Joseph Ville
CSCI 351-01
Data Communications and Networks
Report for Programming Project 4

1. Each of my simulation programs take in, at the command line, the seed, the number of vertices *V*, the mean arrival rate *lambda*, the mean service rate *mu* , the number of queries *N*, and the number of trials *t*.

   Additionally at the command line, `PSweepDiscEventSim` takes in *k*, the graph density parameter, the lower and upper bounds of the rewiring probability range, *lowerP* and *upperP*, and the *p* increment value, *pIncrement*. `KSweepDiscEventSim` takes in the upper and lower bounds of k, *lowerK* and *upperK*, and rewiring probability *p*.

   In both programs a trial of the simulation proceeds as follows: set up a simulation, then generate a small-world graph using the Watts-Strogatz procedure, which generates a k-regular graph and rewires a fraction p of its edges. Next, they set up a query generator. The first query is generated when the query generator is constructed. All other queries are generated by recursive calls to the generate query procedure.

   The **generate query** procedure: A new query is constructed, and the starting and ending vertices are chosen uniformly at random during query construction. Each new query has different starting and ending vertices. The first query will not start processing until the simulation begins. Next, the simulation is run. After the simulation waits an amount of time determined by an exponential pseudorandom number generator created with mu, the first query is processed and the starting node will choose the next node uniformly at random. If the next node is the destination, the query finishes, otherwise, the query is added to that next node. The query is then processed by that next node and every next node chosen randomly thereafter until the destination node is reached.

   When a query finishes, the simulation will wait a period of time determined by another exponential pseudorandom number generator constructed with lambda, and the generate query procedure will be performed recursively until the number of queries generated equals the number specified at the command line.

   Finally, the simulation time is recorded. The time for each trial is added to a total. When all the trials are complete, this total is divided by *t* to get the average total time. `PSweepDiscEventSim` will repeat this entire operation for each *p*-value from the specified range and increment, and `KSweepDiscEventSim` for each *k*-value from the specified range.

2. The programs use the Parallel Java 2 library, so the classpath must be set accordingly before the code can be compiled. Once it is compiled, the following two commands will run their respective programs:

```
java PSweepDiscEventSim <seed> <V> <k> <lowerP> <upperP> <pIncrement>
<lambda> <mu> <N> <t> <transcript>
```

<seed> = seed for graph generation (must not be a decimal number)
<V> = number of vertices (must be an integer)
<k> = parameter to determine graph density (must be an integer)
<lowerP> = bound of p = rewiring probability
<upperP> = upper bound of p = rewiring probability
<pIncrement> = the value by which to increment p
<lambda> = mean query arrival rate
<mu> = mean query service rate
<N> = number of queries to create (must be an integer)
<t> = number of trials (must be an integer)
<transcript> (optional) = a boolean whether to print the transcript. false if omitted.

```
java KSweepDiscEventSim <seed> <V> <lowerK> <upperK> <p> <lambda> <mu> <N>
<t> <transcript>
```

<seed> = seed for graph generation (must not be a decimal number)
<V> = number of vertices (must be an integer)
<lowerK> = lower k parameter (an integer >= 1) to determine graph density
<upperK> = upper k parameter to determine graph density
<p> = rewiring probability; determines graph entropy
<lambda> = mean query arrival rate
<mu> = mean query service rate
<N> = number of queries to create (must be an integer)
<t> = number of trials (must be an integer)
<transcript> (optional) = a boolean whether to print the transcript. false if omitted.

3.

```
import java.text.DecimalFormat;

/**
 * Perform a discrete event simulation that will sweep through a
 * series of specified values for the rewiring probability p.
 *
 * @author Joseph Ville
 *
 */
public class PSweepDiscEventSim
```

```
{
	private static long seed;
	private static int V;
	private static int k;
	private static double lowerP;
	private static double upperP;
	private static double pIncrement;
	private static double lambda;
	private static double mu;
	private static int N;
	private static int t;

	/**
	 * Main method for this program
	 * @param args
	 */
	public static void main(String[] args)
	{
		/*
		 *  there can be either 10 or 11 command line args, because the last
		 *  one, transcript, is optional
		 */
		if(args.length < 10 || args.length > 11)
		{
			usage();
		}
		seed = Long.parseLong(args[0]);
		V = Integer.parseInt(args[1]);
		k = Integer.parseInt(args[2]);
		lowerP = Double.parseDouble(args[3]);
		upperP = Double.parseDouble(args[4]);
		pIncrement = Double.parseDouble(args[5]);
		lambda = Double.parseDouble(args[6]);
		mu = Double.parseDouble(args[7]);
		N = Integer.parseInt(args[8]);
		t = Integer.parseInt(args[9]);

		if(args.length == 11)
		{
			Node.transcript = Boolean.parseBoolean(args[10]);
		}

		if(k > V/2)
		{
```

```
                System.err.println("Usage: k should not be greater than V/2");
                System.exit(0);
            }

            System.out.print("java PSweepDiscEventSim ");
            for(String arg : args)
            {
                System.out.print(arg + " ");
            }
            System.out.println();
            double time = 0.0;
            System.out.println("p\tavg total time");
            for(double p = lowerP; p <= upperP; p+=pIncrement)
            {
                time = TotalTime.average(V, p ,t, seed, k, mu, lambda, N);
                System.out.println(new DecimalFormat("0.000").format(p) + "\t" +
                        new DecimalFormat("0.00000").format(time));
            }
        }

        /**
         * Print a usage message and exit
         */
        public static void usage()
        {
            System.err.println("Usage: java DiscEventSim <seed> <V> <k> <lowerP>
<upperP> <pIncrement> <lambda> <mu> <N> <t> <transcript>\n"
                    + "<seed> = seed for graph generation\n"
                    + "<V> = number of vertices\n"
                    + "<k> = parameter to determine graph density\n"
                    + "<lowerP> = lower bound of p = rewiring probability\n"
                    + "<upperP> = upper bound of p = rewiring probability\n"
                    + "<pIncrement> = the value by which to increment p\n"
                    + "<lambda> = mean query arrival rate\n"
                    + "<mu> = mean query service rate\n"
                    + "<N> = number of queries to create\n"
                    + "<t> = number of trials\n"
                    + "<transcript> (optional) = a boolean whether to print the
transcript. false if omitted.");
            System.exit(0);
        }
}
```

```
import java.text.DecimalFormat;

/**
 * Perform a discrete event simulation that will sweep through a
 * series of specified values for the graph density parameter k.
 *
 * @author Joseph Ville
 *
 */
public class KSweepDiscEventSim
{
	private static long seed;
	private static int V;
	private static int lowerK;
	private static int upperK;
	private static double p;
	private static double lambda;
	private static double mu;
	private static int N;
	private static int t;

	/**
	 * Main method for this program
	 * @param args
	 */
	public static void main(String[] args)
	{
		/*
		 *  there can be either 9 or 10 command line args, because the last
		 *  one, transcript, is optional
		 */
		if(args.length < 9 || args.length > 10)
		{
			usage();
		}
		seed = Long.parseLong(args[0]);
		V = Integer.parseInt(args[1]);
		lowerK = Integer.parseInt(args[2]);
		upperK = Integer.parseInt(args[3]);
		p = Double.parseDouble(args[4]);
		lambda = Double.parseDouble(args[5]);
		mu = Double.parseDouble(args[6]);
		N = Integer.parseInt(args[7]);
		t = Integer.parseInt(args[8]);
```

```
		if(args.length == 10)
		{
			Node.transcript = Boolean.parseBoolean(args[9]);
		}

		if(upperK > V/2)
		{
			System.err.println("Usage: upperK should not be greater than V/2");
			System.exit(0);
		}

		System.out.print("java KSweepDiscEventSim ");
		for(String arg : args)
		{
			System.out.print(arg + " ");
		}
		System.out.println();
		double time = 0.0;
		System.out.println("k\tavg total time");
		for(int k = lowerK; k <= upperK; k++)
		{
			time = TotalTime.average(V, p, t, seed, k, mu, lambda, N);
			System.out.println(k + "\t" + new DecimalFormat("0.00000").format(time));
		}
	}

	/**
	 * Print a usage message and exit
	 */
	public static void usage()
	{
		System.err.println("Usage: java KSweepDiscEventSim <seed> <V> <lowerK>
<upperK> <p> <lambda> <mu> <N> <t> <transcript>\n"
				+ "<seed> = seed for graph generation\n"
				+ "<V> = number of vertices\n"
				+ "<lowerK> = lower k parameter (an integer >= 1) to determine
graph density\n"
				+ "<upperK> = upper k parameter to determine graph density\n"
				+ "<p> = rewiring probability  \n"
				+ "<lambda> = mean query arrival rate\n"
				+ "<mu> = mean query service rate\n"
				+ "<N> = number of queries to create\n"
				+ "<t> = number of trials\n"
```

```
                                        + "<transcript> (optional) = a boolean whether to print the
transcript. false if omitted.");
                        System.exit(0);
                }
}


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

/**
 * A class to hold the method for computing the average simulated time
 * over t trials to process N queries
 *
 * @author Joseph Ville
 *
 */
public class TotalTime
{
        /**
         * Creates simulation, generates graph and query generator, runs simulation,
         * and averages total time over all the trials
         * @param v - number of vertices
         * @param p - rewiring probability
         * @param t - number of trials
         * @param seed - seed value for the prng
         * @param k - graph density parameter
         * @param lambda - mean arrival rate
         * @param mu - mean service rate
         * @param N - number of queries
         * @return average total time for all the trials
         */
        public static double average(int v, double p, int t, long seed, int k,
                        double lambda, double mu, int N)
        {
                double timeTotal = 0.0;
                SmallWorldGraph swg;
                Node[] nodes;
                Random prng;
                Simulation sim;

                // repeat for the given number of trials
                for(int i = 0; i < t; i++)
                {
```

```
                        prng = new Random(seed);

                        // Set up Simulation
                        sim = new Simulation();

                        // Set up the graph
                        swg = new SmallWorldGraph(v, k, p, prng, sim, mu);
                        nodes = swg.generateGraph();

                        // Set up query generator and generate the first query
                        new Generator(sim, lambda, N, prng, nodes, v);

                        // Run the simulation
                        sim.run();
                        timeTotal += sim.time();

                        Query.resetIdCounter();
                }
                return timeTotal/t;
        }
}


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

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

/**
 * Create and generate a small-world graph object
 * @author Joseph Ville
 *
 */
public class SmallWorldGraph
{
        private static Node[] nodes;
        private int V;
        private int k;
        private double p;
        private int E;
        private Random prng;
        private Simulation sim;
        private double mu;
```

```
/**
 * Construct a small world graph
 * @param V - number of vertices
 * @param k - graph density parameter
 * @param p - rewiring probability
 * @param prng - Random
 * @param sim - Simulation
 * @param mu - mean service rate
 */
public SmallWorldGraph(int V, int k, double p, Random prng, Simulation sim, double mu)
{
	nodes = new Node[V];
	this.V = V;
	this.k = k;
	this.p = p;
	this.prng = prng;
	this.sim = sim;
	this.mu = mu;
}

/**
 * Generate a small-world graph by the Watts-Strogatz procedure
 * @return array of nodes, representing the graph
 */
public Node[] generateGraph()
{
	Node a, b = null, c = null;
	double rand = 0.0;

	for(int d = 0; d < V; d++)
	{
		nodes[d] = new Node(d, sim, mu, prng);
	}

	for(int i = 0; i <= V-1; i++)
	{
		a = nodes[i];
		for(int j = 1; j <= k; j++)
		{
			b = nodes[(i+j) % V]; // Edge for k-regular graph

			if((rand = prng.nextDouble()) < p)
			{
```

```
                                        c = nodes[(int)(rand * V)];
                                        while(c.equals(a) || c.equals(b) || a.edgeExists(c) ||
c.edgeExists(a))
                                        {
                                                c = nodes[(int)(prng.nextDouble() * V)];
                                        }
                                        b = c; // Rewired edge for small-world graph
                                }
                                a.addNeighbor(b);
                                b.addNeighbor(a);
                                E++;
                        }
                }
                if(!connectedGraph(nodes))
                {
                        generateGraph();
                }
                return nodes;
        }

        /**
         * Determine whether a graph is connected by performing a breadth-first
         * search on an array of nodes
         * @param vertices - array of nodes
         * @return true - if connected
         *                 false - otherwise
         */
        public boolean connectedGraph(Node[] nodes)
        {
                boolean[] seen = new boolean[nodes.length];
                LinkedList<Integer> queue = new LinkedList<Integer>();
                int start = 0;

                seen[start] = true;
                queue.add(start);

                int current = Integer.MIN_VALUE;
                while(!queue.isEmpty())
                {
                        current = queue.poll();
                        ArrayList<Node> neighbors = nodes[current].neighbors();
                        for(int n = 0; n < neighbors.size(); n++)
                        {
                                if(!seen[neighbors.get(n).id()])
```

```
                                {
                                        seen[neighbors.get(n).id()] = true;
                                        queue.add(neighbors.get(n).id());
                                }
                        }
                }
                for(int i = 0; i < seen.length; i++)
                {
                        if(seen[i] == false)
                        {
                                return false;
                        }
                }
                return true;
        }
}
```

```
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 queries for the distributed database
 * querying simulations
 *
 * @author  Alan Kaminsky
 *                  modified by Joseph Ville
 * @version 18-Apr-2014
 */
public class Generator
{
        private Node[] nodes;
        private Simulation sim;
        private ExponentialPrng lambdaPrng;
        private int N;
        private Node sourceNode;

        private ListSeries respTimeSeries;
        private int n;
```

```
	private int V;

	/**
	 * Create a new request generator.
	 *
	 * @param  sim     Simulation.
	 * @param  lambda    Request mean interarrival time.
	 * @param  N    Number of requests.
	 * @param  prng    Pseudorandom number generator.
	 * @param  sourceNode  Server.
	 * @param V - number of nodes
	 */
	public Generator(Simulation sim, double lambda, int N, Random prng, Node[] nodes, int
V)
	{
		this.sim = sim;
		this.N = N;
		this.lambdaPrng = new ExponentialPrng (prng, lambda);
		this.nodes = nodes;
		this.V = V;
		respTimeSeries = new ListSeries();
		n = 0;

		generateQuery();
	}

	/**
	 * Generate the next request.
	 */
	private void generateQuery()
	{
		// starting node should be chosen uniformly at random here,
		// where the query is being added
		Query query = new Query (sim, respTimeSeries, V);
		sourceNode = nodes[query.getSource()];
		sourceNode.add(query);
		++n;
		if (n < N)
		{
			sim.doAfter (lambdaPrng.next(), new Event()
			{
				public void perform()
				{
					generateQuery();
```

```
                                        }
                                });
                        }
                }

                /**
                 * Returns a data series containing the response time statistics of the
                 * generated requests.
                 *
                 * @return  Response time series.
                 */
                public Series responseTimeSeries()
                {
                        return respTimeSeries;
                }

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

                /**
                 * Returns the drop fraction of the generated requests.
                 */
                public double dropFraction()
                {
                        return (double)(N - respTimeSeries.length())/(double)N;
                }
}



import java.util.ArrayList;

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

```
/**
 * A class to store the attributes of a node representing a distributed
 * database and to process queries traveling from between nodes.
 *
 * @author Joseph Ville
 *
 */
public class Node
{
	/**
	 * Whether to print the transcripts of query processing
	 */
	public static boolean transcript = false;

	private ExponentialPrng muPrng;
	private Simulation sim;
	private int id;
	private ArrayList<Node> neighbors;

	/**
	 * Construct a Node object
	 * @param id - the node's id
	 */
	public Node(int id)
	{
		this.id = id;
	}

	/**
	 * Construct a Node object
	 * @param id - the node's id
	 * @param sim - a Simulation
	 * @param mu - mean request processing time
	 * @param prng - pseudorandom number generator
	 */
	public Node(int id, Simulation sim, double mu, Random prng)
	{
		this.id = id;
		this.sim = sim;

		this.muPrng = new ExponentialPrng(prng, mu);
		neighbors = new ArrayList<Node>();
	}
```

```
/**
 * @return id of the current node
 */
public int id()
{
	return this.id;
}

/**
 * @return neighbors of the current node
 */
public ArrayList<Node> neighbors()
{
	return neighbors;
}

/**
 * Add the given query
 * @param query - Query
 */
public void add(Query query)
{
	if(transcript)
	{
		System.out.printf("%.3f %s added %d%n", sim.time(), query, this.id);
	}
	startProcessing(query);
}

/**
 * Start processing
 * @param query - Query
 */
private void startProcessing(Query query)
{
	if(transcript)
	{
		System.out.printf("%.3f %s starts processing%n", sim.time(), query);
	}
	sim.doAfter(muPrng.next(), new Event()
	{
		public void perform()
		{
			finishProcessing(query);
```

```
                        }
                });
        }

        /**
         * Finish processing
         * @param query - Query
         */
        private void finishProcessing(Query query)
        {
                if(transcript)
                {
                        System.out.printf ("%.3f %s finishes sending from %s to %s%n",
                                        sim.time(), query, query.getSource(), query.getDest());
                }
                Node next = nextHop();

                if(this.id == query.getDest())
                {
                        if(transcript)
                        {
                                System.out.printf("%.3f %s finishes processing%n", sim.time(),
query);
                        }
                        query.finish();
                }
                else
                {
                        next.add(query);
                }
        }

        /**
         * Whether an edge exists between this node and the node
         * at the given index
         * @param v
         * @return true - if there is an edge
         *                 false - otherwise
         */
        public boolean edgeExists(Node v)
        {
                if(neighbors.contains(v))
                {
                        return true;
```

```
		}
		return false;
	}

	/**
	 * Choose the next hop from the list of this node's neighbors
	 * @return the index of the next hop
	 */
	public Node nextHop()
	{
		java.util.Random rand = new java.util.Random();
		int next = rand.nextInt(this.neighbors.size());
		return this.neighbors.get(next);
	}

	/**
	 * Adds a vertex number to the end of this vertex's neighbor list
	 * @param n - the neighbor to be added
	 */
	public void addNeighbor(Node n)
	{
		this.neighbors.add(n);
	}

	/**
	 * Determine whether the current node equals the node passed in
	 * @param n
	 * @return true - if they are equal
	 *			false - otherwise
	 */
	public boolean equals(Node n)
	{
		if(this.id == n.id)
		{
			return true;
		}
		return false;
	}
}
```

```
import java.util.Random;

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

/**
 * Class Query provides a query in the the distributed database
 * querying simulations
 *
 * @author Joseph Ville
 */
public class Query
{
	private static int idCounter = 0;

	private int id;
	private Simulation sim;
	private double startTime;
	private double finishTime;
	private ListSeries respTimeSeries;
	private int V;

	private int source; // index of the source node
	private int dest; // index of the destination node

	private Random rand;

	/**
	 * Construct a new request. The request's start time is set to the current
    * simulation time.
    * @param  sim - Simulation.
	 */
	public Query(Simulation sim)
	{
		this.sim = sim;
		this.id = ++idCounter;
		this.startTime = sim.time();
	}

	/**
	 * 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.
	 * series.
```

```
	 * @param sim - Simulation
	 * @param series - Response time series
	 * @param V - number of vertices
	 */
	public Query(Simulation sim, ListSeries series, int V)
	{
		this (sim);
		this.respTimeSeries = series;
		this.V = V;
		rand = new Random();
		source = rand.nextInt(V);
		dest = rand.nextInt(V);
	}

	/**
	 * To reset the id counter after all the trials have been performed for a
	 * one sweep
	 */
	public static void resetIdCounter()
	{
		idCounter = 0;
	}

	/**
	 * @return the index of the source node of this query object
	 */
	public int getSource()
	{
		return source;
	}

	/**
	 * @returnthe index of the destination node of this query object
	 */
	public int getDest()
	{
		return dest;
	}

	/**
	 * 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 "Query " + id;
	}
}
```

4. According to the results displayed in question 5, when $p = 0.0$, the average total time is the highest. This is because there is no rewiring happening, so the graph generated is a 2-regular graph. Vertices are only connected to other nearby vertices. A lower value for p corresponds to a higher average simulation time because a randomly chosen destination node is likely to be a greater number of hops from the source node when the p-value is low. Fewer rewired edges will require more hops before the query reaches the destination node. The query is more likely to have to travel around the “lattice” structure, rather than being able to cut across the lattice on a rewired edge, saving hops.

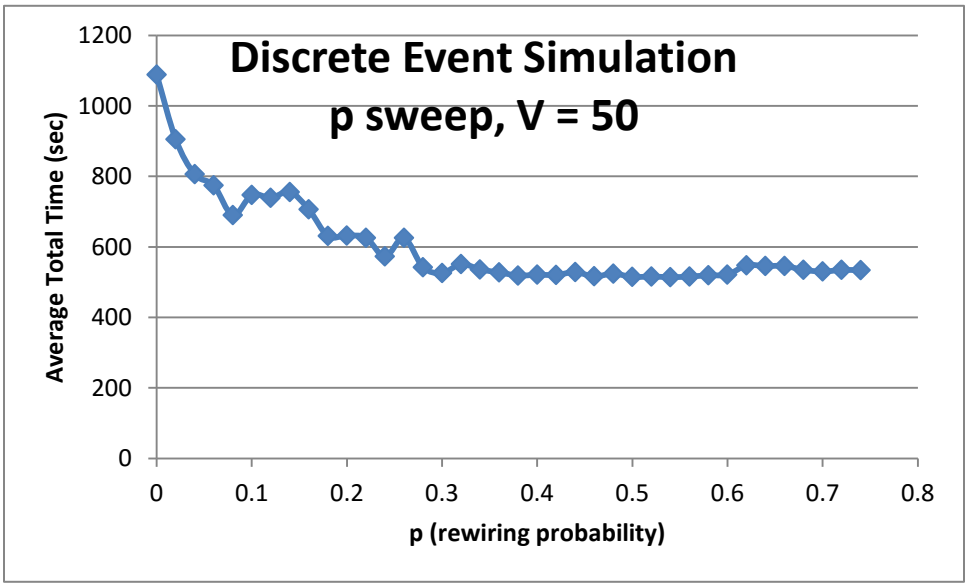
The general trend is that the average total time decreases sharply until p is about 0.1, then spikes up and back down around $p = 0.14$, then becomes more gradual. When p is around 0.3, the time nearly levels off for the remainder of the p sweep.

I am unsure of the reason for the spike in average time at $p = 0.14$. However, this happened consistently in all 6 runs of the program, so it is unlikely to be mere coincidence.

```
5. java PSweepDiscEventSim 87943431 50 2 0.0 0.75 0.02 1.0 2.0 100
1000 > out/p_sweep/v_50_out_p.txt
```

p	avg total time
0.000	1088.62276
0.020	905.15031
0.040	805.72649
0.060	774.21150
0.080	689.55895
0.100	747.18672
0.120	739.15801
0.140	755.36230
0.160	705.93511
0.180	630.66830
0.200	632.14513
0.220	625.73785
0.240	572.46906
0.260	625.16895
0.280	542.23357
0.300	525.69218
0.320	550.97129
0.340	535.95258
0.360	527.35262
0.380	518.61778
0.400	520.65815
0.420	520.41799
0.440	528.00768
0.460	516.10329
0.480	523.89128
0.500	514.62970
0.520	515.83327
0.540	514.03441
0.560	515.75954
0.580	519.24211
0.600	521.26270
0.620	547.27482
0.640	545.56750
0.660	545.68377
0.680	534.73142
0.700	529.85077
0.720	534.46037

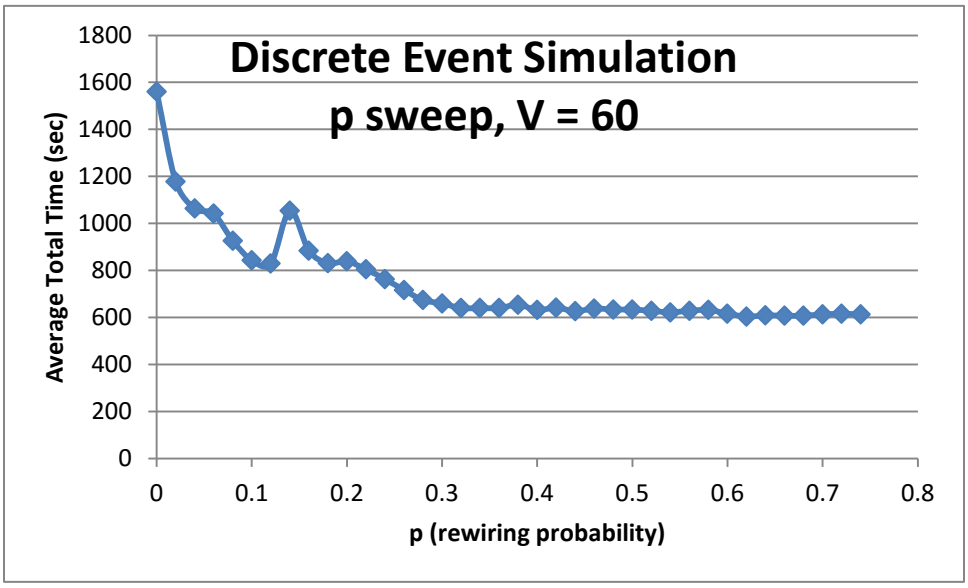
0.740 533.47816




```
java PSweepDiscEventSim 87943431 60 2 0.0 0.75 0.02 1.0 2.0 100
1000 > out/p_sweep/v_60_out_p.txt
```

p	avg total time
0.000	1560.39504
0.020	1177.47976
0.040	1062.74473
0.060	1040.95928
0.080	926.03434
0.100	842.79145
0.120	829.23607
0.140	1054.42263
0.160	884.10748
0.180	830.81949
0.200	838.20572
0.220	804.06946
0.240	762.84499
0.260	716.37783
0.280	673.74389
0.300	659.32550
0.320	639.72768
0.340	639.38412
0.360	639.74175
0.380	653.75488
0.400	631.35782
0.420	641.23182
0.440	626.37033

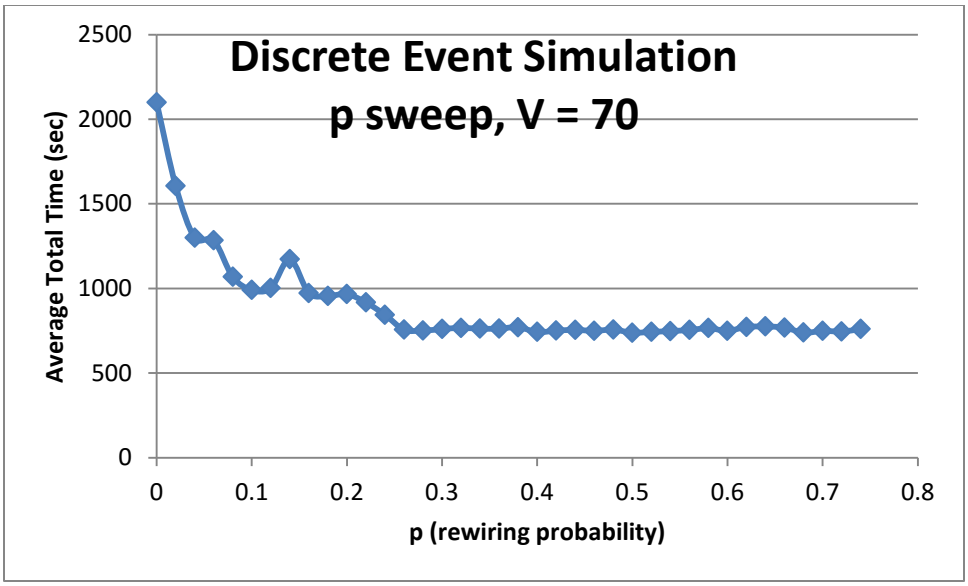
0.460	637.74771
0.480	633.56203
0.500	632.90277
0.520	628.05688
0.540	621.53456
0.560	628.26260
0.580	631.88963
0.600	615.82996
0.620	603.28730
0.640	608.40064
0.660	607.12220
0.680	607.20646
0.700	612.86215
0.720	615.14905
0.740	613.01467




```
java PSweepDiscEventSim 87943431 70 2 0.0 0.75 0.02 1.0 2.0 100
1000 > out/p_sweep/v_70_out_p.txt
```

p	avg total time
0.000	2098.36312
0.020	1605.56891
0.040	1299.53521
0.060	1284.32871
0.080	1067.88518
0.100	991.00180
0.120	1001.60831
0.140	1171.97856
0.160	972.77812

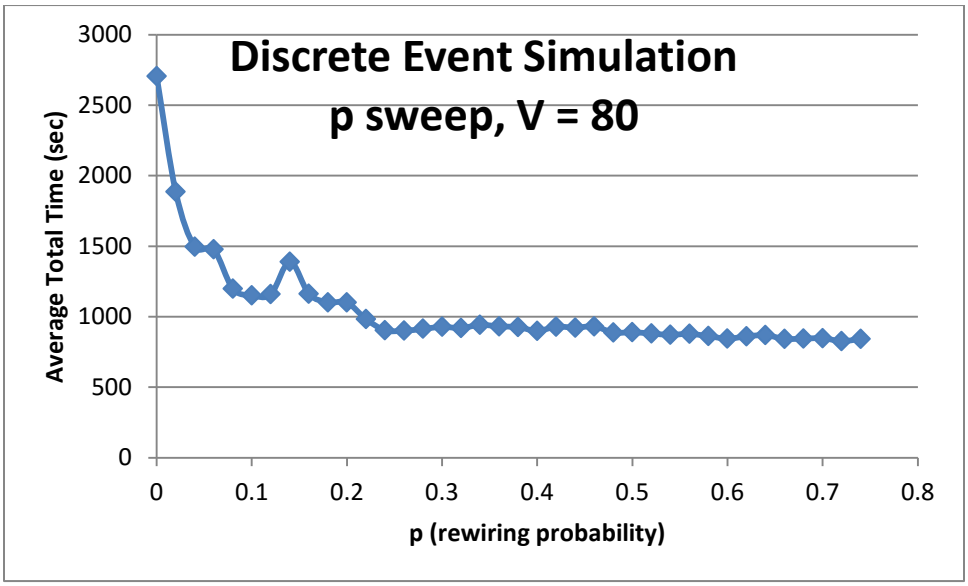
0.180	955.64380
0.200	965.93460
0.220	917.23974
0.240	843.96832
0.260	756.25112
0.280	750.89228
0.300	760.07539
0.320	766.49679
0.340	761.94603
0.360	762.74320
0.380	770.52268
0.400	742.35868
0.420	750.27557
0.440	754.87973
0.460	749.26821
0.480	756.43267
0.500	738.17293
0.520	743.01141
0.540	747.94378
0.560	754.22545
0.580	765.05611
0.600	749.67695
0.620	772.45891
0.640	774.85839
0.660	768.71348
0.680	738.71397
0.700	748.53480
0.720	745.57877
0.740	759.58189




```
java PSweepDiscEventSim 87943431 80 2 0.0 0.75 0.02 1.0 2.0 100
1000 > out/p_sweep/v_80_out_p.txt
```

p	avg total time
0.000	2705.01184
0.020	1885.84351
0.040	1495.44873
0.060	1476.29714
0.080	1199.11885
0.100	1151.04523
0.120	1159.05367
0.140	1389.74190
0.160	1162.19474
0.180	1100.88769
0.200	1100.25502
0.220	982.45596
0.240	903.98993
0.260	899.86961
0.280	913.61119
0.300	927.48864
0.320	919.65499
0.340	941.58986
0.360	931.42580
0.380	925.61606
0.400	899.05912
0.420	928.97086
0.440	922.13658
0.460	930.62621

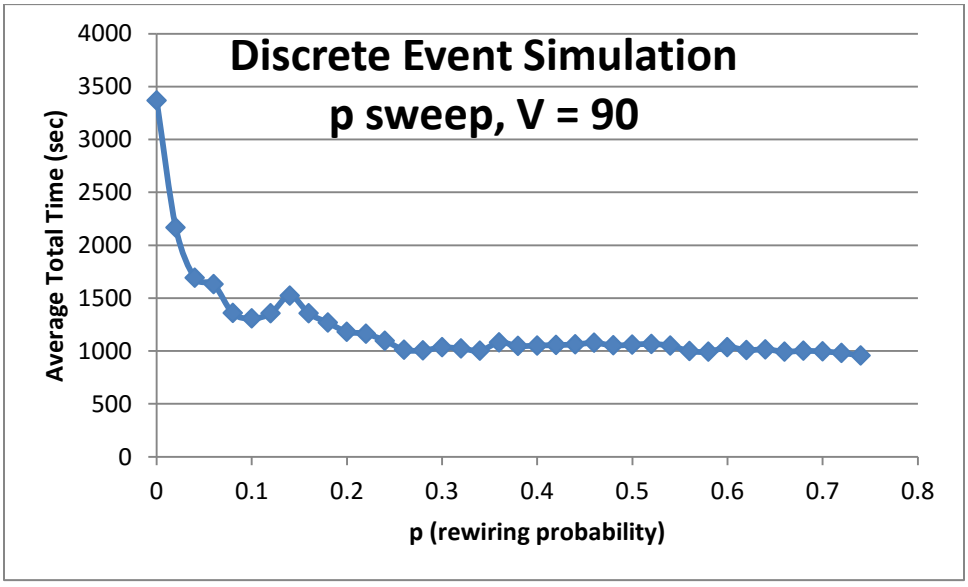
0.480	887.50875
0.500	889.21861
0.520	880.54924
0.540	871.09010
0.560	878.45730
0.580	861.66665
0.600	844.30311
0.620	860.51745
0.640	868.61390
0.660	842.70705
0.680	844.17122
0.700	847.52233
0.720	826.65974
0.740	841.87726




```
java PSweepDiscEventSim 87943431 90 2 0.0 0.75 0.02 1.0 2.0 100
1000 > out/p_sweep/v_90_out_p.txt
```

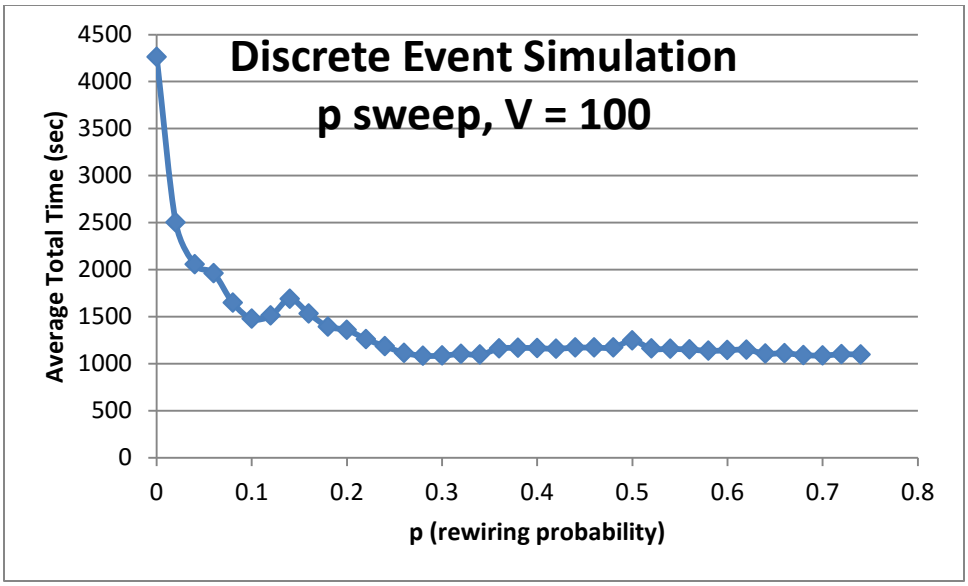
p	avg total time
0.000	3368.67684
0.020	2167.80773
0.040	1691.96407
0.060	1629.59343
0.080	1357.23643
0.100	1306.57833
0.120	1357.06573
0.140	1523.47918
0.160	1355.29701
0.180	1266.71308

0.200	1180.96364
0.220	1163.19317
0.240	1096.69007
0.260	1011.11574
0.280	1004.51878
0.300	1033.80508
0.320	1023.08650
0.340	1002.74688
0.360	1080.51210
0.380	1046.64281
0.400	1051.40876
0.420	1055.64551
0.440	1062.02359
0.460	1076.38647
0.480	1051.98487
0.500	1058.85681
0.520	1066.36848
0.540	1049.78054
0.560	999.23584
0.580	991.46372
0.600	1036.28309
0.620	1008.17817
0.640	1015.01130
0.660	993.48846
0.680	1000.85380
0.700	994.73948
0.720	980.66492
0.740	957.05032



```
java PSweepDiscEventSim 87943431 100 2 0.0 0.75 0.02 1.0 2.0 100
1000 > out/p_sweep/v_100_out_p.txt
```

p	avg total time
0.000	4262.07968
0.020	2502.09275
0.040	2055.67092
0.060	1960.12523
0.080	1646.63738
0.100	1476.53345
0.120	1510.16878
0.140	1687.19301
0.160	1531.06903
0.180	1393.04885
0.200	1358.02563
0.220	1260.72171
0.240	1184.10092
0.260	1114.16524
0.280	1083.01586
0.300	1084.38172
0.320	1103.37981
0.340	1094.67664
0.360	1160.53486
0.380	1167.49495
0.400	1164.49571
0.420	1157.84641
0.440	1170.21033
0.460	1170.15368
0.480	1171.83868
0.500	1244.56043
0.520	1159.74974
0.540	1155.83350
0.560	1150.47224
0.580	1135.59115
0.600	1142.34401
0.620	1148.80676
0.640	1105.10747
0.660	1110.94937
0.680	1089.93953
0.700	1085.64240
0.720	1099.90614
0.740	1095.21502

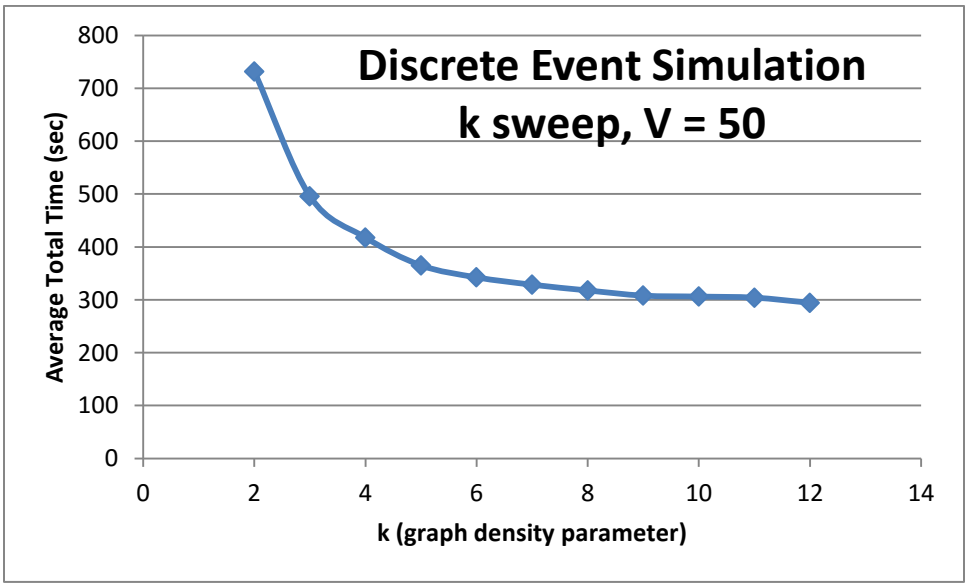


6. The lower the *k*-value for a graph, the less dense the graph will be. According to the results in question 7, a lower *k*-value also means a higher average simulation time. By definition, a high-density graph has a high fraction of the possible number of edges. This means the higher the graph density, the more likely a query is to reach its destination node with fewer hops. Fewer hops means less time, so the simulation would run faster.

In the results, average total time decreased drastically when k increased from 2 to 3, and the slope becomes more gradual the higher *k* gets. This is because when *k* = 2, there are relatively few edges, but as *k* increases, paths between nodes become more numerous, and the query will more frequently choose shorter paths. There comes a point when adding more edges has little effect on the time because there are already so many.

7. `java KSweepDiscEventSim 87943431 50 2 12 0.1 1.0 2.0 100 1000 > out/k_sweep/v_50_out_k.txt`

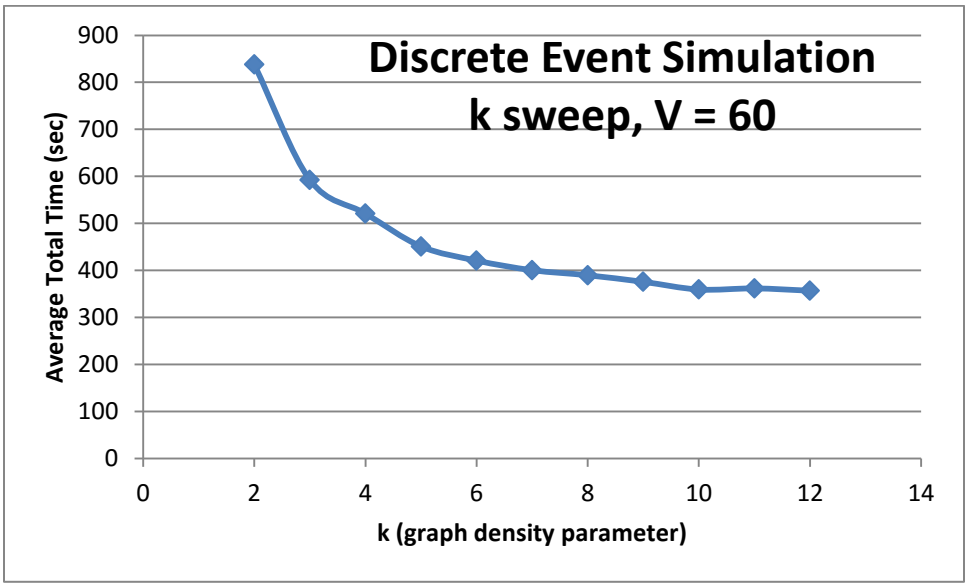
k	avg total time
2	731.90974
3	495.88645
4	417.78316
5	364.98134
6	342.35632
7	328.51018
8	317.59435
9	307.82574
10	306.06357
11	303.99176
12	294.25172



`java KSweepDiscEventSim 87943431 60 2 12 0.1 1.0 2.0 100 1000 > out/k_sweep/v_60_out_k.txt`

k	avg total time
2	837.79845
3	592.44024
4	520.87523
5	450.91191
6	420.72962
7	400.33948
8	389.52392
9	375.53595
10	359.21213
11	361.59038

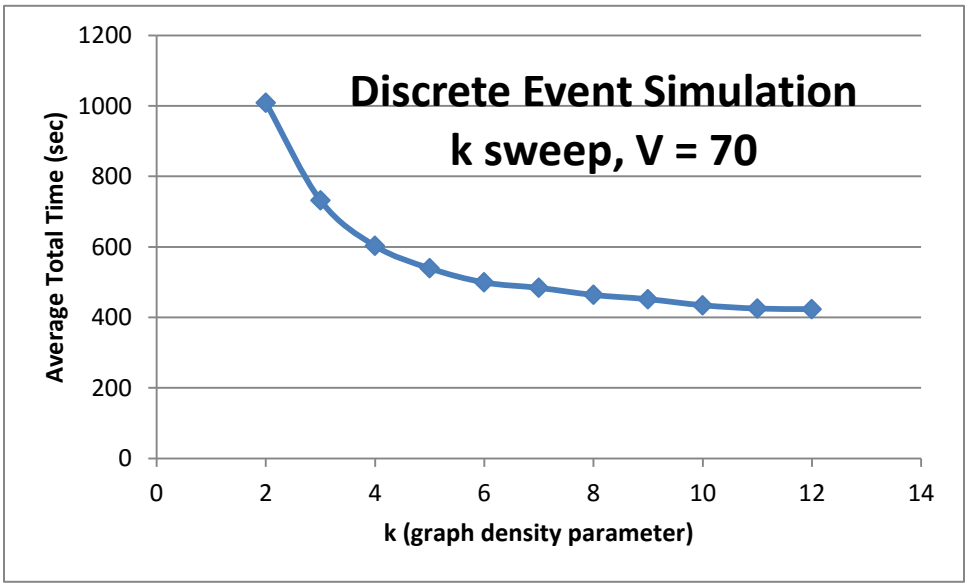
12 356.71575




```
java KSweepDiscEventSim 87943431 70 2 12 0.1 1.0 2.0 100 1000 >
out/k_sweep/v_70_out_k.txt
```

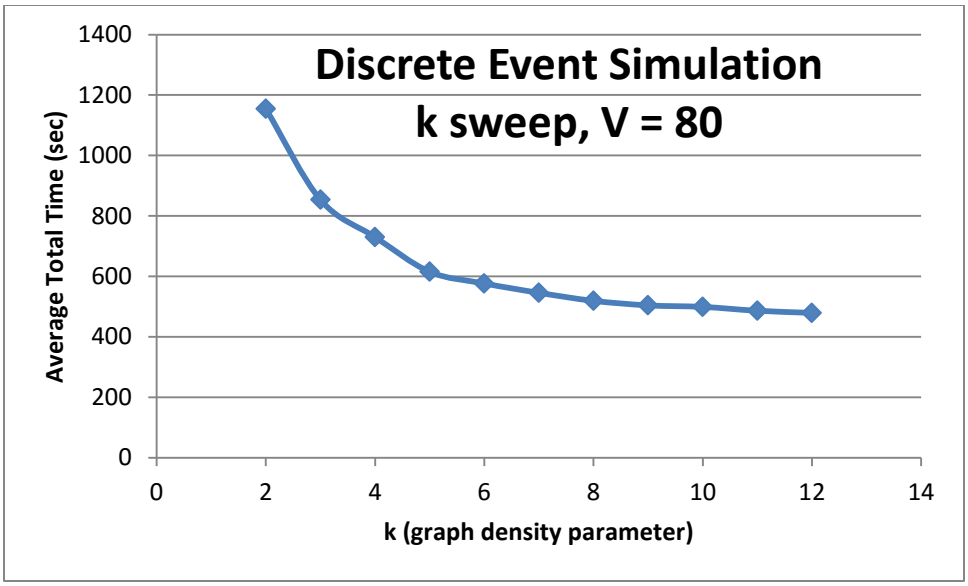
k avg total time
2 1008.57254
3 731.52518
4 602.50581
5 539.12818
6 499.01452
7 483.62237
8 463.53828
9 451.42488
10 434.27020
11 424.96512

12 423.39402



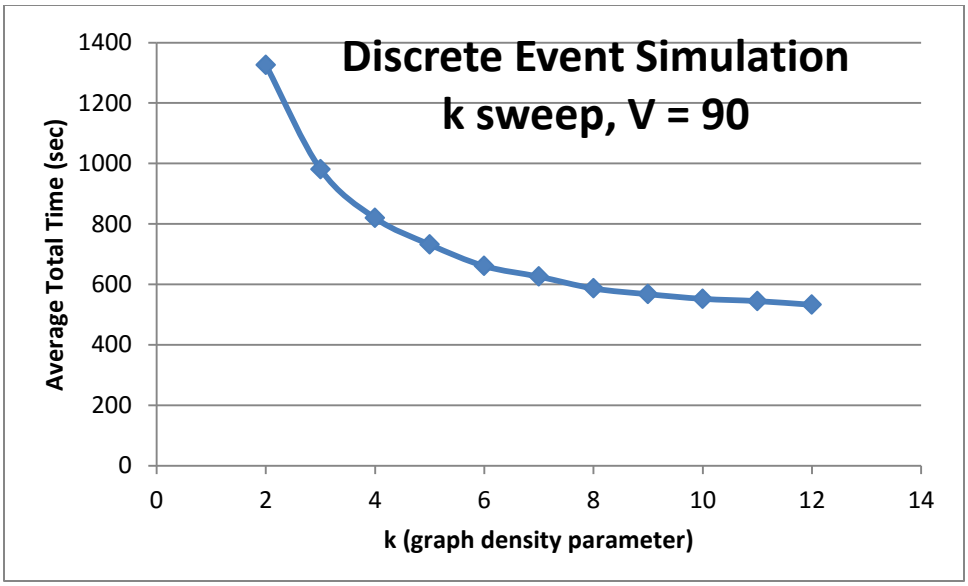

```
java KSweepDiscEventSim 87943431 80 2 12 0.1 1.0 2.0 100 1000 >
out/k_sweep/v_80_out_k.txt
```

k avg total time
2 1154.02111
3 853.21503
4 729.93153
5 615.12657
6 576.42857
7 545.41044
8 518.89172
9 503.93329
10 498.88723
11 486.09843
12 479.05880



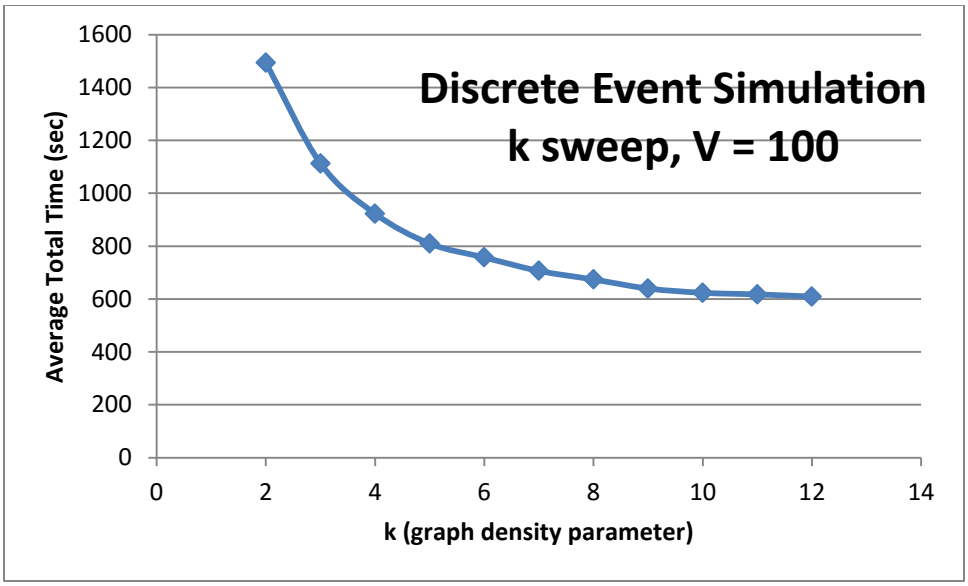

```
java KSweepDiscEventSim 87943431 90 2 12 0.1 1.0 2.0 100 1000 >
out/k_sweep/v_90_out_k.txt
```

k	avg total time
2	1325.88576
3	981.04046
4	819.26526
5	731.66891
6	660.48990
7	625.46762
8	586.27154
9	567.25550
10	551.94102
11	544.52937
12	532.27469




```
java KSweepDiscEventSim 87943431 100 2 12 0.1 1.0 2.0 100 1000 >
out/k_sweep/v_100_out_k.txt
```

k	avg total time
2	1493.64125
3	1112.67326
4	922.55972
5	809.62671
6	757.06624
7	706.35822
8	673.77118
9	639.62016
10	623.41264
11	617.35981
12	609.37791



8. In completing this project, I learned a great deal about discrete event simulation. At first I questioned what real world applications this would have. I now realize that it could be a very useful tool for estimating the amount of time it will take to perform many repetitions of the same task, provided that you know how long it takes to do the task once, on average. It could save companies money by giving more accurate estimates of how long something will take.

I also found it helpful to expand my knowledge of graphs and the applications of rewiring and/or increasing the number of edges.