Jimi Ford
CSCI 351-01 Data Communications and Networks
Programming Project 1 Report

1. Program Inputs, Program Objective, Program Outputs

MonteCarlo.java

This program 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.

PlotHandler.java

This program takes in a list of plot files generated by MonteCarlo.java. After the program ensures proper use of PlotHandler, it deserializes the files into objects that contain the plot data, and displays each plot in its own window. From there, each window opened has the capability of adjusting the plot size, formatting, textual attributes, and saving the plot to an image. Thanks to Prof. Alan Kaminsky's PJ2 library, PlotHandler is a very small program since all of the visual/GUI code is provided by PJ2.

2. Exact Command Line

MonteCarlo.java

note: This program must be invoked by Prof. Alan Kaminsky's Parallel Java 2 library with the CLASSPATH environment variable set as per Prof. Alan Kaminsky's instructions on this.

```
usage: java pj2 MonteCarlo <seed> <min_v> <max_v> <v_grain> <min_p>
<max_p> <p_grain> <num_simulations> <optional plotfile prefix>
```

`<seed>` – Seed value for Prof. Alan Kaminsky's PRNG
`<min_v>` – Lower bound (inclusive) for number of vertices in random graphs, *V*

`<max_v>` – Upper bound (inclusive) for number of vertices in random graphs, *V*
`<v_grain>` – Vertex granularity: amount to increment the number of vertices in the graph by for each round of *n* simulations
`<min_p>` – Lower bound (inclusive) for edge probability, *p*
`<max_p>` – Upper bound (inclusive) for edge probability, *p*
`<p_grain>` – Edge probability granularity: amount to increment the edge probability by for each round of *n* simulations
`<num_simulations>` – Number of random graphs to generate *per V,p* combination, *n*
`<optional plotfile prefix>` – (Optional) Prefix for file output (default = "plot")

PlotHandler.java

note: This program allows for any number (greater than 0) of plot files to be specified in the command line arguments.

```
usage:
java PlotHandler <plot-file-1> (<plot-file-2> <plot-file-3>... etc.)
```

`<plot-file-1>` – The plot file (generated by MonteCarlo) to visualize in an X-Y plot

3. Source Code *(See Appendix A for project's source code)*

---

4. For a given number of vertices *V*, what happens to the average distance as the edge probability increases, and why is this happening?

As evidenced by the data gathered in this report, *(Q5. Supporting Data)*, we see that the average distance between nodes in random graphs decreases and approaches 1 as the edge probability *p*, increases. It's important to note that the average distance doesn't continually decrease; it increases for a short period up to a single global maximum and then decreases and converges to 1. This is because as *p* increases, so do the number of edges in a graph. The more edges there are in a graph, the closer vertices become. For instance, let's say that there exists some path from vertex *A* to *B* to *C* but *A* and *C* are not immediately connected by a single edge. The more edges we add to this graph, the higher the chances are that we will end up with a path connecting vertex *A* and *C* directly, making the distance between the two vertices 1.

5. Supporting Data

Commands used:

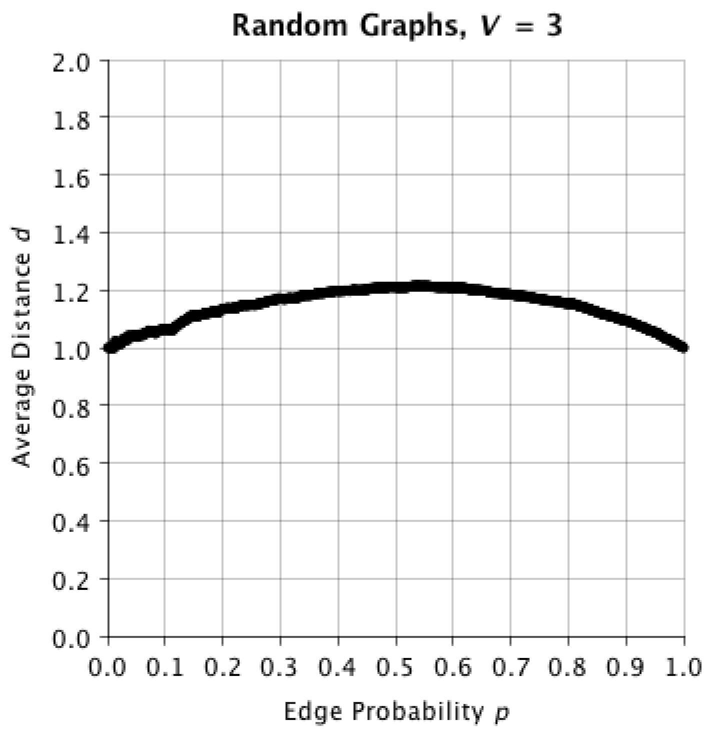
This first command will run the simulations and save *.dwg* files as well as a *.csv* file with the prefix *"plot-Q4"*.

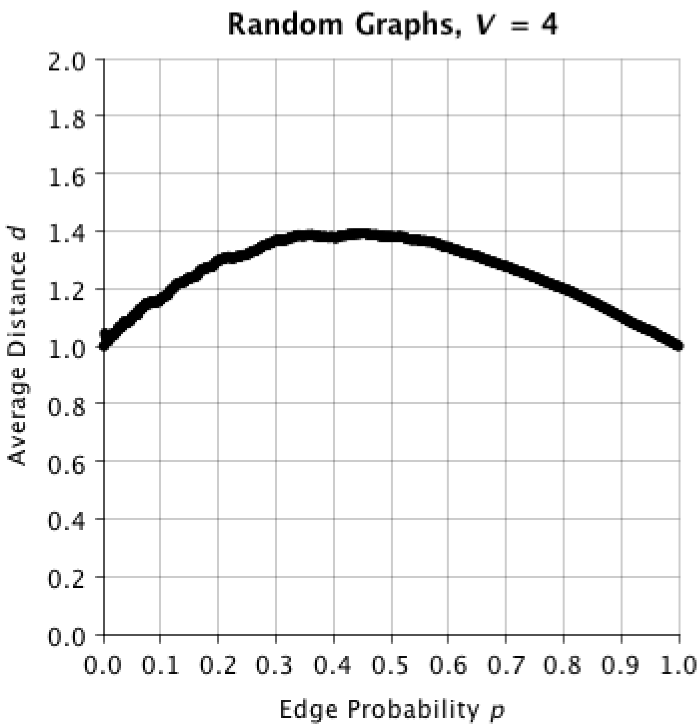
```
java pj2 MonteCarlo 100123456789 3 10 1 0 1 .001 1000 plot-Q4
(expected runtime 2-3 minutes on quad-core)
```

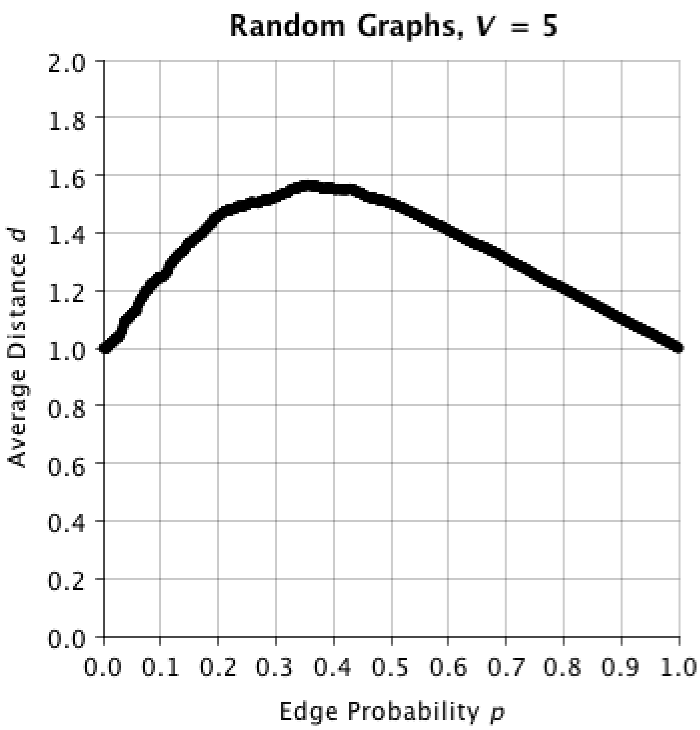
*note: In order to save you time while grading, 1000 simulations are used for each combination of **p** and **V**. As a result, the plots produced aren't extremely smooth but are smooth enough to convey the general trend of varying **p** and holding **V** constant.*

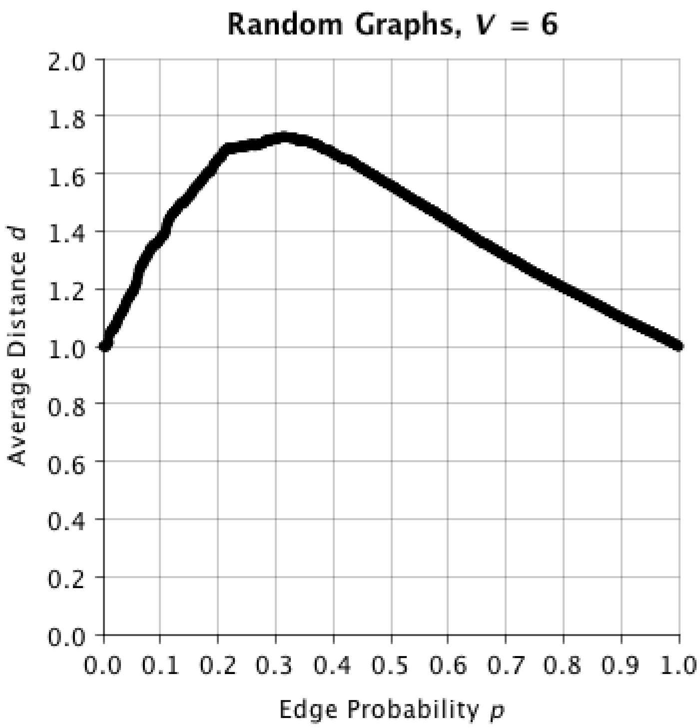
Then after generating the *.dwg* files, you must run the following command to visualize the plots.

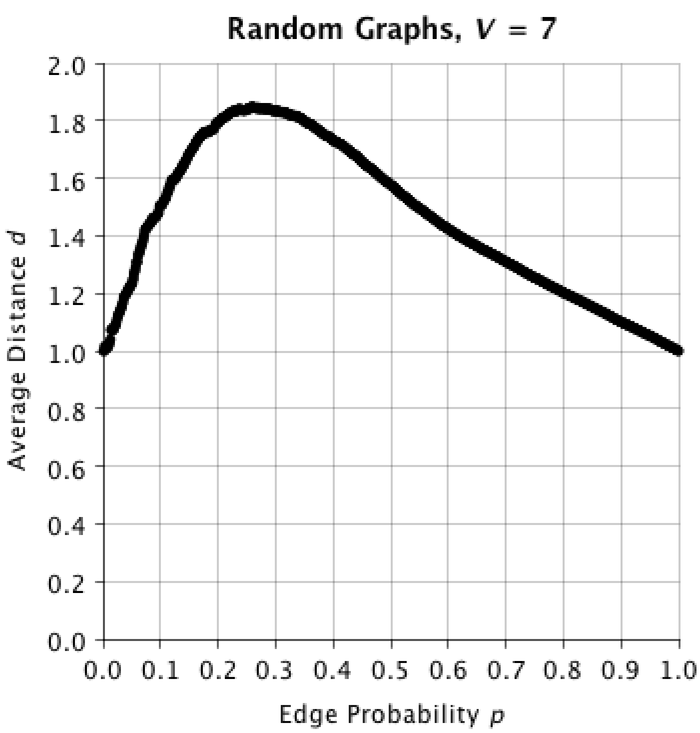
```
java PlotHandler plot-Q4-V-3.dwg plot-Q4-V-4.dwg plot-Q4-V-5.dwg
plot-Q4-V-6.dwg plot-Q4-V-7.dwg plot-Q4-V-8.dwg plot-Q4-V-9.dwg
plot-Q4-V-10.dwg
```

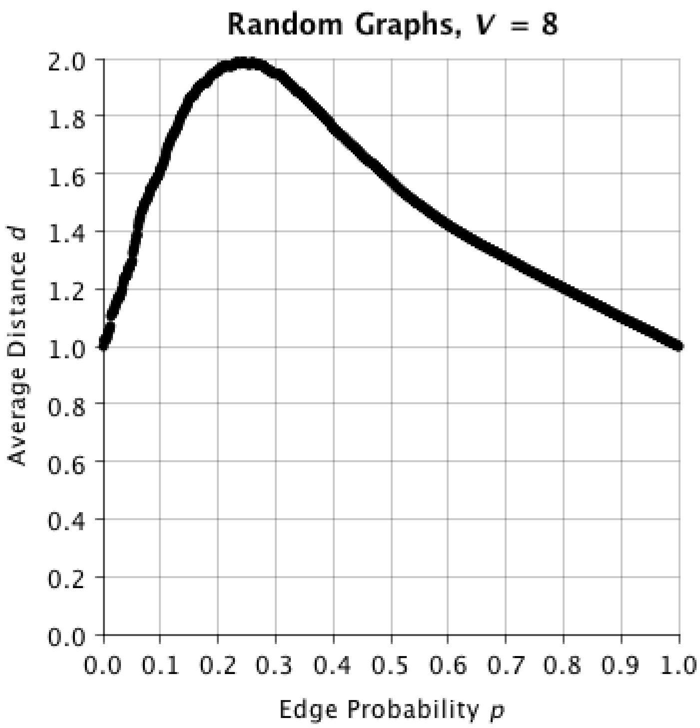



Random Graphs, V = 4
Average Distance d
Edge Probability p
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

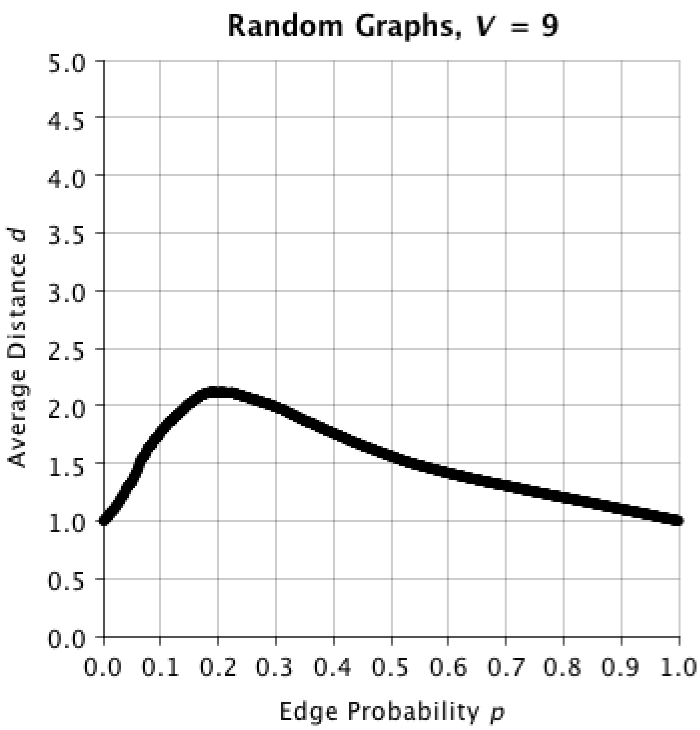

Random Graphs, V = 5
Average Distance d
Edge Probability p
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0


Random Graphs, V = 6
Average Distance d
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Edge Probability p

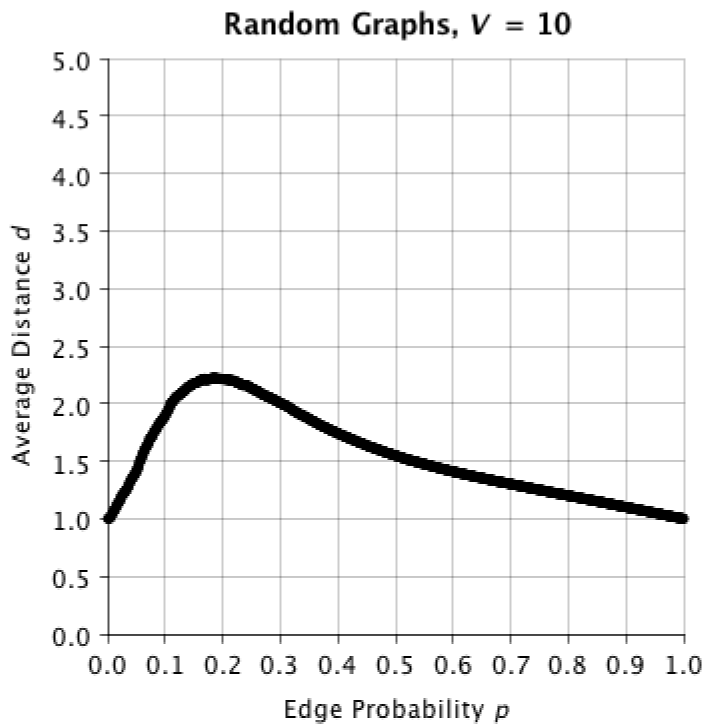

Random Graphs, V = 7
Average Distance d
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Edge Probability p



Random Graphs, V = 8
Average Distance d
Edge Probability p
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0



Random Graphs, V = 9
Average Distance d
Edge Probability p
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0



note: The table included in this section is generated with a different command. This is because in order to produce dense plot data (above), a very low increment value for ***p*** *must be used (.001) and by using this value, 1000 columns are generated in the table. To produce a table that will fit in this report, run the command listed below.*

```
java pj2 MonteCarlo 100123456789 3 10 1 0 1 .1 1000 plot-Q4a
```

This will generate a table (with fewer columns) in CSV format named “plot-Q4a-table.csv”.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
3	1.065217391	1.120171674	1.15450237	1.19047619	1.202253219	1.209094962	1.180533752	1.144140906	1.09494744	1
4	1.163027656	1.287384615	1.363950807	1.377513452	1.38004965	1.343414634	1.282342502	1.20648667	1.109998331	1
5	1.268370607	1.441808272	1.532845309	1.538985939	1.486505766	1.403758911	1.308493263	1.209061684	1.1042	1
6	1.385915493	1.632208158	1.708306504	1.669605339	1.558211983	1.435005467	1.309475942	1.202669336	1.100066667	1
7	1.493717438	1.789776499	1.822219195	1.730421605	1.573167097	1.42658118	1.304331234	1.200142857	1.099666667	1
8	1.620499182	1.947543113	1.93952269	1.758263198	1.574984556	1.419672013	1.301297402	1.199642857	1.100464286	1
9	1.780844835	2.095249625	1.988327227	1.760451228	1.56220631	1.412558424	1.300205601	1.199361111	1.099833333	1
10	1.899729811	2.203152181	2.003951685	1.747028088	1.549112822	1.408770643	1.300148919	1.199466667	1.100333333	1

6. For a given edge probability *p*, what happens to the average distance as the number of vertices *V* increases, and why is this happening?

For this answer, we will need to direct our attention to both *Q5. Supporting Data*, and *Q7. Supporting Data*. Observe the peak value in each plot and how it occurs at a smaller and smaller value for *p* as *V* increases. After this peak value, the average distance for every graph converges to 1. For given *p* values that are greater than the *p* value at which point the peak distance occurs, even as *V* increases, the average distance decreases. For given *p* values that are less than the *p* value at which point the peak distance occurs, as *V* increases, so does the average distance *(see Q5. Supporting Data)*. This is happening because for each combination of *p* and *V*, there is a point where there is optimal (maximal) distance. As more vertices are added to a graph, the upper bound on the maximum possible distance between two vertices is increased since the furthest two vertices can be from each other is *V* - 1. Once we reach that maximal distance for the graph, the distance must converge to 1 as discussed in the previous answer. Since the maximum possible distance increases with more vertices, the maximal (or peak) values for distance increase and have a steeper slope to climb to converge to 1. This explains why average distances at values for *p* to the right of this peak will decrease as *V* increases, because they approach the asymptote more rapidly.

7. Supporting Data

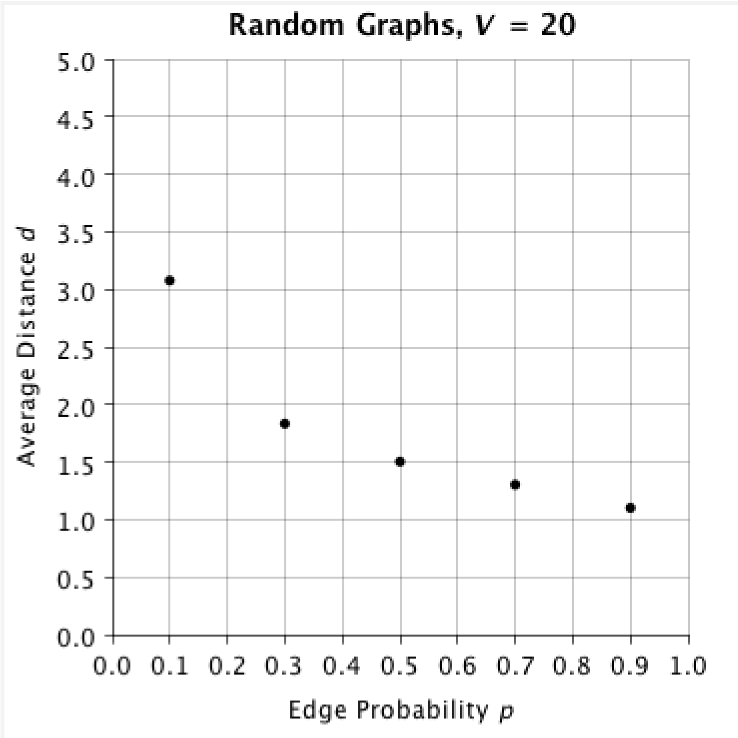
Commands used:

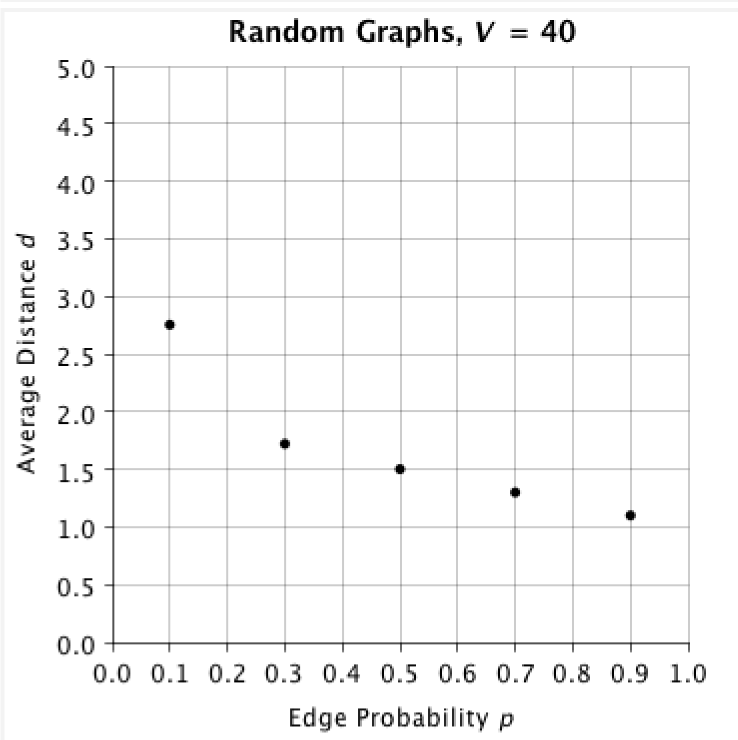
```
java pj2 MonteCarlo 100123456789 20 100 20 .1 .9 .2 100 plot-Q6
(expected runtime 2-3 minutes on quad-core)
```

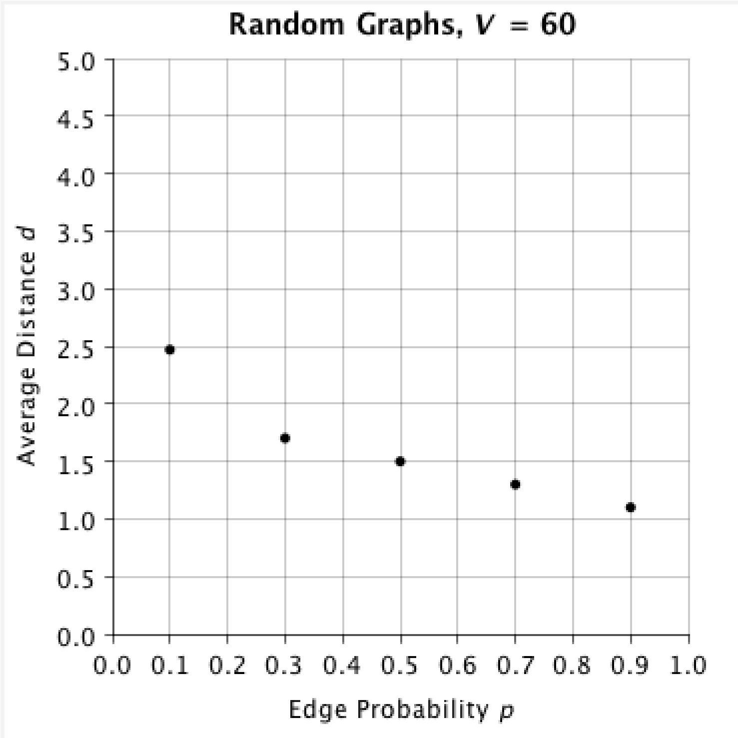
plot-Q6-table.csv:

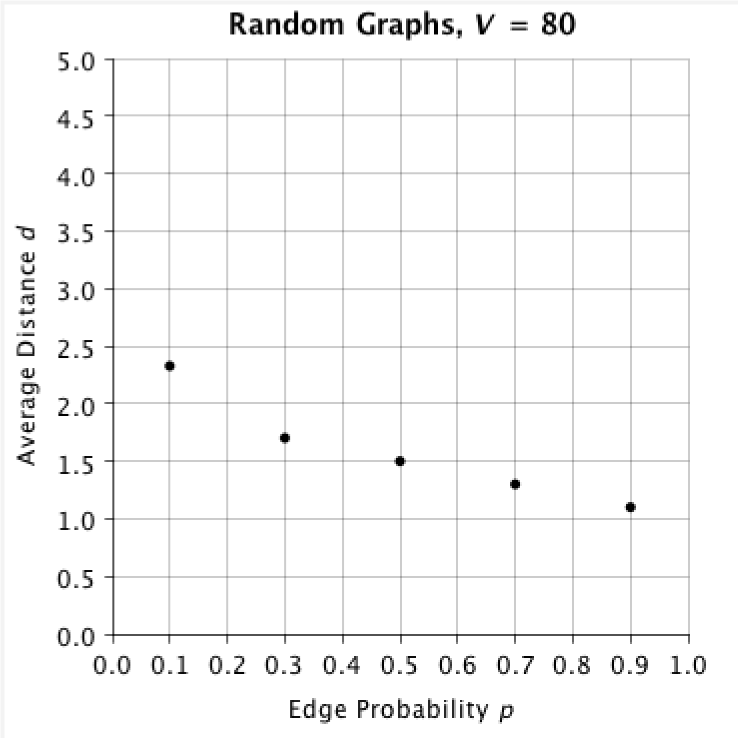
	0.1	0.3	0.5	0.7	0.9
20	3.077801449	1.834329542	1.504263158	1.303210526	1.101368421
40	2.754609476	1.720628205	1.501384615	1.300435897	1.100769231
60	2.469624655	1.702016949	1.499971751	1.299429379	1.100107345
80	2.326933494	1.700822785	1.500022152	1.299648734	1.100221519
100	2.242674747	1.700353535	1.500393939	1.300321212	1.100046465

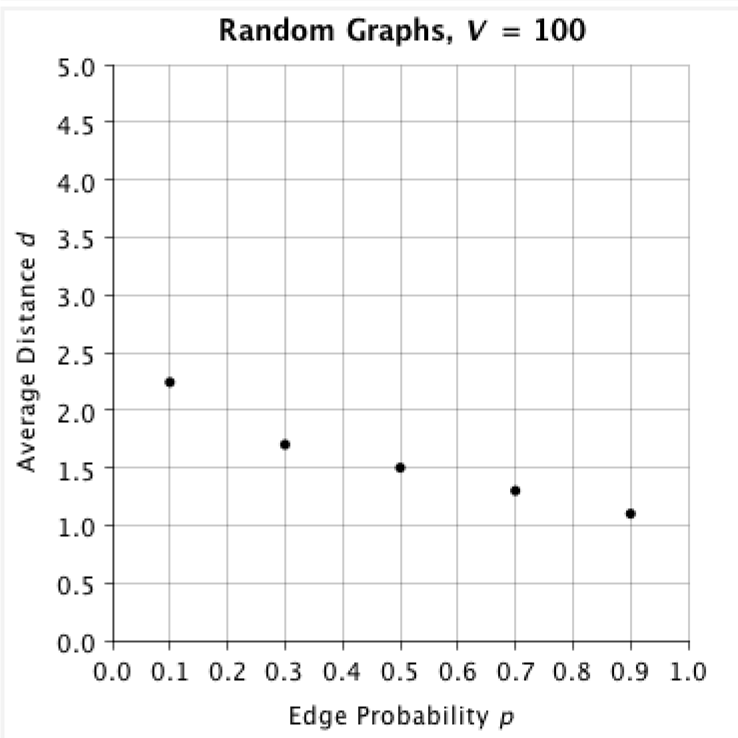
```
java PlotHandler plot-Q6-V-20.dwg plot-Q6-V-40.dwg plot-Q6-V-60.dwg
plot-Q6-V-80.dwg plot-Q6-V-100.dwg
```






Random Graphs, V = 60
Average Distance d
Edge Probability p
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0


Random Graphs, V = 80
Average Distance d
Edge Probability p
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0


Random Graphs, V = 100
Average Distance d
Edge Probability p
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

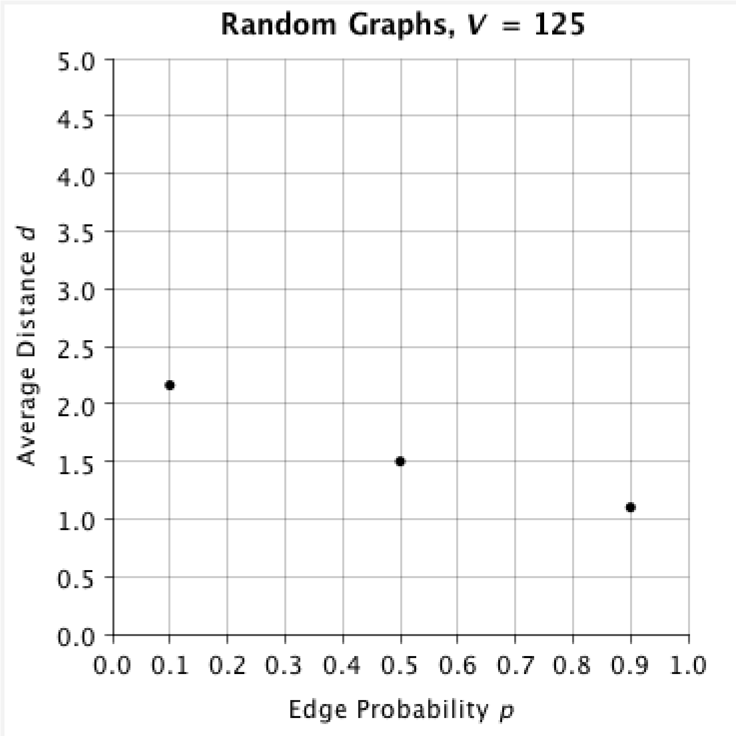
```
java pj2 MonteCarlo 100123456789 125 200 25 .1 .9 .4 10 plot-Q6a
(expected runtime: ~1 minute on quad-core)
```

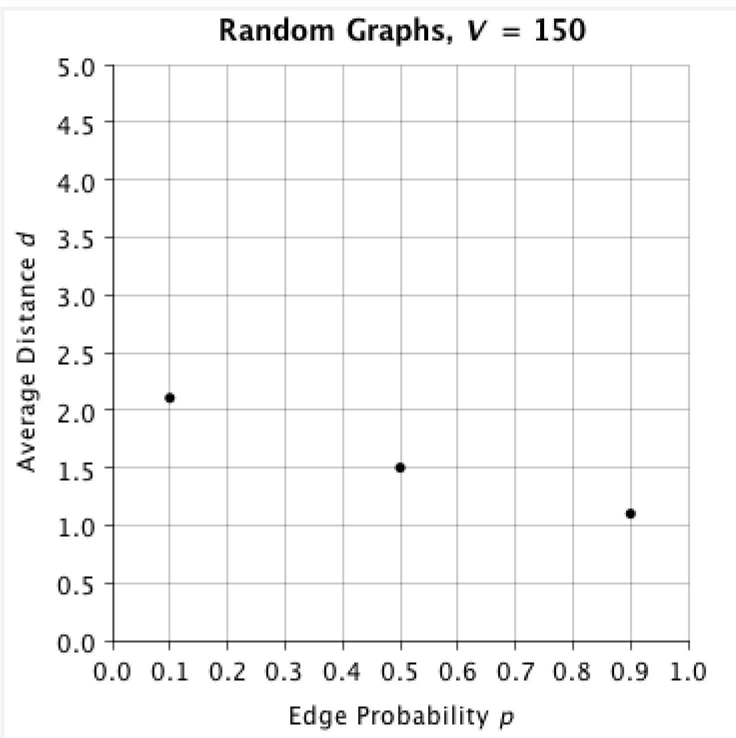
note: To save time, only 10 trials are used to obtain values for average distance with a large number of vertices like 200. Higher numbers used will yield more accurate results, but will take substantially longer to complete.

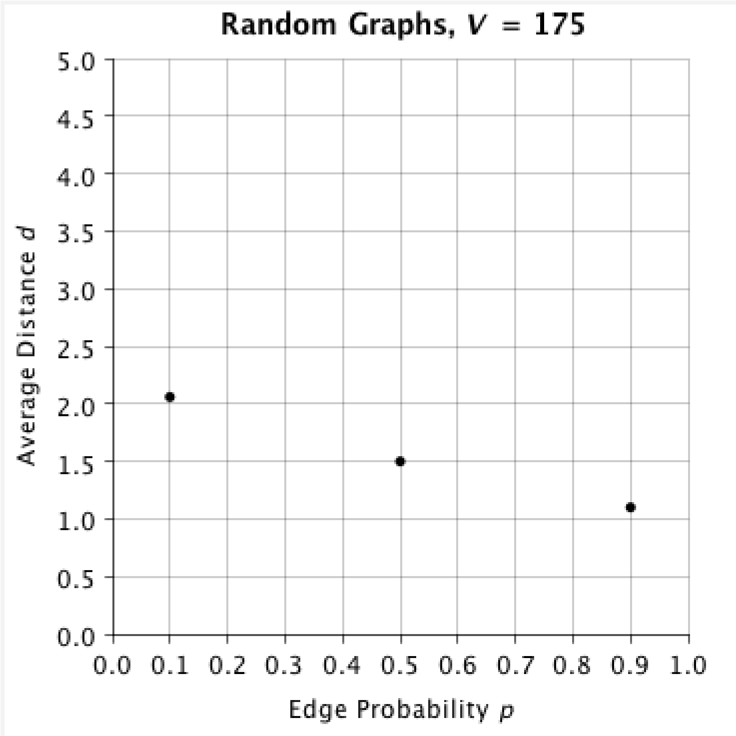
plot-Q6a-table.csv:

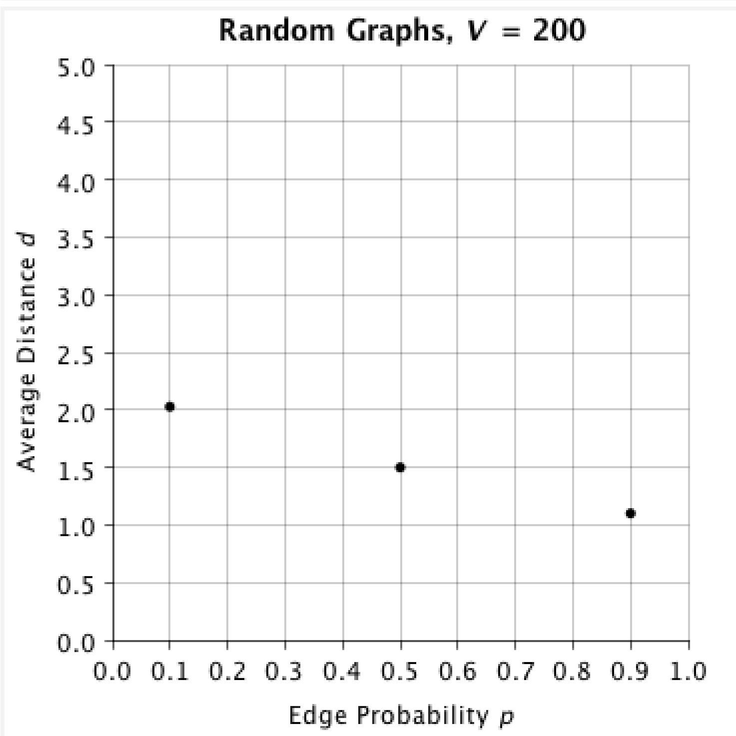
	0.1	0.5	0.9
125	2.165367742	1.500258065	1.101212903
150	2.103740492	1.498765101	1.100384787
175	2.056604269	1.499691297	1.100098522
200	2.022909548	1.500884422	1.100291457

```
java PlotHandler plot-Q6a-V-125.dwg plot-Q6a-V-150.dwg
plot-Q6a-V-175.dwg plot-Q6a-V-200.dwg
```


Random Graphs, V = 125
Average Distance d
Edge Probability p
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0


Random Graphs, V = 150
Average Distance d
Edge Probability p
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0


Random Graphs, V = 175
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Average Distance d
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Edge Probability p


Random Graphs, V = 200
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Average Distance d
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Edge Probability p

8. What I learned

This project taught me how to explore properties of random graphs that aren't necessarily obvious until thoroughly studied. The method of thorough study that I utilized is called the Monte Carlo method or a Monte Carlo simulation. This project helped me realize the value of using a Monte Carlo simulation for problems that are difficult to theorize or express mathematically. If I were a mathematician, I would have an extremely helpful tool to guide research towards finding a formula to represent average distance for a random graph as a function of *p* and *V*. I was surprised to learn that even in a graph with 100 vertices, the majority of edge probabilities produce distances that are less than 3 hops away (on average). For every number of vertices in a random graph, there is a value for *p* that produces a global maximum for the average distance in the graph. This global maximum increases in value as *V* increases and shifts to the left toward edge probability 0. In the first iteration of this project, I wrote the program single-threadedly. After I realized that it was going to take a long time to get accurate results with higher numbers for vertices, and calculating a greater number of edge probabilities, I decided to implement a parallel loop to harness the full power of what the computer was capable of. I learned that programming with PJ2 speeds up runtime substantially. On a quad core, the runtime was decreased by a factor of at least 3, and on an 8-core computer, the runtime was cut by 6 (at the very least). Also I finally learned how to incorporate PJ2 into eclipse! Thanks, Shane Hale.

Appendix A)

Source code

```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,
```

```
        PLOT_FILE_PREFIX = 8;

    /**
     * MonteCarlo's main method to be invoked by Prof. Alan Kaminsky's
     * Parallel Java 2 library.
     *
     * @param args command line arguments
     *
     * <P>
     * usage: java pj2 MonteCarlo &lt;seed&gt; &lt;min_v&gt; &lt;max_v&gt;
     * &lt;v_grain&gt; &lt;min_p&gt; &lt;max_p&gt; &lt;p_grain&gt;
     * &lt;num_simulations&gt; &lt;optional plotfile prefix&gt;
     * <P>
     */
    public void main(String[] args) {
        if(args.length != 8 && args.length != 9) {
            usage();
        }

        long seed = 0;
        int minVertices = 0, maxVertices = 0, vertexGranularity = 0,
                numSimulations = 0;
        double pGrain = 0, minP = 0, maxP = 0;

        try {
            seed = Long.parseLong(args[SEED]);
        } catch (NumberFormatException e) {
            displayError(
                    String.format("Argument %1s must be numeric and between %2d "+
                    "and %3d inclusive.\n", arguments[SEED],
                Long.MIN_VALUE, Long.MAX_VALUE));
        }

        try {
            minVertices = Integer.parseInt(args[MIN_VERTICES]);
            if(minVertices < 1) throw new NumberFormatException();
        } catch (NumberFormatException e) {
            displayError(
                String.format("Argument %1s must be numeric and between 1 "+
                        "and %2d inclusive.\n", arguments[MIN_VERTICES],
                        Integer.MAX_VALUE));
        }

        try {
            maxVertices = Integer.parseInt(args[MAX_VERTICES]);
            if(maxVertices < minVertices)
                displayError(String.format(
                    "Argument %1s must be greater than or equal to %2s.\n",
                    arguments[MAX_VERTICES], arguments[MIN_VERTICES]));
        } catch (NumberFormatException e) {
            displayError(String.format(
                "Argument %1s must be numeric and between 1 and %2d inclusive.\n",
                    arguments[MAX_VERTICES], Integer.MAX_VALUE));
        }

        try {
            vertexGranularity = Integer.parseInt(args[VERTEX_GRANULARITY]);
            if(vertexGranularity < 1) throw new NumberFormatException();
```

```java
        } catch (NumberFormatException e) {
            displayError(String.format(
                "Argument %1s must be numeric and between 1 and %2d inclusive.\n",
                    arguments[VERTEX_GRANULARITY], Integer.MAX_VALUE));
        }

        try {
            minP = Double.parseDouble(args[MIN_P]);
            if(minP < 0 || minP > 1) throw new NumberFormatException();
        } catch (NumberFormatException e) {
            displayError(String.format(
                    "Argument %1s must be numeric and between "+
                    "0 inclusive and 1 inclusive.\n",
                    arguments[MIN_P]));
        }

        try {
            maxP = Double.parseDouble(args[MAX_P]);
            if(maxP < minP)
                displayError(String.format(
                        "Argument %1s must be greater than or equal to %2s.\n",
                        arguments[MAX_P], arguments[MIN_P]));
            if(maxP > 1) throw new NumberFormatException();
        } catch (NumberFormatException e) {
            displayError(String.format(
                "Argument %1s must be numeric and between "+
                "0 inclusive and 1 inclusive.\n",
                    arguments[MAX_P]));
        }

        try {
            pGrain = Double.parseDouble(args[P_GRANULARITY]);
            if(pGrain <= 0 || pGrain > 1)
                throw new NumberFormatException();
        } catch (NumberFormatException e) {
            displayError(String.format(
                "Argument %1s must be numeric and between "+
                "0 exclusive and 1 inclusive.\n",
                    arguments[P_GRANULARITY]));
        }

        try {
            numSimulations = Integer.parseInt(args[NUMBER_OF_SIMULATIONS]);
            if(numSimulations < 1) throw new NumberFormatException();
        } catch (NumberFormatException e) {
            displayError(String.format(
                "Argument %1s must be numeric and between 1 and %2d inclusive.\n",
                    arguments[NUMBER_OF_SIMULATIONS], Integer.MAX_VALUE));
        }

        // store file prefix
        final String plotFilePrefix = args.length == 9 ?
                args[PLOT_FILE_PREFIX] : "plot";

        String pMinStr = Double.toString(minP);
        String pMaxStr = Double.toString(maxP);
        String pGrainStr = Double.toString(pGrain);
        final int sigFig =
```

```java
                Math.max(Math.max(
                        pGrainStr.length() - pGrainStr.indexOf('.') - 1,
                        pMaxStr.length() - pMaxStr.indexOf('.') - 1),
                        pMinStr.length() - pMinStr.indexOf('.') - 1);
        int exp = 1;
        for(int i = 0; i < sigFig; i++) {
            exp *= 10;
        }
        final int pMax = (int) (Math.round(maxP * exp));
        final int pInc = (int) (Math.round(pGrain * exp));
        // if 0 is the lower bound, set pMin to the next "step" of edge probability
        // which is pInc
        final int pMin = ((int) (Math.round(minP * exp))) == 0 ?
                pInc : ((int) (Math.round(minP * exp)));
        pGrainStr = null;



        SimulationResultCollection results = new SimulationResultCollection(
                minVertices, maxVertices, vertexGranularity, pMin, pMax, pInc, exp);

        // loop through number of vertices
        for(int vCount = minVertices; vCount <= maxVertices;
                vCount += vertexGranularity) {
            // loop through edgeProbability
            for(int p = pMin; p <= pMax; p += pInc) {
                double prob = p / (double) exp;
                // loop through each simulation
                results.add(new Simulation(this, seed, vCount, prob,
                        numSimulations).simulate());
            }
            try {
                new PlotHandler(plotFilePrefix, results, vCount).write();
            } catch (IOException e) {
                System.err.println("Error writing file for v="+vCount);
            }
        }

        StringBuilder builder = new StringBuilder();
        for(int p = 0; p<= pMax; p+= pInc) {
            builder.append(", " + (p / ((double) exp)));
        }
        builder.append('\n');
        for(int v = minVertices; v<= maxVertices; v+= vertexGranularity) {
            builder.append(v + ", ");
            for(int p = pMin; p <= pMax; p+=pInc) {
                builder.append(results.get(v,p)+", ");
            }
            builder.append('\n');
        }
        PrintWriter tableWriter = null;
        final String tableSuffix = "-table.csv";
        try {
            tableWriter = new PrintWriter(plotFilePrefix + tableSuffix);
            tableWriter.print(builder.toString());
        } catch (FileNotFoundException e) {
            System.err.println("Error writing table data to file \""+
                plotFilePrefix + tableSuffix +"\"");
```

```java
        } finally {
            if(tableWriter != null) tableWriter.close();
        }
        System.out.println("Finished simulations! run \"java PlotHandler\" "+
        "followed by any number of .dwg files (that were previously generated) "+
        "to visualize the results.");
    } // main


    /**
     * Display the proper usage of this program and exit.
     */
    private static void usage() {
        System.err.printf ("Usage: java pj2 MonteCarlo "+
                "%1s %2s %3s %4s %5s %6s %7s %8s %9s\n",
                arguments[SEED],
                arguments[MIN_VERTICES],
                arguments[MAX_VERTICES],
                arguments[VERTEX_GRANULARITY],
                arguments[MIN_P],
                arguments[MAX_P],
                arguments[P_GRANULARITY],
                arguments[NUMBER_OF_SIMULATIONS],
                arguments[PLOT_FILE_PREFIX]);
        System.exit(1);
    }

    /**
     * Print an error message to System.err and gracefully exit
     * @param msg the error message to display
     */
    private static void displayError(String msg) {
        System.err.println(msg);
        usage();
    }
}
```

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

import java.io.File;
import java.io.IOException;
import java.text.DecimalFormat;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.plot.Dots;
import edu.rit.numeric.plot.Plot;

/**
 * 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]);
        }

        Plot plot = new Plot()
            .plotTitle (String.format
               ("Random Graphs, <I>V</I> = %1s", Integer.toString(v)))
            .xAxisTitle ("Edge Probability <I>p</I>")
            .xAxisTickFormat(new DecimalFormat("0.0"))
            .yAxisTitle ("Average Distance <I>d</I>")
            .yAxisTickFormat (new DecimalFormat ("0.0"))
            .seriesDots (Dots.circle (5))
            .seriesStroke (null)
            .xySeries (results);
        Plot.write(plot, new File(fileName));
    }

    /**
     * Open a GUI for each plot in order to visualize the results of a
     * previously run set of simulations.
     *
     * @param args each plot file generated that you wish to visualize
     */
    public static void main(String args[]) {
        if(args.length < 1) {
            System.err.println("Must specify at least 1 plot file.");
            usage();
        }

        for(int i = 0; i < args.length; i++) {
            try {
                Plot plot = Plot.read(args[i]);
                plot.getFrame().setVisible(true);
            } catch (ClassNotFoundException e) {
                System.err.println("Could not deserialize " + args[i]);
            } catch (IOException e) {
                System.err.println("Could not open " + args[i]);
            } catch (IllegalArgumentException e) {
                System.err.println("Error in file " + args[i]);
            }
        }

    }

    /**
     * Print the usage message for this program and gracefully exit.
     */
    private static void usage() {
        System.err.println("usage: java PlotHandler <plot-file-1> "+
                "(<plot-file-2> <plot-file-3>... etc.)");
        System.exit(1);
    }
}
```

```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() {
```

```
            Random prng;
            DoubleVbl.Mean thrAverage;

            @Override
            public void start() {
                prng = new Random(seed + rank());
                thrAverage = threadLocal(average);
            }

            @Override
            public void run(int i) {
                UndirectedGraph.randomGraph(prng, v, p).
                    accumulateDistances(thrAverage);
            }

        });

        return new SimulationResult(v, p, average.doubleValue());
    }
}
```

```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
```

```java
     * incremented.
     */
    public final int pExp;

    /**
     * Construct a simulation result collection. The parameter values
     * should reflect the values passed into the program through the
     * command line arguments.
     *
     * @param vMin The lower bound on number of vertices
     * @param vMax The upper bound on number of vertices
     * @param vInc The amount to increment the number of vertices by in
     *             each set of trials
     * @param pMin The scaled lower bound on edge probability
     * @param pMax The scaled upper bound on edge probability
     * @param pInc The scaled amount to increment the edge probability by
     *             in each set of trials
     * @param pExp The number of decimal places used to convert the edge
     *             probability into an integer
     */
    public SimulationResultCollection (int vMin, int vMax, int vInc,
            int pMin, int pMax, int pInc, int pExp) {
        this.vMin = vMin;
        this.vMax = vMax;
        this.vInc = vInc;
        this.pMin = pMin;
        this.pMax = pMax;
        this.pInc = pInc;
        this.pExp = pExp;
        this.rows = (vMax - vMin + vInc) / vInc;
        this.cols = (pMax - pMin + pInc) / pInc;
        this.averages = new double[rows][cols];
    }

    /**
     * Add a simulation result to the data matrix.
     *
     * @param result the simulation result to record
     */
    public void add(SimulationResult result) {
        int p = p(result.p);
        int col = col(p);
        int row = row(result.v);
        averages[row][col] = result.averageDistance;

    }

    /**
     * Get the average distance recorded for a given vertex count
     * and a scaled edge probability
     *
     * @param v the vertex count
     * @param p the scaled edge probability
     * @return the average distance recorded for this pair
     */
    public double get(int v, int p) {
        int row = row(v);
        int col = col(p);
```

```java
        return averages[row][col];
    }

    /**
     * Get an array of averages for varying edge probabilities and
     * a given vertex count.
     *
     * @param v the vertex count of interest
     * @return the array of averages for this vertex count
     */
    public double[] getAveragesForV(int v) {
        return averages[row(v)].clone();
    }

    /**
     * Convert a vertex value into its associated row value in the
     * data matrix.
     *
     * @param v the vertex count (or number of vertices) to convert
     * @return the associated row value in the data matrix
     */
    private int row(int v) {
        return (v - vMin) / vInc;
    }

    /**
     * Convert an edge probability into a scaled integer in order
     * to get rid of floating point arithmetic errors.
     *
     * @param p the edge probability to convert
     * @return the scaled integer ranging from pMin to pMax
     */
    private int p(double p) {
        return (int) (Math.round(p * pExp));
    }

    /**
     * Convert a scaled edge probability into the associated
     * column value in the data matrix.
     *
     * @param p the scaled edge probability to convert
     * @return the associated column value in the data matrix
     */
    private int col(int p) {
        return (p - pMin) / pInc;
    }

}
```

```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++) {
                int distance = BFS(i, j);
                // only accumulate the distance if the two vertices
                // are actually connected
                if(distance > 0) {
                    thrLocal.accumulate(distance);
                }
            }
        }
    }
```

```java

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

```
     *
     * @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;
    }
}
```