Joseph Ville
CSCI 351-01
Report - Programming Project 1

1. My programs are Monte Carlo simulations that generate random graphs and compute the average diameters of the graphs. For MonteCarloPSmp at the command line, the user specifies a seed for the pseudo-random number generator, the number of vertices, a range of edge probabillities, the number of trials, and the value by which to increment edge probability. The increment value must be a decimal number between 0.0 and 1.0.

   For MonteCarloVSmp at the command line, the user specifies a seed for the pseudo-random number generator, a range of vertices, the edge probability, the number of trials, and the value by which to increment the range of vertices. The increment value must be an integer.

   More specifically, the programs use the seed and a pseudo-random number generator to generate a random graph by the Gilbert procedure. They then perform a breadth first traversal to find the shortest path from every vertex to every other vertex. For each vertex *A*, the radius is then the longest distance from *A* to any other vertex *B*. Next the programs take the highest radius over all vertices of the graph, which is that graph's diameter. The diameter is then averaged. This procedure is repeated for every combination of the number of vertices and edge probability.

   Lastly, the programs prints the *p* and average diameter values.

2. The program MonteCarloPSmp sweeps over p, while holding V constant. For this program, enter the following command line:

```
   java pj2 MonteCarloPSmp <seed> <V> <lowerP> <upperP> <T>
<increment>
```

where

```
<seed> = Random seed
<V> = number of vertices
<lowerP> = Lower bound of edge probability
<upperP> = Upper bound of edge probability
<T> = Number of trials
 <increment> = number by which to increment the knob
```

The program MonteCarloVSmp sweeps over V, while holding p constant. For this program, enter the following command line:

```
java pj2 MonteCarloVSmp <seed> <lowerV> <upperV> <p> <T>
<increment>
```

where

```
 <seed> = Random seed
 <lowerV> = Lower bound of number of vertices
```

```
<upperV> = Upper bound of number of vertices
<p> = Edge probability
<T> = Number of trials
<increment> = number by which to increment the knob
```

```
3.   import java.util.LinkedList;
import java.util.ArrayList;
import java.util.Arrays;
import java.text.DecimalFormat;
import edu.rit.pj2.Task;
import edu.rit.util.Random;

import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Task;
import edu.rit.pj2.vbl.DoubleVbl;
import edu.rit.pj2.vbl.DoubleVbl.Sum;

/**
 * Perform a Monte Carlo simulation, using seed, p, and increment as the knob values
 *
 *  Usage: java pj2 MonteCarloPSmp <seed> <V> <lowerP> <upperP> <T> <increment>
 *   <seed> = Random seed
 * <V> = number of vertices
 *   <lowerP> = Lower bound of edge probability
 *   <upperP> = Upper bound of edge probability
 * <increment> = number by which to increment the knob
 *   <T> = Number of trials
 *
 * @author Joseph Ville
 *
 */
public class MonteCarloPSmp extends Task
{
    private long seed; // seed for pseudorandom graph generation
    private int V; // number of vertices
    private double lowerP; // upper bound edge probability
    private double upperP; // edge probability
    private long T; // # of trials
    private double increment; // the value by which to increment V

    /**
     * The default constructor for the class
     */
    public MonteCarloPSmp()
    {
    }

    /**
 * Main method for the program
 * @param args - the command line arguments
 */
public void main(String[] args) throws Exception
{
    int V; // number of vertices for the current iteration

    if(args.length != 6)
    {
        usage();
    }

    seed = Long.parseLong(args[0]);
    V = Integer.parseInt(args[1]);
    lowerP = Double.parseDouble(args[2]);
    upperP = Double.parseDouble(args[3]);
    T = Long.parseLong(args[4]);
    increment = Double.parseDouble(args[5]);

    // print the command line used to run this code
    System.out.print("$ java pj2 MonteCarloPSeq");
    for(String arg : args)
    {
        System.out.print(" " + arg);
```

```
}
System.out.println();

// int sum = 0;
// double avg = 0.0;
// double count = 0.0;
System.out.println("p\t\tAvg d");

for(double p1 = lowerP; p1 <= upperP; p1 += increment)
{
	final DoubleVbl.Sum sumVbl = new DoubleVbl.Sum();

	double pHold = p1;
	double avg;

	// do T trials in parallel
	parallelFor(0, T - 1).exec(new LongLoop()
	{
		// Per-thread variables
		ArrayList<Vertex> vertices;
		Random rand;
		Graph graph;
		DoubleVbl.Sum thrSum;
```

```
                /**
                 * initialize per-thread variables
                 */
                public void start()
                {
                    rand = new Random(seed + rank());
                    graph = new Graph(rand);
                    thrSum = threadLocal(sumVbl);
                }

                /**
                 * Loop body
                 */
                public void run(long t)
                {
                    // tHold = (int)t; // assign the storage variable
                    vertices = graph.generateGraph(V, pHold);
                    thrSum.item += (double)graph.diameter(V,
vertices);
                }
            });
            avg = sumVbl.doubleValue() / T;
            System.out.println(pHold + "\t\t" + avg);
            avg = 0.0;
        }
        }// end main()

        /**
         * Print a usage message and throw exception
         */
        private static void usage()
        {
                System.err.println("Usage: java pj2 MonteCarloPSmp <seed> <V>
<lowerP> <upperP> <T> <increment>\n" +
                                "<seed> = Random seed\n" +
                                "<V> = the number of vertices\n" +
                                "<lowerP> = Lower bound of Edge probability range\n" +
                                "<upperP> = Upper bound of Edge probability range\n" +
                                "<T> = Number of trials\n" +
                                "<increment> = the value by which to increment p (a
decimal number)");
                throw new IllegalArgumentException();
        }
    }

     import java.util.LinkedList;
    import java.util.ArrayList;
    import java.util.Arrays;
    import java.text.DecimalFormat;
    import edu.rit.pj2.Task;
    import edu.rit.util.Random;
    import edu.rit.pj2.LongLoop;
    import edu.rit.pj2.Task;
    import edu.rit.pj2.vbl.DoubleVbl;
    import edu.rit.pj2.vbl.DoubleVbl.Sum;

    /**
     * Perform a Monte Carlo simulation using seed, V, and increment as the knob values
     *
     *  Usage: java pj2 MonteCarloVSmp <seed> <lowerV> <upperV> <p> <T>
<increment>
     *    <seed> = Random seed
     *    <lowerV> = Lower bound of number of vertices
     *    <upperV> = Upper bound of number of vertices
     *  <p> = Edge probability
     *  <increment> = number by which to increment the knob
     *    <T> = Number of trials
     *
     * @author Joseph Ville
     *
     */
    public class MonteCarloVSmp extends Task
    {
        private long seed; // seed for pseudorandom graph generation
        private int lowerV; // upper bound of number of vertices
        private int upperV; // lower bound of number of vertices
        private double p; // edge probability
        private long T; // # of trials
        private int increment; // the value by which to increment the knob

        /**
         * The default constructor for the class
         */
        public MonteCarloVSmp()
        {
        }

        /**
         * Main method for the program
         * @param args - the command line arguments
         */
        public void main(String[] args) throws Exception
        {
                int V; // number of vertices for the current iteration

                if(args.length != 6)
                {
                        usage();
                }
```

```
seed = Long.parseLong(args[0]);
lowerV = Integer.parseInt(args[1]);
upperV = Integer.parseInt(args[2]);
p = Double.parseDouble(args[3]);
T = Long.parseLong(args[4]);
increment = Integer.parseInt(args[5]);

// print the command line used to run this code
System.out.print("$ java pj2 MonteCarloPSeq");
for(String arg : args)
{
        System.out.print(" " + arg);
}
System.out.println();

System.out.println("V\t\tAvg d");

for(int v1 = lowerV; v1 <= upperV; v1 += increment)
{
        final DoubleVbl.Sum sumVbl = new DoubleVbl.Sum();

        int vHold = v1;
        double avg;
```

```
// do T trials in parallel
parallelFor(0, T - 1).exec(new LongLoop()
{
        // Per-thread variables
        ArrayList<Vertex> vertices;
        Random rand;
        Graph graph;
        DoubleVbl.Sum thrSum;

        /**
         * initialize per-thread variables
         */
        public void start()
        {
                rand = new Random(seed + rank());
                graph = new Graph(rand);
                thrSum = threadLocal(sumVbl);
        }

        /**
         * Loop body
         */
        public void run(long t)
```

```
                {
                    vertices = graph.generateGraph(vHold, p);
                    // thrSum.reduce(new
DoubleVbl((double)graph.diameter(vHold, vertices)));
                    thrSum.item += (double)graph.diameter(vHold,
vertices);
                }
            });
            avg = sumVbl.item / T;
            System.out.println(vHold + "\t\t" + avg);
        }
    }// end main()

    /**
     * Print a usage message and throw exception
     */
    private static void usage()
    {
        System.err.println("Usage: java pj2 MonteCarloVSmp <seed> <lowerV>
<upperV> <p> <T> <increment>\n" +
                "<seed> = Random seed\n" +
                "<lowerV> = Lower bound of number of vertices\n" +
                "<upperV> = Upper bound of number of vertices\n" +
                "<p> = Edge probability\n" +
                "<T> = Number of trials\n" +
                "<increment> = the value by which to increment V (an
integer)");
        throw new IllegalArgumentException();
    }
}


import java.util.LinkedList;
import java.util.ArrayList;
import edu.rit.util.Random;

/**
 * Generate a random graph
 * @author Joseph Ville
 */
public class Graph
{
    private Random rand; // pseudorandom number generator
    private int[][] distances;

    /**
     * Construct a graph object
     */
    public Graph(Random rand)
    {
            this.rand = rand;
    }
```

```
/**
 * Initialize the array to hold distances
 * @param distances
 * @param V
 */
public void initializeDist(int[][] distances, int V)
{
	for(int i = 0; i < V; i++)
	{
		for(int j = 0; j < V; j++)
		{
			distances[i][j] = -1;
		}
	}
}

/**
 * Compute the diameter of the graph
 * @param V - number of vertices
 * @param vertices - list of vertices
 * @return diameter
 */
public int diameter(int V, ArrayList<Vertex> vertices)
```

```
    {
        distances = new int[V][V];
        initializeDist(distances, V);
        int radius = 0;
        int currDist = 0;
        int diameter = 0;

        // find distance - (# of edges in path) from every vertex to every other
vertex
        for(Vertex vertex : vertices)
        {
            for(int i = 0; i < V; i++)
            {
                // find radius - max distance from one vertex to another
vertex
                if(distances[vertex.index()][i] != -1)
                {
                    currDist = distances[vertex.index()][i];
                }
                else
                {
                    currDist = distance(vertices, V, vertex.index(), i);
                    distances[vertex.index()][i] = currDist;
                    distances[i][vertex.index()] = currDist;
                    if(currDist > radius)
                    {
                        radius = currDist;
                    }
                }
            }
            // find diameter - max radius over all vertices
            if(radius > diameter)
            {
                diameter = radius;
            }
        }
        return diameter;
    }

    /**
     * Find the distance between two vertices
     * @param adj - adjacency list of vertices
     * @param V - the number of vertices
     * @param start - index of the starting vertex
     * @param dest - index of the destination vertex
     * @return distance - from start to dest
     */
    public int distance(ArrayList<Vertex> adj, int V, int start, int dest)
    {
        LinkedList<Integer> queue = new LinkedList<Integer>(); // queue of vertex indices
```

```
Vertex current; // the current vertex
int[] parent = new int[V]; // parents of each vertex
boolean[] seen = new boolean[V]; // whether this vertex has been seen
int distance = 0;

for(int i = 0; i < V; i++) // initialize the arrays
{
        parent[i] = -1;
        seen[i] = false;
}

seen[start] = true;
int length = 0;

queue.add(start);
int a = -1;


//      find shortest path from A to B using breadth first traversal
while(queue.size() != 0)
{
        a = queue.poll(); // remove the head of the queue
        current = adj.get(a); // store current vertex
```

```
            for(Integer b : current.getNeighbors()) // loop through all neighbors
of current vertex
            {
                if(!seen[b])
                {
                    seen[b] = true;
                    queue.add(b); // add b to end of queue
                    parent[b] = a; // set the parent

                    if(b == dest) // check if destination is reached
                    {
                        int y = b; // copy b into another variable so I
don't have to change b
                        while(y != start) // backtrack up the path until
starting vertex is reached
                        {
                            if(parent[y] >= 0) // to ensure we
don't try to access parent of root vertex
                            {
                                y = parent[y]; // y's parent
now becomes y for the next pass
                                distance++;
                            } // end if
                        } // end while
                    } // end if
                } // end if
            } // end for
```

```
    } // end while
    return distance;
}

/**
 * Generate the graph
 * @param V - the number of vertices
 * @param p - the edge probability
 * @return a list of the vertices of the graph
 */
public ArrayList<Vertex> generateGraph(int V, double p)
{
    ArrayList<Vertex> vertices = new ArrayList<Vertex>(V);

    for(int i = 0; i < V; i++)
    {
        vertices.add(new Vertex(i, new ArrayList<Integer>(V-1)));

    }

    // For each vertex
    for(int a = 0; a < V - 1; a++)
    {
        for(int b = a + 1; b < V; b++)
        {
                if(rand.nextDouble() < p)
                {
                    vertices.get(a).getNeighbors().add(b);
                    vertices.get(b).getNeighbors().add(a);
                }
            }
        }
        return vertices;
    }
}


import java.util.LinkedList;
import java.util.ArrayList;

/**
 * A class to store attributes of a single vertex
 * @author Joseph Ville
 */
public class Vertex
{
    private int index; /** Index of this vertex */
    private ArrayList<Integer> neighbors; /** Neighbors of this vertex */
```

```
/**
 * Construct an object of this class
 * @param index - the index of this vertex, that is, its ID
 * @param neighbors - a list of all the neighbors of this vertex
 */
public Vertex(int index, ArrayList<Integer> neighbors)
{
        this.index = index;
        this.neighbors = neighbors;
}

/**
 * Add the specified Integer to this list of neighbors
 * @param n - the Integer to add
 * @return true - if the add was successful
 *               false - if the add failed
 */
public boolean addNeighbor(Integer n)
{
        return neighbors.add(n);
}

/**
 * Retrieve the list of neighbors associated with this vertex
 * @return the list of neighbors
 */
public ArrayList<Integer> getNeighbors()
{
        return neighbors;
}

/**
 * Retrieve the index location of this vertex
 * @return the index location
 */
public int index()
{
        return index;
}

/**
 * String representation of this object
 * @return a String representation of the vertex
 */
public String toString()
{
        return "\n" + index + ": " + neighbors.toString();
}
```

```
}
```

4. As the edge probability increases, the average diameter increases very quickly
5. $ java pj2 MonteCarloPSeq 465867 20 .005 1 1000 .005

	p	Avg d
0.005	0.67	
0.01	1.129	
0.015	1.503	
0.02	1.845	
0.025	2.226	
0.03	2.599	
0.035	3.003	
0.04	3.417	
0.045	3.818	
0.05	4.259	
0.055	4.638	
0.06	4.94	
0.065	5.299	
0.07	5.672	
0.075	5.881	
0.08	6.112	
0.085	6.242	
0.09	6.403	

0.095	6.528
0.1	6.547
0.105	6.525
0.11	6.485
0.115	6.459
0.12	6.326
0.125	6.219
0.13	6.126
0.135	6.035
0.14	5.885
0.145	5.742
0.15	5.625
0.155	5.499
0.16	5.37
0.165	5.253
0.17	5.142
0.175	5.046
0.18	4.921
0.185	4.842
0.19	4.747
0.195	4.659
0.2	4.578
0.205	4.476
0.21	4.383

0.215	4.311
0.22	4.232
0.225	4.154
0.23	4.078
0.235	4.009
0.24	3.936
0.245	3.881
0.25	3.818
0.255	3.744
0.26	3.684
0.265	3.632
0.27	3.573
0.275	3.517
0.28	3.467
0.285	3.419
0.29	3.373
0.295	3.318
0.3	3.277
0.305	3.242
0.31	3.216
0.315	3.187
0.32	3.163
0.325	3.139
0.33	3.121

0.335	3.1
0.34	3.089
0.345	3.075
0.35	3.056
0.355	3.039
0.36	3.025
0.365	3.015
0.37	3.002
0.375	2.984
0.38	2.97
0.385	2.955
0.39	2.939
0.395	2.93
0.4	2.914
0.405	2.896
0.41	2.868
0.415	2.846
0.42	2.814
0.425	2.795
0.43	2.767
0.435	2.743
0.44	2.71
0.445	2.675
0.45	2.639

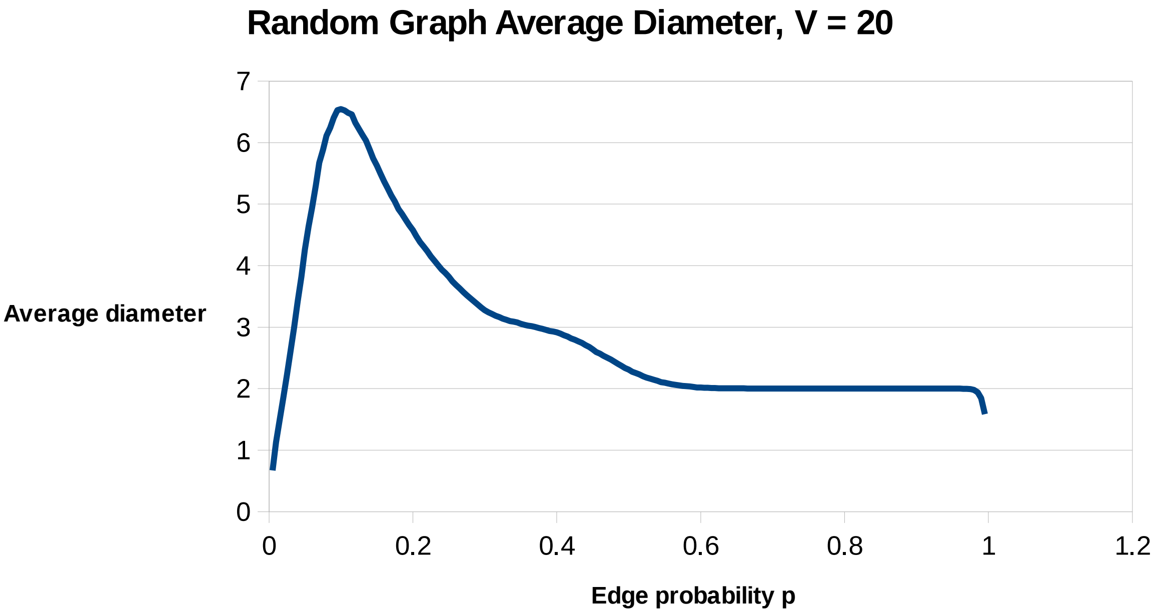
0.455	2.597
0.46	2.569
0.465	2.533
0.47	2.502
0.475	2.475
0.48	2.441
0.485	2.405
0.49	2.369
0.495	2.335
0.5	2.308
0.505	2.275
0.51	2.254
0.515	2.229
0.52	2.201
0.525	2.179
0.53	2.163
0.535	2.143
0.54	2.126
0.545	2.107
0.55	2.096
0.555	2.084
0.56	2.069
0.565	2.061
0.57	2.051

0.575	2.047
0.58	2.04
0.585	2.035
0.59	2.026
0.595	2.02
0.6	2.019
0.605	2.016
0.61	2.013
0.615	2.01
0.62	2.008
0.625	2.007
0.63	2.006
0.635	2.006
0.64	2.005
0.645	2.005
0.65	2.004
0.655	2.003
0.66	2.003
0.665	2.002
0.67	2.001
0.675	2.001
0.68	2.001
0.685	2.001
0.69	2.001

0.695	2.001
0.7	2
0.705	2
0.71	2
0.715	2
0.72	2
0.725	2
0.73	2
0.735	2
0.74	2
0.745	2
0.75	2
0.755	2
0.76	2
0.765	2
0.77	2
0.775	2
0.78	2
0.785	2
0.79	2
0.795	2
0.8	2
0.805	2
0.81	2

0.815	2
0.82	2
0.825	2
0.83	2
0.835	2
0.84	2
0.845	2
0.85	2
0.855	2
0.86	2
0.865	2
0.87	2
0.875	2
0.88	2
0.885	2
0.89	2
0.895	2
0.9	2
0.905	2
0.91	2
0.915	2
0.92	2
0.925	2
0.93	2

```
0.935   2
0.94    2
0.945   2
0.95    2
0.955   2
0.96    2
0.965   1.998
0.97    1.997
0.975   1.991
0.98    1.979
0.985   1.942
0.99    1.845
0.995   1.584
```



```
                                                                    $
java pj2 MonteCarloPSeq 465867 30 .005 1 1000 .005
p       Avg d
0.005   1.15
0.01    1.81
0.015   2.488
0.02    3.21
0.025   3.939
0.03    4.808
0.035   5.678
0.04    6.412
```

0.045	7.135
0.05	7.658
0.055	8.039
0.06	8.171
0.065	8.313
0.07	8.187
0.075	8.017
0.08	7.693
0.085	7.428
0.09	7.126
0.095	6.851
0.1	6.598
0.105	6.369
0.11	6.093
0.115	5.921
0.12	5.714
0.125	5.503
0.13	5.342
0.135	5.186
0.14	5.025
0.145	4.895
0.15	4.763
0.155	4.65
0.16	4.517

0.165	4.427
0.17	4.332
0.175	4.246
0.18	4.148
0.185	4.073
0.19	4.014
0.195	3.945
0.2	3.87
0.205	3.794
0.21	3.711
0.215	3.639
0.22	3.57
0.225	3.497
0.23	3.43
0.235	3.381
0.24	3.333
0.245	3.269
0.25	3.237
0.255	3.196
0.26	3.154
0.265	3.13
0.27	3.105
0.275	3.075
0.28	3.06

0.285	3.044
0.29	3.034
0.295	3.023
0.3	3.015
0.305	3.011
0.31	3.007
0.315	3.005
0.32	3.004
0.325	3.003
0.33	2.996
0.335	2.99
0.34	2.983
0.345	2.978
0.35	2.967
0.355	2.954
0.36	2.933
0.365	2.919
0.37	2.897
0.375	2.874
0.38	2.846
0.385	2.815
0.39	2.779
0.395	2.74
0.4	2.708

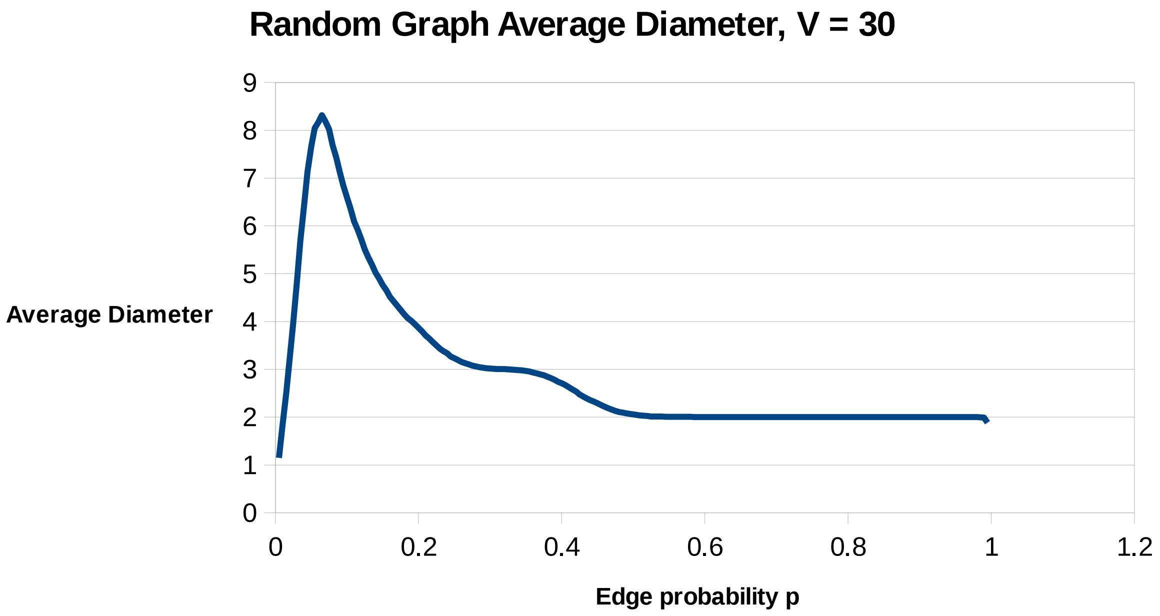
0.405	2.671
0.41	2.627
0.415	2.584
0.42	2.536
0.425	2.474
0.43	2.432
0.435	2.389
0.44	2.355
0.445	2.325
0.45	2.29
0.455	2.254
0.46	2.217
0.465	2.187
0.47	2.156
0.475	2.132
0.48	2.11
0.485	2.095
0.49	2.081
0.495	2.067
0.5	2.056
0.505	2.048
0.51	2.036
0.515	2.027
0.52	2.022

0.525	2.015
0.53	2.014
0.535	2.01
0.54	2.009
0.545	2.007
0.55	2.007
0.555	2.007
0.56	2.006
0.565	2.006
0.57	2.006
0.575	2.006
0.58	2.006
0.585	2.004
0.59	2.003
0.595	2.003
0.6	2.002
0.605	2
0.61	2
0.615	2
0.62	2
0.625	2
0.63	2
0.635	2
0.64	2

0.645	2
0.65	2
0.655	2
0.66	2
0.665	2
0.67	2
0.675	2
0.68	2
0.685	2
0.69	2
0.695	2
0.7	2
0.705	2
0.71	2
0.715	2
0.72	2
0.725	2
0.73	2
0.735	2
0.74	2
0.745	2
0.75	2
0.755	2
0.76	2

0.765	2
0.77	2
0.775	2
0.78	2
0.785	2
0.79	2
0.795	2
0.8	2
0.805	2
0.81	2
0.815	2
0.82	2
0.825	2
0.83	2
0.835	2
0.84	2
0.845	2
0.85	2
0.855	2
0.86	2
0.865	2
0.87	2
0.875	2
0.88	2

0.885	2
0.89	2
0.895	2
0.9	2
0.905	2
0.91	2
0.915	2
0.92	2
0.925	2
0.93	2
0.935	2
0.94	2
0.945	2
0.95	2
0.955	2
0.96	2
0.965	2
0.97	2
0.975	2
0.98	2
0.985	1.997
0.99	1.991
0.995	1.886



```
$ java pj2 MonteCarloPSeq 465867 40 .005 1 1000 .005
p       Avg d
0.005   1.488
0.01    2.527
0.015   3.597
0.02    4.866
0.025   6.201
0.03    7.706
0.035   8.894
0.04    9.617
0.045   9.799
0.05    9.561
0.055   9.179
0.06    8.687
0.065   8.17
0.07    7.714
0.075   7.324
0.08    6.897
0.085   6.576
0.09    6.249
```

0.095	5.977
0.1	5.737
0.105	5.505
0.11	5.292
0.115	5.12
0.12	4.945
0.125	4.782
0.13	4.643
0.135	4.533
0.14	4.418
0.145	4.305
0.15	4.218
0.155	4.133
0.16	4.063
0.165	3.984
0.17	3.924
0.175	3.845
0.18	3.763
0.185	3.682
0.19	3.588
0.195	3.487
0.2	3.42
0.205	3.334
0.21	3.26

0.215	3.208
0.22	3.161
0.225	3.12
0.23	3.086
0.235	3.065
0.24	3.05
0.245	3.029
0.25	3.017
0.255	3.013
0.26	3.008
0.265	3.004
0.27	3.003
0.275	3.002
0.28	3.001
0.285	3
0.29	3
0.295	3
0.3	2.999
0.305	2.999
0.31	2.994
0.315	2.989
0.32	2.983
0.325	2.977
0.33	2.97

0.335	2.957
0.34	2.936
0.345	2.911
0.35	2.887
0.355	2.84
0.36	2.795
0.365	2.76
0.37	2.731
0.375	2.678
0.38	2.616
0.385	2.552
0.39	2.501
0.395	2.459
0.4	2.404
0.405	2.353
0.41	2.316
0.415	2.27
0.42	2.227
0.425	2.195
0.43	2.16
0.435	2.14
0.44	2.112
0.445	2.091
0.45	2.073

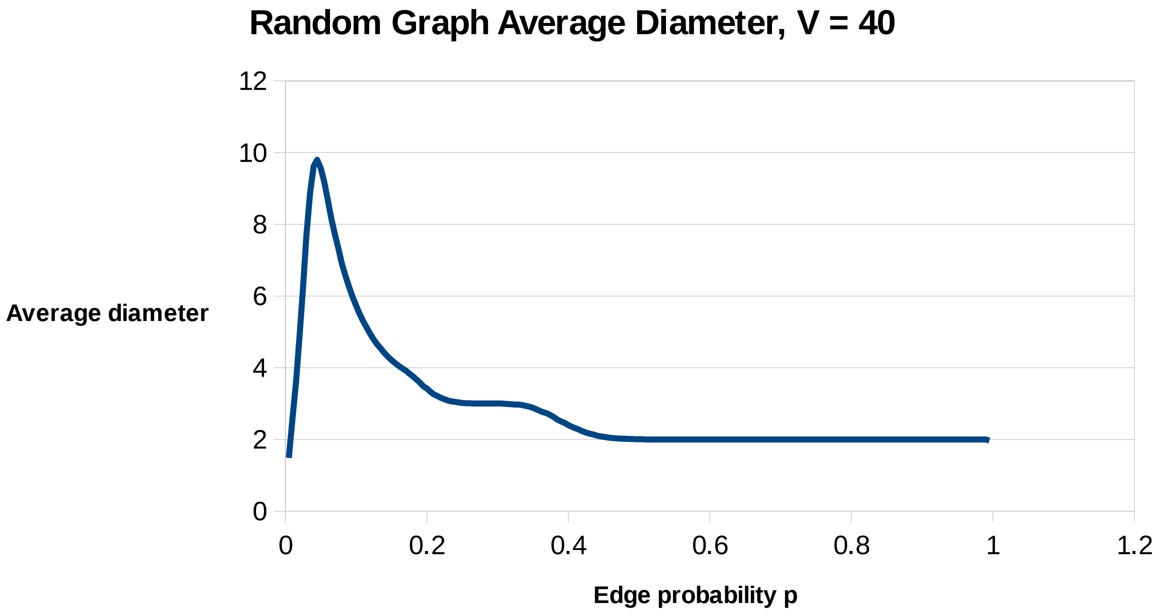
0.455	2.059
0.46	2.046
0.465	2.038
0.47	2.03
0.475	2.021
0.48	2.017
0.485	2.015
0.49	2.014
0.495	2.009
0.5	2.007
0.505	2.004
0.51	2.002
0.515	2.002
0.52	2.001
0.525	2.001
0.53	2.001
0.535	2.001
0.54	2.001
0.545	2.001
0.55	2
0.555	2
0.56	2
0.565	2
0.57	2

0.575	2
0.58	2
0.585	2
0.59	2
0.595	2
0.6	2
0.605	2
0.61	2
0.615	2
0.62	2
0.625	2
0.63	2
0.635	2
0.64	2
0.645	2
0.65	2
0.655	2
0.66	2
0.665	2
0.67	2
0.675	2
0.68	2
0.685	2
0.69	2

0.695	2
0.7	2
0.705	2
0.71	2
0.715	2
0.72	2
0.725	2
0.73	2
0.735	2
0.74	2
0.745	2
0.75	2
0.755	2
0.76	2
0.765	2
0.77	2
0.775	2
0.78	2
0.785	2
0.79	2
0.795	2
0.8	2
0.805	2
0.81	2

0.815	2
0.82	2
0.825	2
0.83	2
0.835	2
0.84	2
0.845	2
0.85	2
0.855	2
0.86	2
0.865	2
0.87	2
0.875	2
0.88	2
0.885	2
0.89	2
0.895	2
0.9	2
0.905	2
0.91	2
0.915	2
0.92	2
0.925	2
0.93	2

```
0.935   2
0.94    2
0.945   2
0.95    2
0.955   2
0.96    2
0.965   2
0.97    2
0.975   2
0.98    2
0.985   2
0.99    1.998
0.995   1.969
```



```
$ java pj2 MonteCarloPSeq 465867 50 .005 1 1000 .005
p       Avg d
0.005   1.91
0.01    3.369
0.015   4.985
0.02    6.995
0.025   8.956
0.03    10.38
0.035   10.847
```

0.04	10.416
0.045	9.798
0.05	9.097
0.055	8.328
0.06	7.753
0.065	7.25
0.07	6.805
0.075	6.44
0.08	6.105
0.085	5.785
0.09	5.553
0.095	5.325
0.1	5.107
0.105	4.915
0.11	4.75
0.115	4.593
0.12	4.447
0.125	4.314
0.13	4.215
0.135	4.126
0.14	4.058
0.145	4.003
0.15	3.942
0.155	3.867

0.16	3.776
0.165	3.69
0.17	3.585
0.175	3.478
0.18	3.37
0.185	3.272
0.19	3.199
0.195	3.158
0.2	3.114
0.205	3.085
0.21	3.058
0.215	3.041
0.22	3.022
0.225	3.016
0.23	3.011
0.235	3.007
0.24	3.006
0.245	3.003
0.25	3.001
0.255	3
0.26	3
0.265	3
0.27	3
0.275	3

0.28	3
0.285	3
0.29	2.999
0.295	2.998
0.3	2.996
0.305	2.991
0.31	2.978
0.315	2.964
0.32	2.948
0.325	2.924
0.33	2.888
0.335	2.842
0.34	2.783
0.345	2.721
0.35	2.667
0.355	2.596
0.36	2.526
0.365	2.462
0.37	2.416
0.375	2.36
0.38	2.304
0.385	2.257
0.39	2.215
0.395	2.187

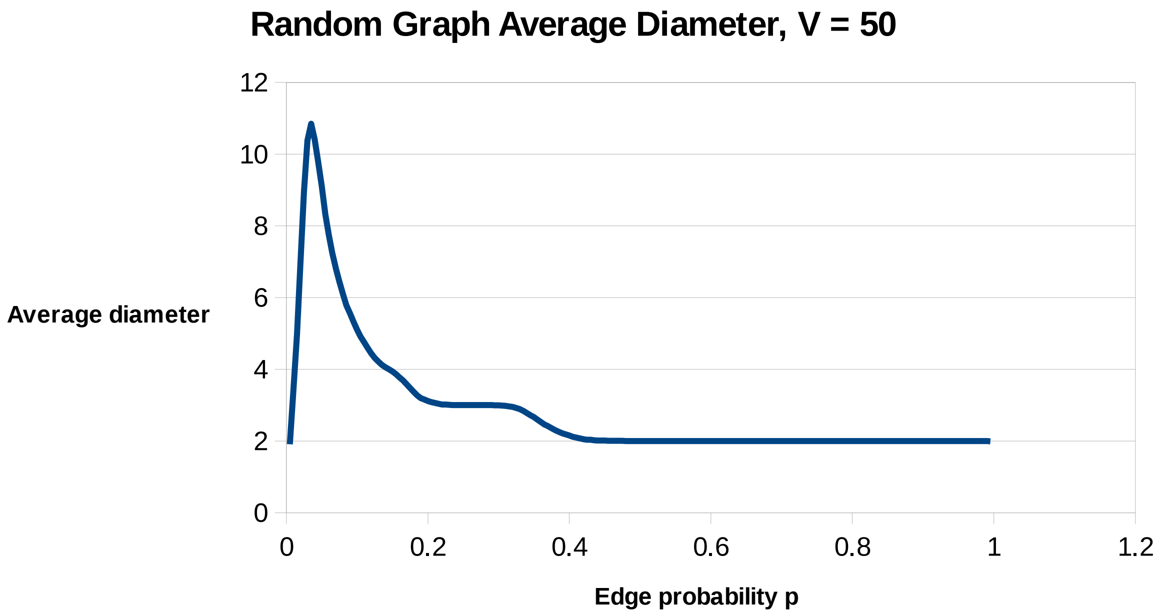
0.4	2.153
0.405	2.12
0.41	2.095
0.415	2.072
0.42	2.054
0.425	2.039
0.43	2.035
0.435	2.024
0.44	2.016
0.445	2.012
0.45	2.011
0.455	2.009
0.46	2.007
0.465	2.006
0.47	2.006
0.475	2.006
0.48	2.002
0.485	2
0.49	2
0.495	2
0.5	2
0.505	2
0.51	2
0.515	2

0.52	2
0.525	2
0.53	2
0.535	2
0.54	2
0.545	2
0.55	2
0.555	2
0.56	2
0.565	2
0.57	2
0.575	2
0.58	2
0.585	2
0.59	2
0.595	2
0.6	2
0.605	2
0.61	2
0.615	2
0.62	2
0.625	2
0.63	2
0.635	2

0.64	2
0.645	2
0.65	2
0.655	2
0.66	2
0.665	2
0.67	2
0.675	2
0.68	2
0.685	2
0.69	2
0.695	2
0.7	2
0.705	2
0.71	2
0.715	2
0.72	2
0.725	2
0.73	2
0.735	2
0.74	2
0.745	2
0.75	2
0.755	2

0.76	2
0.765	2
0.77	2
0.775	2
0.78	2
0.785	2
0.79	2
0.795	2
0.8	2
0.805	2
0.81	2
0.815	2
0.82	2
0.825	2
0.83	2
0.835	2
0.84	2
0.845	2
0.85	2
0.855	2
0.86	2
0.865	2
0.87	2
0.875	2

0.88	2
0.885	2
0.89	2
0.895	2
0.9	2
0.905	2
0.91	2
0.915	2
0.92	2
0.925	2
0.93	2
0.935	2
0.94	2
0.945	2
0.95	2
0.955	2
0.96	2
0.965	2
0.97	2
0.975	2
0.98	2
0.985	2
0.99	2
0.995	1.994



6.

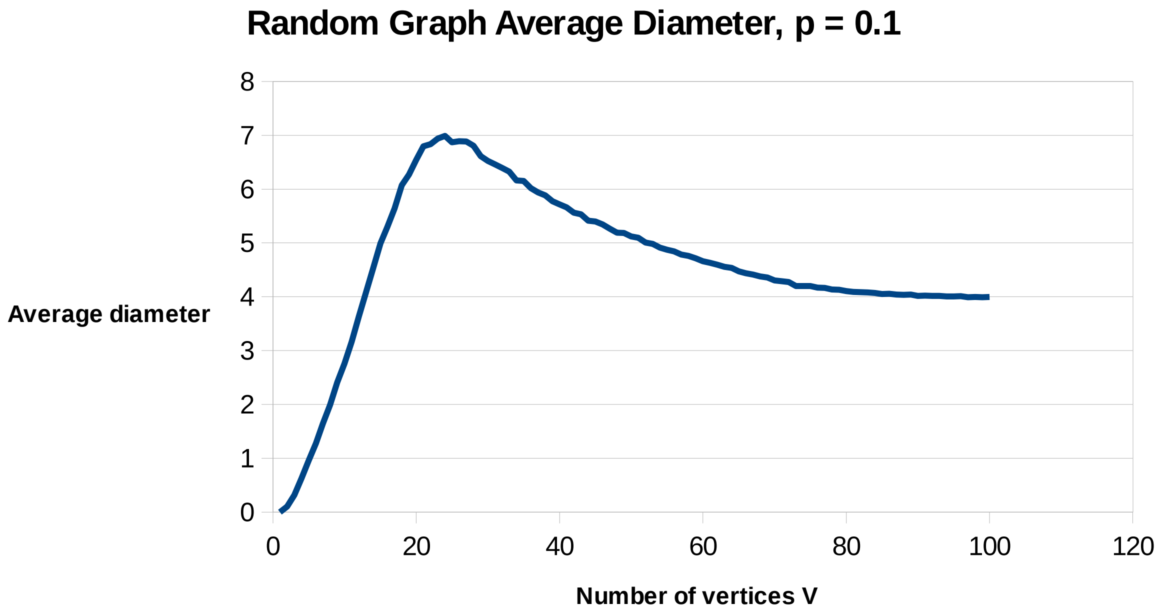
```
7. $ java pj2 MonteCarloPSeq 4657987 1 100 .1 1000 1
V       Avg d
1       0
2       0.101
3       0.316
4       0.625
5       0.96
6       1.274
7       1.645
8       1.989
9       2.409
10      2.758
11      3.165
12      3.635
13      4.083
14      4.538
15      4.99
16      5.297
17      5.643
```

18	6.074
19	6.268
20	6.541
21	6.795
22	6.833
23	6.939
24	6.987
25	6.869
26	6.892
27	6.886
28	6.803
29	6.611
30	6.521
31	6.46
32	6.394
33	6.325
34	6.16
35	6.153
36	6.019
37	5.939
38	5.882
39	5.777
40	5.717
41	5.661

42	5.56
43	5.531
44	5.412
45	5.4
46	5.342
47	5.263
48	5.188
49	5.183
50	5.118
51	5.094
52	5.007
53	4.984
54	4.914
55	4.872
56	4.842
57	4.782
58	4.759
59	4.716
60	4.658
61	4.628
62	4.595
63	4.558
64	4.534
65	4.473

66	4.435
67	4.411
68	4.379
69	4.358
70	4.306
71	4.288
72	4.271
73	4.202
74	4.197
75	4.199
76	4.171
77	4.165
78	4.136
79	4.131
80	4.103
81	4.092
82	4.086
83	4.078
84	4.073
85	4.053
86	4.054
87	4.043
88	4.037
89	4.039

```
90       4.016
91       4.019
92       4.014
93       4.018
94       4.003
95       4.005
96       4.012
97       3.989
98       3.994
99       3.991
100      3.993
```



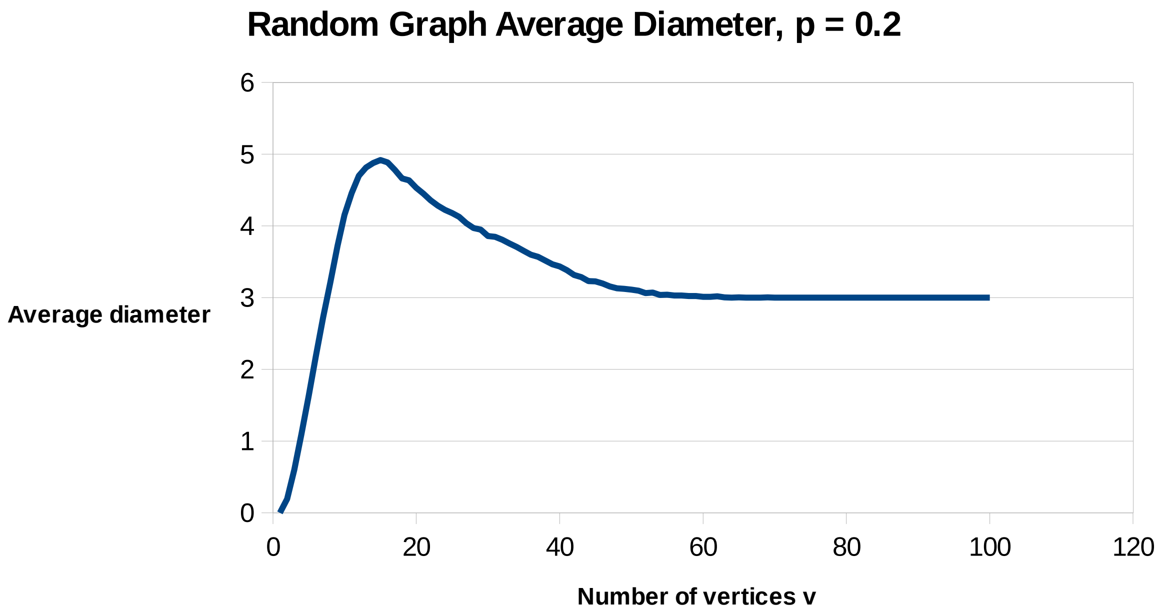
```
$ java pj2 MonteCarloPSeq 4657987 1 100 .2 1000 1
V        Avg d
1        0
2        0.194
3        0.602
4        1.104
5        1.633
6        2.189
7        2.725
8        3.212
9        3.721
```

10	4.154
11	4.457
12	4.699
13	4.815
14	4.877
15	4.92
16	4.887
17	4.78
18	4.661
19	4.637
20	4.533
21	4.452
22	4.359
23	4.284
24	4.223
25	4.178
26	4.128
27	4.039
28	3.97
29	3.947
30	3.858
31	3.847
32	3.808
33	3.756

34	3.708
35	3.653
36	3.598
37	3.57
38	3.517
39	3.466
40	3.435
41	3.382
42	3.318
43	3.287
44	3.23
45	3.228
46	3.197
47	3.155
48	3.129
49	3.122
50	3.113
51	3.095
52	3.065
53	3.069
54	3.037
55	3.042
56	3.03
57	3.031

58	3.024
59	3.022
60	3.011
61	3.011
62	3.018
63	3.004
64	3.002
65	3.004
66	3.001
67	3.002
68	3
69	3.003
70	3.001
71	3
72	3.002
73	3.001
74	3.001
75	3.001
76	3.001
77	3.001
78	3
79	3
80	3
81	3

82	3
83	3
84	3
85	3
86	3
87	3
88	3
89	3
90	3
91	3
92	3
93	3
94	3
95	3
96	3
97	3
98	3
99	3
100	3



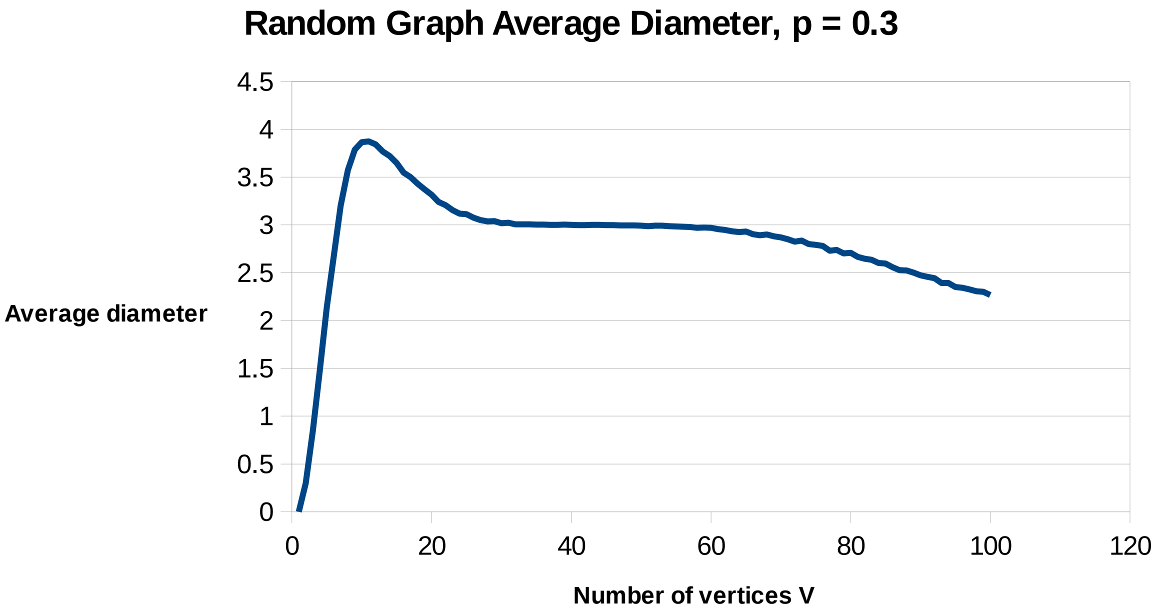
```
$ java pj2 MonteCarloPSeq 4657987 1 100 .3 1000 1
V       Avg d
1       0
2       0.299
3       0.853
4       1.484
5       2.14
6       2.669
7       3.202
8       3.573
9       3.787
10      3.866
11      3.873
12      3.842
13      3.767
14      3.721
15      3.646
16      3.547
17      3.501
18      3.43
19      3.372
```

20	3.316
21	3.239
22	3.206
23	3.154
24	3.117
25	3.112
26	3.075
27	3.052
28	3.037
29	3.038
30	3.018
31	3.023
32	3.007
33	3.007
34	3.005
35	3.002
36	3.003
37	3.001
38	3.001
39	3.002
40	3
41	2.998
42	2.999
43	3

44	3
45	2.998
46	2.998
47	2.995
48	2.996
49	2.994
50	2.991
51	2.988
52	2.993
53	2.992
54	2.987
55	2.984
56	2.981
57	2.977
58	2.97
59	2.971
60	2.97
61	2.957
62	2.948
63	2.935
64	2.925
65	2.932
66	2.903
67	2.893

68	2.901
69	2.88
70	2.87
71	2.851
72	2.824
73	2.835
74	2.8
75	2.79
76	2.78
77	2.731
78	2.738
79	2.701
80	2.708
81	2.667
82	2.646
83	2.635
84	2.602
85	2.596
86	2.558
87	2.526
88	2.524
89	2.501
90	2.474
91	2.456

```
92      2.442
93      2.392
94      2.391
95      2.351
96      2.342
97      2.326
98      2.306
99      2.299
100     2.268
```



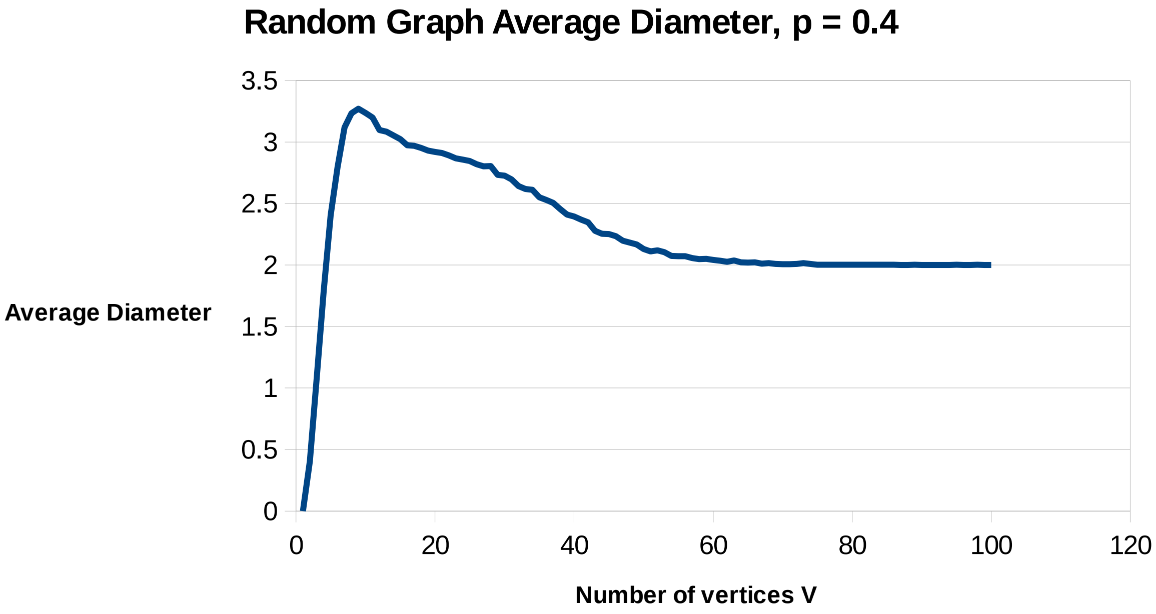
```
$ java pj2 MonteCarloPSeq 4657987 1 100 .4 1000 1
V       Avg d
1       0
2       0.405
3       1.089
4       1.791
5       2.405
6       2.797
7       3.119
```

8	3.234
9	3.271
10	3.237
11	3.199
12	3.096
13	3.083
14	3.054
15	3.024
16	2.973
17	2.969
18	2.952
19	2.93
20	2.92
21	2.911
22	2.89
23	2.866
24	2.856
25	2.846
26	2.819
27	2.802
28	2.805
29	2.733
30	2.726
31	2.696

32	2.642
33	2.618
34	2.611
35	2.552
36	2.529
37	2.505
38	2.455
39	2.41
40	2.395
41	2.369
42	2.348
43	2.278
44	2.253
45	2.251
46	2.235
47	2.197
48	2.181
49	2.166
50	2.13
51	2.11
52	2.119
53	2.103
54	2.074
55	2.072

56	2.072
57	2.057
58	2.047
59	2.049
60	2.042
61	2.034
62	2.027
63	2.036
64	2.023
65	2.02
66	2.021
67	2.012
68	2.015
69	2.009
70	2.006
71	2.007
72	2.009
73	2.015
74	2.008
75	2.002
76	2.002
77	2.003
78	2.001
79	2.003

80	2.002
81	2.003
82	2.001
83	2.002
84	2.002
85	2.001
86	2.002
87	2
88	2
89	2.001
90	2
91	2
92	2
93	2
94	2
95	2.001
96	2
97	2
98	2.001
99	2
100	2

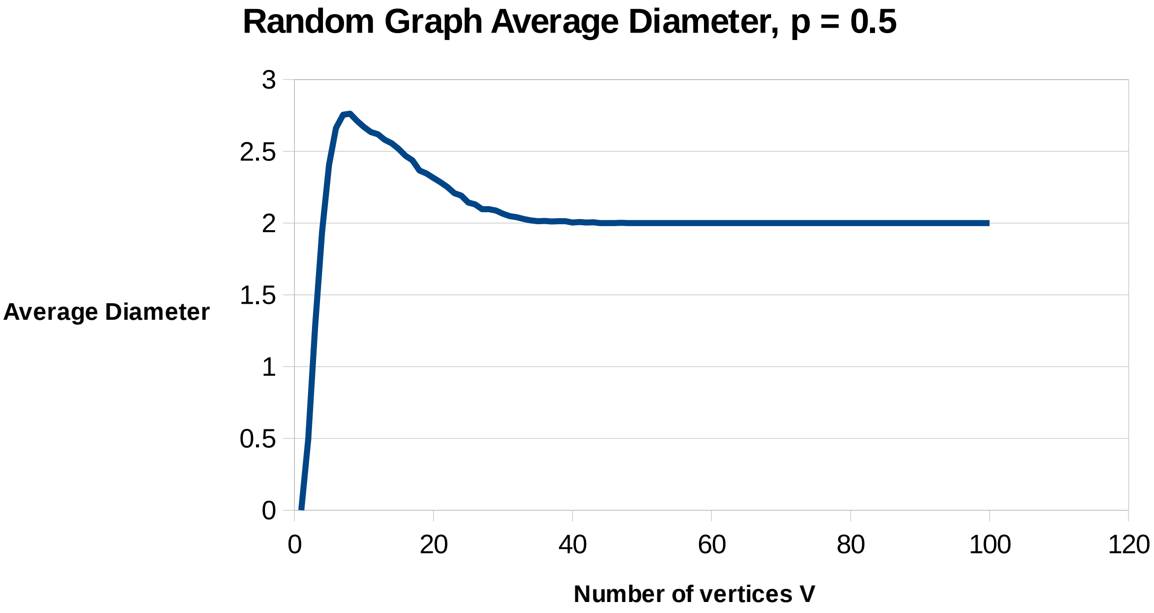

Random Graph Average Diameter, p = 0.4

```
$ java pj2 MonteCarloPSeq 465987 1 100 .5 1000 1
V      Avg d
1      0
2      0.495
3      1.285
4      1.941
5      2.405
6      2.662
7      2.754
8      2.763
9      2.712
10     2.669
11     2.635
12     2.619
13     2.581
14     2.556
15     2.517
16     2.469
17     2.438
18     2.367
19     2.347
20     2.315
21     2.285
22     2.251
```

23	2.208
24	2.192
25	2.144
26	2.13
27	2.097
28	2.097
29	2.087
30	2.065
31	2.048
32	2.042
33	2.028
34	2.02
35	2.014
36	2.015
37	2.012
38	2.013
39	2.013
40	2.004
41	2.007
42	2.004
43	2.005
44	2.001
45	2
46	2.001

47	2.003
48	2
49	2.001
50	2.001
51	2
52	2.001
53	2
54	2
55	2
56	2
57	2
58	2
59	2
60	2
61	2
62	2
63	2
64	2
65	2
66	2
67	2
68	2
69	2
70	2

71	2
72	2
73	2
74	2
75	2
76	2
77	2
78	2
79	2
80	2
81	2
82	2
83	2
84	2
85	2
86	2
87	2
88	2
89	2
90	2
91	2
92	2
93	2
94	2
95	2
96	2
97	2
98	2
99	2
100	2


Random Graph Average Diameter, p = 0.5