Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

1. Describe, at a high level but completely, what your simulation program(s)' inputs are, what the program(s) do, and what the outputs are.

The simulation program can perform a simulation for one given simulation, a sweep of simulations over values k or p. The inputs are: the mean request processing time, mean request interarrival time, number of requests, number of vertices, density parameter, rewiring probability, and a random seed.

Single Simulation:
A single simulation first sets up a Server Cluster based on a small world graph. The small world graph is created based on the algorithm that is a modification of the regular world graph -- all nodes connected in some repeating order. For each connection, based on a given probability, the node is either wired in the regular graph pattern, or it is wired to another random node.

After the graph is created, the generator is created in order to create the requests for the simulation. The generator creates requests which are database calls to a distributed database -- our small world graph -- with a random start node and a random destination node. The generator will then place each request into the simulation.

When a request is placed into a simulation, the generator checks to see if the request has found its database node for its data, if not, the current node is changed to a random adjacent node. This process repeats until the correct database node is found. For each change of state this request takes, the time a node takes to find its destination node is increase.

Once all the requests are added to the simulation, the simulation is started, and the mean time is computed and printed out.

Sweep Simulations for p and k:
The two sweep simulations do the same process as the above single simulation but in a monte carlo simulation. At the end of each monte carlo simulation for a particular p or k, the mean is printed out, and the value of p or k is increased. The values of p and k are increased until they reach the limit that is given from the command line.

Important: Be sure to describe exactly how your program(s) generate small-world graphs. Be sure to describe exactly how your program(s) perform the discrete event simulations.

2. State the exact command line(s) for running the program(s), including the Java main program class name and a description of each command line argument.

Usage: java DatabaseSim <type> <tproc> <treq> <nreq> <V> <k> <p> <seed> [transcript]
<type> = Type of the project.

- sweep_p : sweeps the p variable (see <p> note)

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

- sweep_k : sweeps the k variable (see <k> note)
- single : run a single simulation

<tproc> = Mean request processing time (μ)
<treq> = Mean request interarrival time (λ)
<nreq> = Number of requests
<V> = Number of vertices
<k> = Density parameter

- sweep_k : sweeps up to the k value given

<p> = rewiring probability

- sweep_p : sweeps up to the p value given

<seed> = Random seed
[transcript] = turn the transcript on/off for a simulation. Default off. (true or false).

For running a program that sweeps the p value up to .7, the command would be:
java DatabaseSim sweep_p 1.0 2.0 20 10 2 .7 123124

For running a program that sweeps the k value up to 13, the command would be:
java DatabaseSim sweep_k 1.0 2.0 20 20 13 .25 123124

3. Put in the report the complete source code of your simulation program(s).

```
import edu.rit.numeric.ExponentialPrng;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;

import java.util.*;

/**
 * Database server.
 * @author  Tyler Paulsen
 * @author  Alan Kaminsky - templated from https://cs.rit.edu/~ark/351/sim/java2html.php?file=6
 * @version 23-Apr-2015
 */
public class ServerCluster {
	private ExponentialPrng tprocPrng;
	private Simulation sim;
	private int V, k;
	private float p;
	private Random prng;
	public boolean debug = false ;
	public boolean hasError;
	Node[] graph;
```

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

```
	/**
	 * True to print transcript, false to omit transcript.
	 */
	public boolean transcript = false;

	/**
	 * Construct a new server. The server's request processing time is
	 * exponentially distributed with the given mean.
	 *
	 * @param  sim    Simulation.
	 * @param  tproc  Mean request processing time.
	 * @param  prng   Pseudorandom number generator.
	 */
	private ServerCluster(Simulation sim, double tproc, int V, int k, float p, Random prng) {
		this.sim = sim;
		this.tprocPrng = new ExponentialPrng (prng, 1.0/tproc);
		this.V = V;
		this.k = k;
		this.p = p;
		this.prng = prng;
		graph = new Node[V];
		hasError = false;
		for(int i = 0; i < V; ++i)
			graph[i] = new Node(i);
		generateSmallGraph();
	}

	/**
	 * generates a server cluster
	 * @param sim sumulation
	 * @param tproc = Mean request processing time
	 * @param V number of vertices
	 * @param k density parameter
	 * @param p rewiring probability
	 * @param prng random number generator
	 * @return
	 */
	public static ServerCluster generateCluster(Simulation sim, double tproc, int V, int k, float
p, Random prng){
		ServerCluster sc = new ServerCluster(sim,tproc, V, k, p, prng);
		//System.out.println(sc.hasError);
		//stem.out.println(isConnectedGraph(sc));
```

```
            while(sc.hasError || !isConnectedGraph(sc)){
                //stem.out.println("error)=:::);");
                sc = new ServerCluster(sim,tproc, V, k, p, prng);
            }
            return sc;
        }

        /**
         * checks to see if the graph is connected or not
         * @param sc - the graph
         * @return true if graph is connected else false.
         */
        public static boolean isConnectedGraph(ServerCluster sc){
            Node graph[] = sc.graph;
            HashSet<Integer> seen = new HashSet();
            LinkedList<Integer> queue = new LinkedList();
            int A = 0;
            queue.addLast(A);
            seen.add(A);
            while (!queue.isEmpty()) {
                A = queue.poll();
                for (Node B : graph[A].nodes) {
                    if (!seen.contains(B.id)) {
                        seen.add(B.id);
                        queue.addLast(B.id);
                    }
                }
            }
            return seen.size() == sc.getV();
        }


        /**
         * Add the given request to this server's queue.
         *
         * @param  request  Request.
         */
        public void add(Request request) {
            if (transcript)
                System.out.printf("%.3f %s added curr=%d dest=%d%n", sim.time(),
                                  request,
                                  request.currentDatabase,
                                  request.databaseNeeded);
```

```
                startProcessing(request);
        }

        /**
         * Start processing the request.
         */
        private void startProcessing(Request request) {
                if (transcript)
                        System.out.printf ("%.3f %s starts processing curr=%d dest=%d%n",
sim.time(),
                                        request,
                                        request.currentDatabase,
                                        request.databaseNeeded);

                if(request.foundDatabase()) {
                        sim.doAfter(tprocPrng.next(), new Event() {
                                public void perform() {
                                        finishProcessing(request);
                                }
                        });
                }else{
                        sim.doAfter(tprocPrng.next(), new Event() {
                                public void perform() {
                                        int d = request.currentDatabase;
                                        request.currentDatabase =
graph[d].getRandomNeighbor(request.currentDatabase);
                                        startProcessing(request);
                                }
                        });
                }
        }

        /**
         * Finish processing the first request in this server's queue.
         */
        private void finishProcessing(Request request) {
                if (transcript)
                        System.out.printf ("%.3f %s finishes processing curr=%d dest=%d%n",
sim.time(),
                                        request,
                                        request.currentDatabase,
                                        request.databaseNeeded);
                request.finish();
```

```
	}

	// start of Small World Graph cluster structure.
	/**
	 * print the graph. Used to debug.
	 */
	private void printGraph(){
		if(debug)System.out.println("\nGraph:");
		for(int i=0; i < V & debug; ++i)
			System.out.println(graph[i]);
	}

	/**
	 * @return number of vertices.
	 */
	public int getV(){
		return V;
	}
	/**
	 * generate a small world graph.
	 *
	 * Assumes the current graph is a regular graph.
	 */
	private void generateSmallGraph(){
		Node A,B,C;
		int a;
		for(int i = 0; i < V; ++i){
			A = graph[i];
			for(int j = 1; j <= k; ++j){
				B = graph[(i+j) % V];
				if(prng.nextFloat() < p){
					C = graph[prng.nextInt(V)];
					while(C.equals(A) || C.equals(B) || C.isAdjacent(A) ) {
						if (completeNode(A,B)) {
							hasError = true;
							return;
						}
						C = graph[prng.nextInt(V)];
					}
					B = C;
				}
				A.addNeighbor(B);
			}
```

```
                }
                printGraph();
        }
        private boolean completeNode(Node A, Node B){
                return (!A.nodes.contains(B.id) && (A.nodes.size() + 2) >= V);
        }
        /**
         * represents a node in the graph.
         */
        class Node{
                private HashSet<Node> nodes;
                int id;
                Node(int id){
                        this.id = id;
                        nodes = new HashSet<>();
                }
                @Override
                public boolean equals(Object o){
                        return ((Node)o).id == id;
                }

                /**
                 * finds and returns the nth neighbor
                 * @param curr
                 * @return
                 */
                public int getRandomNeighbor(int curr){
                        ArrayList<Node> random = new ArrayList(nodes);
                        Collections.shuffle(random);
                        try {
                                int ret = random.iterator().next().id;
                                while (ret == curr) {
                                        Collections.shuffle(random);
                                        ret = random.iterator().next().id;
                                }
                                return ret;
                        }catch(NoSuchElementException nsee){
                                System.out.println(random);
                                System.out.println(nodes);
                        }
                        return -1;
                }
                /**
```

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

```
     * remove the link between two nodes
     * @param n
     */
    public void remove(Node n){
            nodes.remove(n);
            n.removeSingle(this);
    }

    /**
     * remove a single node
     * @param n
     */
    private void removeSingle(Node n){
            nodes.remove(n);
    }
    /**
     * add an edge between nodes.
     * @param n
     */
    public void addNeighbor(Node n){
            nodes.add(n);
            n.addNeighborSingle(this);
    }

    /**
     * add a single link.
     * @param n
     */
    private void addNeighborSingle(Node n){
            nodes.add(n);
    }

    /**
     * is adjacent
     * @param n - node
     * @return
     */
    public boolean isAdjacent(Node n){
            return nodes.contains(n);
    }

    //to string
    public String toString() {
```

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

```
                        String ret = id + " : ";
                        for (Node n : nodes)
                                ret += n.id + " ";
                        return ret;
                }

        }

}

import edu.rit.numeric.Series;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;

/**
 * Class DatabaseSim is the Database serverCluster simulation main program.
 *
 * @author  Tyler Paulsen
 * @author  Alan Kaminsky - Templated from :
https://cs.rit.edu/~ark/351/sim/java2html.php?file=6
 * @version 18-Apr-2014
 */
public class DatabaseSim {
        private static double tproc;
        private static double treq;
        private static int nreq;
        private static long seed;
        private static int V;
        private static int k;
        private static float p;
        private static Random prng;
        private static Simulation sim;
        private static ServerCluster serverCluster;
        private static Generator generator;
        private static boolean transcript = false;
        private static final int MONTE_CARLO_RUNS = 2000;

        /**
         * Main program.
         */
        public static void main(String[] args) {
                // Parse command line arguments.
                switch(args[0]){
```

```
                        case "sweep_p":
                                parse(args);
                                sweep_p();
                                break;
                        case "sweep_k":
                                parse(args);
                                sweep_k();
                                break;
                        case "single":
                                parse(args);
                                single();
                                break;
                        default:
                                usage();
                }

        }

        /**
         * a monte carlo simulation sweeping k
         */
        private static void sweep_k(){
                System.out.println("k\tmean");
                int seed_count = 0;
                for(int i = 1; i < k; ++i) {
                        float mean = 0.0f;
                        for(int T = 0; T < MONTE_CARLO_RUNS; ++T) {
                                // Set up pseudorandom number generator.
                                prng = new Random(seed + seed_count++);
                                // Set up simulation.
                                sim = new Simulation();
                                // Set up one serverCluster.
                                //System.out.println(T);
                                serverCluster = ServerCluster.generateCluster(sim, tproc, V, i, p,
prng);
                                //serverCluster = new ServerCluster(sim, tproc, V, i, p, prng);
                                serverCluster.transcript = transcript;
                                // Set up request generator and generate first request.
                                generator = new Generator(sim, treq, nreq, prng, serverCluster);
                                // Run the simulation.
                                sim.run();
                                //gather stats
                                Series.Stats stats = generator.responseTimeStats();
```

```
                        mean += stats.mean;
                        //System.out.println(T);
                    }
                    System.out.printf ("%d\t%.3f%n", i, mean/MONTE_CARLO_RUNS);
                }
        }

        /**
         * a monte carlo simulation sweeping p
         */
        private static void sweep_p(){
                System.out.println("prob\tmean");
                int seed_count = 0;
                for(float i = 0; i < p; i += 0.01f) {
                        float mean = 0.0f;
                        for(int T = 0; T < MONTE_CARLO_RUNS; ++T) {
                                // Set up pseudorandom number generator.
                                prng = new Random(seed + seed_count++);
                                // Set up simulation.
                                sim = new Simulation();
                                // Set up one serverCluster.
                                //serverCluster = new ServerCluster(sim, tproc, V, k, i, prng);
                                serverCluster = ServerCluster.generateCluster(sim, tproc, V, k, i,
prng);
                                serverCluster.transcript = transcript;
                                // Set up request generator and generate first request.
                                generator = new Generator(sim, treq, nreq, prng, serverCluster);
                                // Run the simulation.
                                sim.run();
                                //gather stats
                                Series.Stats stats = generator.responseTimeStats();
                                mean += stats.mean;
                        }
                        System.out.printf ("%.02f\t%.3f%n", i, mean/MONTE_CARLO_RUNS);
                }
        }

        /**
         * a single simulation
         */
        private static void single(){
                // Set up pseudorandom number generator.
                prng = new Random (seed);
```

```
		// Set up simulation.
		sim = new Simulation();

		// Set up one serverCluster.
		serverCluster = ServerCluster.generateCluster(sim, tproc, V, k, p, prng);
		serverCluster.transcript = transcript;

		// Set up request generator and generate first request.
		generator = new Generator (sim, treq, nreq, prng, serverCluster);

		// Run the simulation.
		sim.run();

		// Print the response time mean and standard deviation.
		Series.Stats stats = generator.responseTimeStats();
		System.out.printf ("Response time mean   = %.3f%n", stats.mean);
		System.out.printf ("Response time stddev = %.3f%n", stats.stddev);
	}

	/**
	 * parses the args for a single simulation
	 * @param args
	 */
	private static void parse(String args[]){
		if (args.length < 8 || args.length        > 9) usage();
		try {
			tproc = Double.parseDouble(args[1]);
			treq = Double.parseDouble(args[2]);
			nreq = Integer.parseInt(args[3]);
			V = Integer.parseInt(args[4]);
			k = Integer.parseInt(args[5]);
			p = Float.parseFloat(args[6]);
			seed = Long.parseLong(args[7]);
			if (args.length == 9)
				transcript = Boolean.parseBoolean(args[8]);
		} catch (Exception e) {
			e.printStackTrace();
			usage();
		}
	}
	/**
	 * Print a usage message and exit.
```

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

```
		 */
		private static void usage() {
			System.err.println ("Usage: java DatabaseSim <type> <tproc> <treq> <nreq>
<V> <k> <p> <seed> [transcript]");
			System.err.println ("<type> = Type of the project. " +
				"\n\t sweep_p : sweeps the p variable" +
				"\n\t sweep_k : sweeps the k variable" +
				"\n\t single : run a single simulation");
			System.err.println ("<tproc> = Mean request processing time");
			System.err.println ("<treq> = Mean request interarrival time");
			System.err.println ("<nreq> = Number of requests");
			System.err.println ("<V> = Number of vertices");
			System.err.println ("<k> = Density parameter" +
				"\n\t sweep_k : sweeps up to the k value given");
			System.err.println ("<p> = rewiring probability" +
				"\n\t sweep_p : sweeps up to the p value given");
			System.err.println ("<seed> = Random seed");
			System.err.println ("[transcript] = turn the transcript on/off for a simulation. Default
off. (true or false).");
			System.exit (1);
		}
}

import edu.rit.numeric.ExponentialPrng;
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.Series;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;

/**
 * Class Generator generates requests for the database serverCluster simulations.
 *
 * @author Tyler Paulsen
 * @author  Alan Kaminsky - templated from:
https://cs.rit.edu/~ark/351/sim/java2html.php?file=8
 * @version 18-Apr-2014
 */
public class Generator {
		private Simulation sim;
		private ExponentialPrng treqPrng;
		private int nreq;
		private Random prng;
```

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

```
	private ServerCluster serverCluster;
	private ListSeries respTimeSeries;
	private int n;

	/**
	 * Create a new request generator.
	 *
	 * @param  sim     Simulation.
	 * @param  treq    Request mean interarrival time.
	 * @param  prng    Pseudorandom number generator.
	 * @param  serverCluster  ServerCluster.
	 */
	public Generator(Simulation sim, double treq, int nreq,  Random prng, ServerCluster
serverCluster) {
		this.sim = sim;
		this.prng = prng;
		this.treqPrng = new ExponentialPrng (prng, 1.0/treq);
		this.nreq = nreq;
		this.serverCluster = serverCluster;

		respTimeSeries = new ListSeries();
		n = 0;

		generateRequest();
	}

	/**
	 * Generate the next request.
	 */
	private void generateRequest() {
		int currentDatabase = prng.nextInt(serverCluster.getV());
		int databaseNeeded = prng.nextInt(serverCluster.getV());
		serverCluster.add (new Request (sim, databaseNeeded, currentDatabase,
respTimeSeries));
		++n;
		if (n < nreq){
			sim.doAfter (treqPrng.next(), new Event(){
				public void perform(){
					generateRequest();
				}
			});
		}
	}
```

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

```
	/**
	 * Returns the response time statistics of the generated requests.
	 *
	 * @return  Response time statistics (mean, standard deviation, variance).
	 */
	public Series.Stats responseTimeStats(){
		return respTimeSeries.stats();
	}

}

import edu.rit.numeric.ListSeries;
import edu.rit.sim.Simulation;

/**
 * Class Request provides a request in the database simulation.
 *
 * @author  Tyler Paulsen
 * @author  Alan Kaminsky - templated from:
https://cs.rit.edu/~ark/351/sim/java2html.php?file=1
 * @version 18-Apr-2014
 */
public class Request {
	private static int idCounter = 0;

	private int id;
	private Simulation sim;
	private double startTime;
	private double finishTime;
	private ListSeries respTimeSeries;
	public int databaseNeeded;
	public int currentDatabase;

	/**
	 * Construct a new request. The request's start time is set to the current
	 * simulation time.
	 *
	 * @param  sim     Simulation.
	 */
	public Request(Simulation sim, int databaseNeeded, int currentDatabase) {
		this.id = ++ idCounter;
		this.sim = sim;
```

```
                this.databaseNeeded = databaseNeeded;
                this.currentDatabase = currentDatabase;
                this.startTime = sim.time();
        }

        /**
         * returns the database of the request.
         * @return
         */
        public boolean foundDatabase(){
                return currentDatabase == databaseNeeded;
        }

        /**
         * Construct a new request. The request's start time is set to the current
         * simulation time. The request's response time will be recorded in the
         * given series.
         *
         * @param  sim     Simulation.
         * @param  series  Response time series.
         */
        public Request(Simulation sim, int databaseNeeded, int currentDatabase, ListSeries
series) {
                this (sim, databaseNeeded, currentDatabase);
                this.respTimeSeries = series;
        }

        /**
         * Mark this request as finished. The request's finish time is set to the
         * current simulation time. The request's response time is recorded in the
         * response time series.
         */
        public void finish() {
                finishTime = sim.time();
                if (respTimeSeries != null) respTimeSeries.add (responseTime());
        }

        /**
         * Returns this request's response time.
         *
         * @return  Response time.
         */
        public double responseTime() {
```

```
            return finishTime - startTime;
        }

        /**
         * Returns a string version of this request.
         *
         * @return  String version.
         */
        public String toString() {
            return "Request " + id;
        }
}
```

4. Based on data generated by running your simulation program(s), discuss this question: For a given number of vertices *V*, what happens to the average total time as the rewiring probability *p* increases, and why is this happening? Use the following fixed knob values: $k = 2$, $\lambda = 1.0$, $\mu = 2.0$.

The mean time eventually reaches some sort of a limit while increasing the value of p. With a small number of vertices, the rewiring probability will not change much resulting in a mostly regular graph. A large number of vertices will have a longer more standard amount of time when finding the target node. If we re-wire these graphs with a large amount of vertices, the chance that we can take some sort of shortcut to get to our target vertex increases.

5. Include in your report the data that you analyzed to answer the preceding question. The data must be presented ***both*** in a table or tables ***and*** in a plot or plots. ***Also include*** the exact program command line(s) that generated the data.

V=10		V=20		V=50	
prob	mean	prob	mean	prob	mean
0	22.15	0	68.426	0	376.343
0.01	22.029	0.01	68.791	0.01	344.565
0.02	22.031	0.02	67.667	0.02	317.858
0.03	22.194	0.03	67.102	0.03	299.365
0.04	22.263	0.04	66.418	0.04	282.816
0.05	22.124	0.05	64.862	0.05	267.745
0.06	22.219	0.06	64.891	0.06	252.111

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

0.07	22.337	0.07	63.833	0.07	245.068
0.08	22.242	0.08	63.948	0.08	236.745
0.09	22.624	0.09	63.053	0.09	227.134
0.1	22.388	0.1	63.137	0.1	220.405
0.11	22.257	0.11	62.943	0.11	213.769
0.12	22.473	0.12	61.9	0.12	208.366
0.13	22.356	0.13	61.276	0.13	203.917
0.14	22.649	0.14	60.675	0.14	200.166
0.15	22.578	0.15	60.455	0.15	195.778
0.16	22.53	0.16	59.964	0.16	192.943
0.17	22.711	0.17	59.417	0.17	190.251
0.18	22.765	0.18	59.351	0.18	187.224
0.19	22.781	0.19	59.168	0.19	184.039
0.2	22.612	0.2	58.884	0.2	182.602
0.21	22.908	0.21	58.538	0.21	178.572
0.22	22.855	0.22	58.843	0.22	178.05
0.23	22.886	0.23	58.197	0.23	175.328
0.24	22.796	0.24	58.723	0.24	173.639
0.25	22.944	0.25	57.584	0.25	173.354
0.26	22.91	0.26	57.503	0.26	169.865
0.27	23.109	0.27	57.442	0.27	169.631
0.28	23.031	0.28	57.377	0.28	168.643
0.29	22.821	0.29	56.922	0.29	166.737
0.3	22.995	0.3	56.766	0.3	167.853
0.31	23.108	0.31	56.606	0.31	166.928

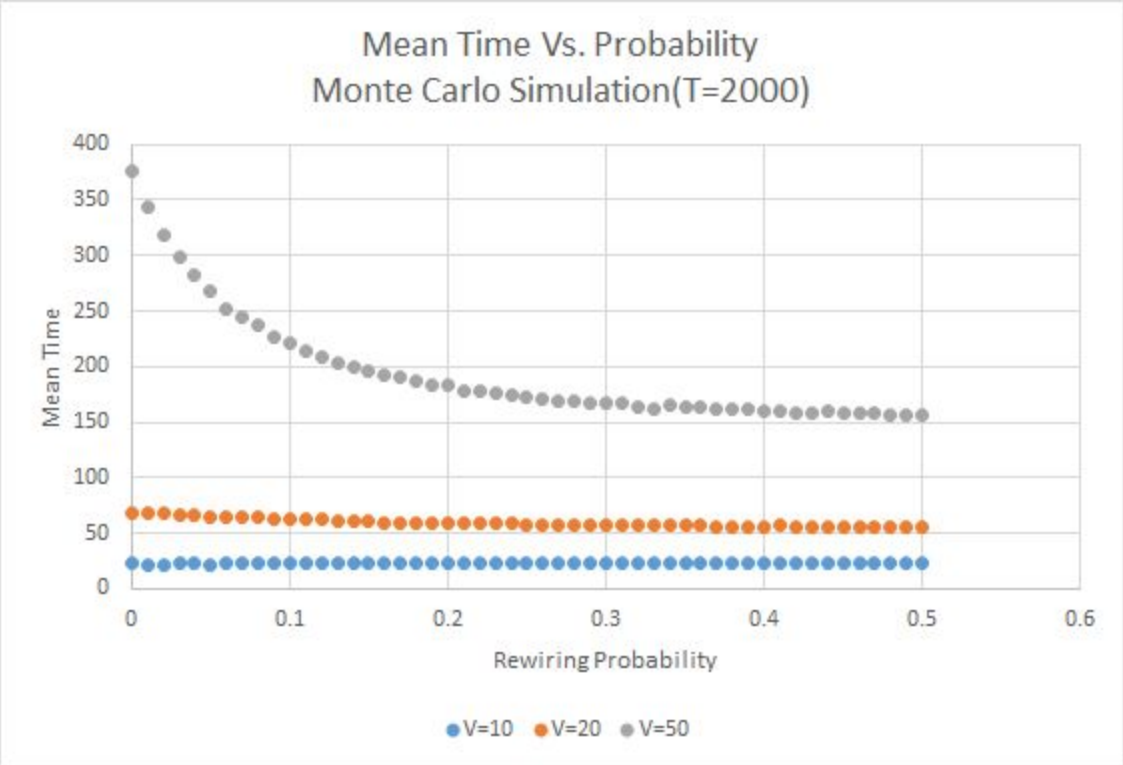
0.32	23.215	0.32	56.621	0.32	163.792
0.33	22.819	0.33	56.714	0.33	162.323
0.34	23.102	0.34	56.62	0.34	165.003
0.35	23.102	0.35	56.446	0.35	162.922
0.36	23.172	0.36	56.61	0.36	163.168
0.37	23.121	0.37	56.33	0.37	161.653

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

0.38	22.992	0.38	55.978	0.38	160.934
0.39	23.208	0.39	56.008	0.39	161.11
0.4	23.123	0.4	55.982	0.4	159.491
0.41	23.248	0.41	56.486	0.41	159.258
0.42	23.446	0.42	55.965	0.42	157.915
0.43	23.245	0.43	56.117	0.43	158.76
0.44	23.407	0.44	55.98	0.44	159.719
0.45	23.256	0.45	55.723	0.45	157.364
0.46	23.103	0.46	55.099	0.46	157.334
0.47	23.373	0.47	55.546	0.47	157.407
0.48	23.253	0.48	55.296	0.48	156.414
0.49	23.14	0.49	55.283	0.49	155.578
0.5	23.283	0.5	56.193	0.5	155.531
0.51	23.391	0.51	55.767	0.51	156.484
0.52	23.359	0.52	55.259	0.52	155.452
0.53	23.587	0.53	56.07	0.53	155.548
0.54	23.374	0.54	55.822	0.54	155.562
0.55	23.36	0.55	54.753	0.55	156.419
0.56	23.415	0.56	56.031	0.56	156.054
0.57	23.513	0.57	55.56	0.57	156.276
0.58	23.415	0.58	55.35	0.58	154.975
0.59	23.297	0.59	55.141	0.59	155.043
0.6	23.454	0.6	54.795	0.6	156.615
0.61	23.507	0.61	54.843	0.61	155.799
0.62	23.323	0.62	55.36	0.62	153.381
0.63	23.606	0.63	55.171	0.63	154.638
0.64	23.485	0.64	54.907	0.64	155.494
0.65	23.331	0.65	55.45	0.65	155.243
0.66	23.3	0.66	55.515	0.66	156.401
0.67	23.43	0.67	55.332	0.67	156.218
0.68	23.516	0.68	55.84	0.68	154.943

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

0.69	23.14	0.69	55.21	0.69	154.106
0.7	23.764	0.7	54.838	0.7	155.081



Important: I am expecting data for several values of *V*, not just one or two. I am expecting data for many values of *p*, starting from *p* = 0.0.

6. Based on data generated by running your simulation program(s), discuss this question: For a given number of vertices *V*, what happens to the average total time as the density parameter *k* increases, and why is this happening? Use the following fixed knob values: *p* = 0.1, *λ* = 1.0, *μ* = 2.0.

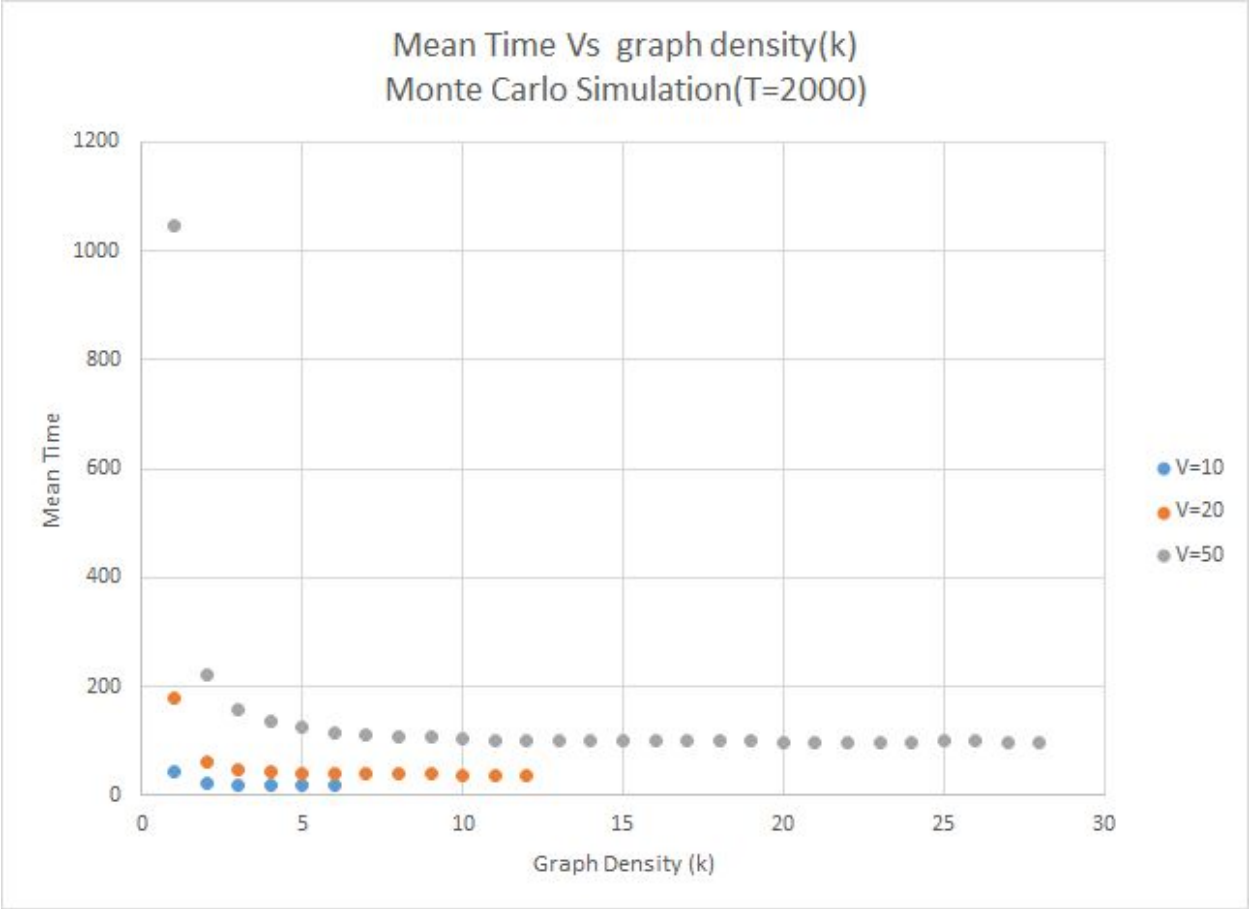
The mean time eventually reaches some sort of a limit while increasing the value of k. Since k is the amount of nodes that one node is connected to, increasing this value would drive down the mean time since the chance of the target node being adjacent to the current node has a higher chance. While the number of vertices increased, the number of nodes that are not the target node goes up, increasing the mean time.

7. Include in your report the data that you analyzed to answer the preceding question. The data must be presented ***both*** in a table or tables ***and*** in a plot or plots. ***Also include*** the exact program command line(s) that generated the data.

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4

V=10		V=20		V=50	
k	mean	k	mean	k	mean
1	43.428	1	178.53	1	1046.684
2	22.309	2	62.504	2	223.1
3	19.536	3	48.615	3	158.041
4	18.583	4	43.382	4	135.417
5	18.354	5	40.917	5	124.072
6	18.347	6	39.92	6	116.199
		7	39.342	7	111.149
		8	38.7	8	107.603
		9	38.267	9	106.495
		10	38.447	10	104.836
		11	38.329	11	102.304
		12	38.309	12	101.749
				13	101.8
				14	100.89
				15	99.393
				16	99.561
				17	99.171
				18	99.74
				19	100.456
				20	98.298
				21	97.943
				22	98.296
				23	97.86
				24	98.747
				25	98.882
				26	98.868
				27	98.418
				28	98.569

Tyler Paulsen
CSCI-351
4/27/2016
Programming Assignment #4



Important: I am expecting data for several values of *V*, not just one or two. I am expecting data for several values of *k*, not just one or two.

8. Write a paragraph describing what you learned from this project.

The simulation process introduced to me that there is an actual field in network simulations. The interesting thing about the two graphs is that each sweep eventually hit some sort of a limit. The limit can help someone decide how much the small world graph parameters needs to be adjusted in order to get the best mean time. The tools used to create the simulation helped learn how a simulations should be in industry along with the understanding on how quickly learn a newly introduced Library.