**
45      * The scaled upper bound on edge probability
46      */
47     public final int pMax;
48
49     /**
50      * The amount to increment the edge probability by in each set of trials
51      */
52     public final int pInc;
53
54     /**
55      * The number of decimal places necessary to convert the edge probability
56      * into an integer. This is in order to combat floating point arithmetic.
57      * One can't just loop from 0 to 1 incrementing by .1 because compounding
58      * error is accumulated on each increment. Integers play nicely when
```

```

59     * incremented.
60     */
61     public final int pExp;
62
63     /**
64     * Construct a simulation result collection. The parameter values
65     * should reflect the values passed into the program through the
66     * command line arguments.
67     *
68     * @param vMin The lower bound on number of vertices
69     * @param vMax The upper bound on number of vertices
70     * @param vInc The amount to increment the number of vertices by in
71     *             each set of trials
72     * @param pMin The scaled lower bound on edge probability
73     * @param pMax The scaled upper bound on edge probability
74     * @param pInc The scaled amount to increment the edge probability by
75     *             in each set of trials
76     * @param pExp The number of decimal places used to convert the edge
77     *             probability into an integer
78     */
79     public SimulationResultCollection (int vMin, int vMax, int vInc,
80         int pMin, int pMax, int pInc, int pExp) {
81         this.vMin = vMin;
82         this.vMax = vMax;
83         this.vInc = vInc;
84         this.pMin = pMin;
85         this.pMax = pMax;
86         this.pInc = pInc;
87         this.pExp = pExp;
88         this.rows = (vMax - vMin + vInc) / vInc;
89         this.cols = (pMax - pMin + pInc) / pInc;
90         this.averages = new double[rows][cols];
91     }
92
93     /**
94     * Add a simulation result to the data matrix.
95     *
96     * @param result the simulation result to record
97     */
98     public void add(SimulationResult result) {
99         int p = p(result.p);
100         int col = col(p);
101         int row = row(result.v);
102         averages[row][col] = result.averageDistance;
103     }
104
105     /**
106     * Get the average distance recorded for a given vertex count
107     * and a scaled edge probability
108     *
109     * @param v the vertex count
110     * @param p the scaled edge probability
111     * @return the average distance recorded for this pair
112     */
113
114     public double get(int v, int p) {
115         int row = row(v);
116         int col = col(p);

```

```

117     return averages[row][col];
118 }
119
120 /**
121  * Get an array of averages for varying edge probabilities and
122  * a given vertex count.
123  *
124  * @param v the vertex count of interest
125  * @return the array of averages for this vertex count
126  */
127 public double[] getAveragesForV(int v) {
128     return averages[row(v)].clone();
129 }
130
131 /**
132  * Convert a vertex value into its associated row value in the
133  * data matrix.
134  *
135  * @param v the vertex count (or number of vertices) to convert
136  * @return the associated row value in the data matrix
137  */
138 private int row(int v) {
139     return (v - vMin) / vInc;
140 }
141
142 /**
143  * Convert an edge probability into a scaled integer in order
144  * to get rid of floating point arithmetic errors.
145  *
146  * @param p the edge probability to convert
147  * @return the scaled integer ranging from pMin to pMax
148  */
149 private int p(double p) {
150     return (int) (Math.round(p * pExp));
151 }
152
153 /**
154  * Convert a scaled edge probability into the associated
155  * column value in the data matrix.
156  *
157  * @param p the scaled edge probability to convert
158  * @return the associated column value in the data matrix
159  */
160 private int col(int p) {
161     return (p - pMin) / pInc;
162 }
163
164 }
165

```

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 Vertex 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, Vertex a, Vertex 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 Vertex other(Vertex current) {
55         if(current == null) return null;
56         return current == a && current.n == a.n ? b : a;
57     }
58 }
```

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     private ArrayList<Vertex> 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) {
40         this.v = v;
41         vertices = new ArrayList<Vertex>(v);
42         edges = new ArrayList<UndirectedEdge>();
43         for(int i = 0; i < v; i++) {
44             vertices.add(new Vertex(i));
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(Vertex start, Vertex goal) {
67      int distance = 0, verticesToProcess = 1, uniqueNeighbors = 0;
68      LinkedList<Vertex> queue = new LinkedList<Vertex>();
69      boolean[] visited = new boolean[v];
70      visited[start.n] = true;
71      Vertex 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.edgeCount(); 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      return 0;
94  }
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  }

```


UndirectedGraph.java

```

117
118 /**
119  * Generate a random graph with a PRNG, a specified vertex count and
120  * an edge probability
121  *
122  * @param prng Prof. Alan Kaminsky's Perfect Random Number Generator
123  * @param v number of vertices to use
124  * @param p edge probability between vertices
125  * @return the randomly generated graph
126  */
127 public static UndirectedGraph randomGraph(Random prng, int v, double p) {
128     UndirectedGraph g = new UndirectedGraph(v);
129     UndirectedEdge edge;
130     Vertex a, b;
131     int edgeCount = 0;
132     for (int i = 0; i < v; i++) {
133         for (int j = i + 1; j < v; j++) {
134             // connect edges
135             // always order it `i` then `j`
136             if(prng.nextDouble() <= p) {
137                 a = g.vertices.get(i);
138                 b = g.vertices.get(j);
139                 edge = new UndirectedEdge(edgeCount++, a, b);
140                 g.edges.add(edge);
141             }
142         }
143     }
144     return g;
145 }
146 }
147

```

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     private 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 edgeCount() {
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
```