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

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import edu.rit.pj2.Task;

/**
 * Class MonteCarlo takes in a seed value for a random number
 * generator, the upper and lower boundaries for the number of vertices in
 * each graph as well as a number to increment by, the upper and lower
 * boundaries for the edge probability as well as a number to increment by,
 * the number of random graphs to generate for each combination of V (vertices)
 * and p (edge probability), and finally, a prefix for naming each plot
 * generated by this program. After checking for valid input, this program
 * loops through each combination of vertices and edge probabilities, running
 * the specified number of simulations on each combination. Each random graph
 * (or simulation) is generated by looking at every possible pair of vertices,
 * generating a random floating point between 0 and 1, and marking these
 * vertices with an edge connecting them if the random value is less than or
 * equal to the specified edge probability (for that unique graph). In each
 * simulation, the distance values of each graph are calculated with a breadth
 * first search from vertex A to vertex B using the depth of the search as the
 * distance from A to B.
 *
 * @author  Jimi Ford
 * @version 2-15-2015
 */
public class MonteCarlo extends Task {

    // Private constants
    private static final String[] arguments = {
        "<seed>",
        "<min_v>",
        "<max_v>",
        "<v_grain>",
        "<min_p>",
        "<max_p>",
        "<p_grain>",
        "<num_simulations>",
        "<optional plotfile prefix>"
    };

    private static final int
        SEED = 0,
        MIN_VERTICES = 1,
        MAX_VERTICES = 2,
        VERTEX_GRANULARITY = 3,
        MIN_P = 4,
        MAX_P = 5,
        P_GRANULARITY = 6,
        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 &lt;seed&gt; &lt;min_v&gt; &lt;max_v&gt;
69       * &lt;v_grain&gt; &lt;min_p&gt; &lt;max_p&gt; &lt;p_grain&gt;
70       * &lt;num_simulations&gt; &lt;optional plotfile prefix&gt;
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 pMin = (int) (Math.round(minP * exp));
185         final int pInc = (int) (Math.round(pGrain * exp));
186         pGrainStr = null;
187 
188 
189 
190         SimulationResultCollection results = new SimulationResultCollection(
191                 minVertices, maxVertices, vertexGranularity, pMin, pMax, pInc, exp);
192 
193         // loop through number of vertices
194         for(int vCount = minVertices; vCount <= maxVertices;
195                 vCount += vertexGranularity) {
196             // loop through edgeProbability
197             for(int p = pMin; p <= pMax; p += pInc) {
198                 double prob = p / (double) exp;
199                 // loop through each simulation
200                 results.add(new Simulation(this, seed, vCount, prob,
201                         numSimulations).simulate());
202             }
203             try {
204                 new PlotHandler(plotFilePrefix, results, vCount).write();
205             } catch (IOException e) {
206                 System.err.println("Error writing file for v="+vCount);
207             }
208         }
209 
210         StringBuilder builder = new StringBuilder();
211         for(int p = 0; p<= pMax; p+= pInc) {
212             builder.append(", " + (p / ((double) exp)));
213         }
214         builder.append('\n');
215         for(int v = minVertices; v<= maxVertices; v+= vertexGranularity) {
216             builder.append(v + ", ");
217             for(int p = pMin; p <= pMax; p+=pInc) {
218                 builder.append(results.get(v,p)+", ");
219             }
220             builder.append('\n');
221         }
222         PrintWriter tableWriter = null;
223         final String tableSuffix = "-table.csv";
224         try {
225             tableWriter = new PrintWriter(plotFilePrefix + tableSuffix);
226             tableWriter.print(builder.toString());
227         } catch (FileNotFoundException e) {
228             System.err.println("Error writing table data to file \""+
229                 plotFilePrefix + tableSuffix +"\"");
230         } finally {
231             if(tableWriter != null) tableWriter.close();
232         }
```

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

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

import java.io.File;

/**
 * Class PlotHandler is the delegate for dealing with visualizing the data
 * generated by the "number crunching" program, MonteCarlo. Its purpose is to
 * be instantiated in MonteCarlo with the data to plot, where the write()
 * method should then be called. Running this program and specifying in
 * the command line arguments the plot files previously generated will
 * open a graphical representation of these plots for each file.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 *
 */
public class PlotHandler {

    // private data members
    private final String fileName;
    private final int v;
    private final SimulationResultCollection collection;

    /**
     * Construct a new plot handler that plots average distances for a fixed
     * vertex count v, while varying the edge probability p
     *
     * @param   plotFilePrefix prefix to be used in the name of
     *          the plot file
     * @param   collection collection of results of the finished set of
     *          simulations.
     * @param v number of vertices used in each simulation
     */
    public PlotHandler(String plotFilePrefix,
            SimulationResultCollection collection, int v) {
        fileName = plotFilePrefix + "-V-" + v + ".dwg";
        this.v = v;
        this.collection = collection;
    }

    /**
     * Save the plot information into a file to visualize by running
     * the main method of this class
     *
     * @throws IOException if it can't write to the file specified
     */
    public void write() throws IOException {
        ListXYSeries results = new ListXYSeries();
        double[] values = collection.getAveragesForV(v);
        for(int i = 0, p = collection.pMin; i < values.length; i++,
                p += collection.pInc) {
            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              }
100         }
101
102     }
103
104     /**
105      * Print the usage message for this program and gracefully exit.
106      */
107     private static void usage() {
108         System.err.println("usage: java PlotHandler <plot-file-1> "+
109                 "(<plot-file-2> <plot-file-3>... etc.)");
110         System.exit(1);
111     }
112 }
113
```

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

import edu.rit.pj2.Loop;

/**
 * Class Simulation takes the necessary input to run a specified number of
 * simulations generating random graphs and averaging the distance over all
 * the graphs.
 *
 * @author  Jimi Ford
 * @version 2-15-2015
 */
public class Simulation {

    // private data members
    private int v, n;
    private double p;
    private Task ref;
    private long seed;
    private DoubleVbl.Mean average;

    /**
     * Construct a simulation object
     *
     * @param   ref reference to the Task program in order to utilize its
     *          parallelFor loop
     * @param seed the seed value for the PRNG
     * @param v number of vertices in the graph
     * @param p edge probability of any two vertices being connected
     * @param n number of simulations to run (or graphs to generate)
     */
    public Simulation(Task ref, long seed, int v, double p, int n) {
        this.v = v;
        this.p = p;
        this.n = n;
        this.seed = seed;
        this.ref = ref;
        this.average = new DoubleVbl.Mean();
    }


    /**
     * Loop through the <I>n</I> simulations and accumulate the distances
     * between each pair of vertices. The looping in this method is where
     * most of the computation takes place, so to combat this, a parallel
     * loop is used.
     *
     * @return the results of the <I>n</I> simulations
     */
    public SimulationResult simulate() {
        // run "n" simulations
        this.ref.parallelFor(0, n - 1).exec(new Loop() {
```

```
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
```

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


/**
 * Class SimulationResult is designed to be just a data container for recording
 * the results of running <I>n</I> simulations given a number of vertices
 * <I>v</I> and an edge probability <I>p</I>.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class SimulationResult {

    /**
     * The average distance between each pair of vertices in <I>n</I> graphs
     */
    public final double averageDistance;

    /**
     * The edge probability used in these <I>n</I> simulations
     */
    public final double p;

    /**
     * The number of vertices in each graph
     */
    public final int v;

    /**
     * Construct a simulation result in order to store the outcome of
     * a certain number of graphs generated with the given number of
     * vertices and edge probability.
     *
     * @param v number of vertices
     * @param p edge probability used
     * @param averageDistance the resulting average distance measured
     */
    public SimulationResult(int v, double p, double averageDistance) {
        this.averageDistance = averageDistance;
        this.v = v;
        this.p = p;
    }

}

```

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

/**
 * Class SimulationResultcollection keeps track of the average distance measured
 * for each pair of edge probability values and number of vertices. It also
 * contains the necessary computation to account for using integers as
 * probabilities, treating them as floating point values ranging from 0 to 1.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class SimulationResultCollection {

    // private data members
    private double[][] averages;
    private int rows, cols;

    /**
     * The lower bound on number of vertices
     */
    public final int vMin;

    /**
     * The upper bound on number of vertices
     */
    public final int vMax;

    /**
     * The amount to increment the number of vertices by in each set of trials
     */
    public final int vInc;

    /**
     * The scaled lower bound on edge probability
     */
    public final int pMin;

    /**
     * The scaled upper bound on edge probability
     */
    public final int pMax;

    /**
     * The amount to increment the edge probability by in each set of trials
     */
    public final int pInc;

    /**
     * The number of decimal places necessary to convert the edge probability
     * into an integer. This is in order to combat floating point arithmetic.
     * One can't just loop from 0 to 1 incrementing by .1 because compounding
     * 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    /**
107     * Get the average distance recorded for a given vertex count
108     * and a scaled edge probability
109     *
110     * @param v the vertex count
111     * @param p the scaled edge probability
112     * @return the average distance recorded for this pair
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 
```

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

/**
 * Class UndirectedEdge represents an edge in a graph that connects two
 * vertices. It's important to note that the edge does not have a direction nor
 * weight.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class UndirectedEdge {

    // private data members
    private Vertex a, b;

    // future projects may rely on a unique identifier for an edge
    private final int id;

    /**
     * Construct an undirected edge
     * @param id a unique identifier to distinguish between other edges
     * @param a one vertex in the graph
     * @param b another vertex in the graph not equal to <I>a</I>
     */
    public UndirectedEdge(int id, Vertex a, Vertex b) {
        this.id = id;
        // enforce that a.n is always less than b.n
        if(a.n < b.n) {
            this.a = a;
            this.b = b;
        } else if(b.n < a.n) {
            this.a = b;
            this.b = a;
        } else {
            throw new IllegalArgumentException("Cannot have self loop");
        }
        this.a.addEdge(this);
        this.b.addEdge(this);
    }

    /**
     * Get the <I>other</I> vertex given a certain vertex connected to
     * this edge
     *
     * @param current the current vertex
     * @return the other vertex connected to this edge
     */
    public Vertex other(Vertex current) {
        if(current == null) return null;
        return current == a && current.n == a.n ? b : a;
    }
}
```

```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;
    private ArrayList<Vertex> vertices;
    private int v;

    // Prevent construction
    private UndirectedGraph() {

    }

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

    /**
     * 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));
    }

```

```java
    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the reference to the start vertex
     * @param goal the reference to the goal vertex
     * @return the minimum distance between the two vertices
     */
    private int BFS(Vertex start, Vertex goal) {
        int distance = 0, verticesToProcess = 1, uniqueNeighbors = 0;
        LinkedList<Vertex> queue = new LinkedList<Vertex>();
        boolean[] visited = new boolean[v];
        visited[start.n] = true;
        Vertex current, t2;
        queue.add(start);
        while(!queue.isEmpty()) {
            current = queue.removeFirst();
            if(current.equals(goal)) {
                return distance;
            }
            for(int i = 0; i < current.edgeCount(); i++) {
                t2 = current.getEdges().get(i).other(current);
                if(!visited[t2.n]) {
                    visited[t2.n] = true;
                    queue.add(t2);
                    uniqueNeighbors++;
                }
            }
            verticesToProcess--;
            if(verticesToProcess <= 0) {
                verticesToProcess = uniqueNeighbors;
                uniqueNeighbors = 0;
                distance++;
            }

        }
        return 0;
    }

    /**
     * Accumulate the distances of each pair of vertices into
     * a "running total" to be averaged
     *
     * @param thrLocal the reference to the "running total"
     * Prof. Alan Kaminsky's library handles averaging this
     * accumulated value.
     */
    public void accumulateDistances(DoubleVbl.Mean thrLocal) {
        for(int i = 0; i < v; i++) {
            for(int j = i + 1; j < v; j++) {
                thrLocal.accumulate(BFS(i, j));
            }
        }
    }

    /**
     * Generate a random graph with a PRNG, a specified vertex count and
     * an edge probability
     *
```

```java
     * @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) {
        UndirectedGraph g = new UndirectedGraph(v);
        UndirectedEdge edge;
        Vertex 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;
    }
}

```

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

import java.util.ArrayList;

/**
 * Class Vertex represents a single vertex in a graph. Vertices can be connected
 * to other vertices through undirected edges.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class Vertex {

    // private data members
    private ArrayList<UndirectedEdge> edges = new ArrayList<UndirectedEdge>();

    /**
     * The unique identifier for this vertex
     */
    public final int n;

    /**
     * Construct a vertex with a unique identifier <I>n</I>
     *
     * @param n the unique identifier to distinguish this vertex from
     *          all other vertices in the graph
     */
    public Vertex(int n) {
        this.n = n;
    }

    /**
     * Get the number of edges connected to this vertex
     *
     * @return the number of edges connected to this vertex
     */
    public int edgeCount() {
        return edges.size();
    }

    /**
     * Get the reference to the collection of edges connected to
     * this vertex.
     *
     * @return the reference to the collection of edges
     */
    public ArrayList<UndirectedEdge> getEdges() {
        return this.edges;
    }

    /**
     * Add an edge to this vertex
```

```java
     *
     * @param e the edge to add
     */
    public void addEdge(UndirectedEdge e) {
        this.edges.add(e);
    }

    /**
     * Compare another object to this one
     *
     * @param o the other object to compare to this one
     * @return true if the other object is equivalent to this one
     */
    public boolean equals(Object o) {
        if( !(o instanceof Vertex)) {
            return false;
        }
        if(o == this) {
            return true;
        }
        Vertex casted = (Vertex) o;

        return casted.n == this.n;
    }
}

```