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

There is one program that takes in an input graph and a choice of 4 different graph analysis algorithms: Closeness Centrality, Degree Distribution, Degree Centrality, and Connected Components.

Main:
Takes in a graph file and any evaluations wanted. The graph file is read in, and the internal state of the graph is created. After graph creation, the program will iterate through the different evaluations, and output the results for each.

Degree Distribution:
The degree distribution is used to determine what type of graph it is -- small world, or scale free. The function iterates through each vertex and adds the degree to a histogram. Then, the program iterates through the histogram and computes the probability and expected probability, and adds them to 3 separate XY series -- the results (degree, probability), exponential function (degree, log(probability)), and power function (log(degree), log(probability). After this, the graphs for the power function, and the exponential function are shown with their corresponding correlation values to allow the user to determine what type of graph it is.

Connected Components:
The connected components iterates through all the vertices in a graph with a breadth first search, and returns the list of vertices it has seen. The seen array is the number of connected components for the graph. The remaining vertices is then pruned of the returned seen values, and the next unseen vertex is evaluated, and the loop continues on the remaining vertices. The number of components are printed as they are found, so they will most likely be out of order.

Degree Centrality:
The degree centrality retrieves the largest component in the graph, then iterates through the vertices. For each vertex, the number of neighbors is found, and added to a TreeMap with the key being the neighbors, and the value being the vertex id. Since the TreeMap sorts values as they are added to the tree, the top 40 can be easily taken off the top of the structure.

Closeness Centrality:
The closeness centrality retrieves the largest component in the graph, then iterates through all of the vertices. For each vertice, a BFS that keeps track of the number of total hops between vertices. Once the BFS is completed, the total number of hops is divided by the size of the seen array, and this is the average distance for a given vertex. The average distance is then added to a TreeMap with the key being the average distance, and the value being the vertex. Since the

TreeMap sorts values as they are added to the tree, the top 40 can be easily taken off the top of the structure.

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

Commands:
There is one program, and it only requires the graph file, and the functions that are needed to be ran. The program will read the graph, and iterate over each function specified.

Standard command:
java pj2 GraphAnalysis <graphfile> <graph measurement>
<graphfile> = graph file name.
<graph measurement> = [degreeDistribution | connectedComponents | closenessCentrality | degreeCentrality] The graph measurement to be ran.

Multiple functions:
java GraphAnalysis CA-HepTh-graph.txt closenessCentrality degreeDistribution

Single functions:
java GraphAnalysis CA-HepTh-graph.txt closenessCentrality
java GraphAnalysis CA-HepTh-graph.txt degreeDistribution
java GraphAnalysis CA-HepTh-graph.txt degreeCentrality
java GraphAnalysis CA-HepTh-graph.txt connectedComponents

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

```
import java.io.*;
import java.util.*;
import edu.rit.numeric.Histogram;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.XYSeries;
import edu.rit.numeric.plot.Dots;
import edu.rit.numeric.plot.Plot;

import java.io.File;
import java.text.DecimalFormat;


/**
* Created by Tyler Paulsen on 2/20/2016.
* Class to compute various network science algorithms for a given
graph.
*/
public class GraphAnalysis {
   private HashMap<Integer, HashSet<Integer>> graph;
   private String graphfile;
   private int V;

   public static void main(String args[]) throws Exception{
       if (args.length < 2 || args.length > 5) usage();
       GraphAnalysis ga = new GraphAnalysis(args[0]);

       for(int i = 1; i < args.length; ++i){
           String arg = args[i];
           switch (arg){
               case "degreeDistribution":
                   ga.degreeDistribution();
                   break;
               case "connectedComponents":
                   ga.connectedComponents();
                   break;
               case "closenessCentrality":
                   ga.closenessCentrality();
                   break;
               case "degreeCentrality":
                   ga.degreeCentrality();
                   break;
               default:
                   usage();
           }
       }
   }

   GraphAnalysis(String f) throws IOException{
       graphfile = new String(f);
       graph = new HashMap();
       construct_graph(new File(f));
       //System.out.println(graph);
   }

   /**
    * Construct a graph from a file. Must conform to the spec:
https://cs.rit.edu/~ark/351/analysis/graphfile.shtml
```

```
     *  If the spec is not follow, results may vary.
     * @param fileName - name of the graph file
     */
    void construct_graph(File fileName)throws IOException{
        // The name of the file to open.
        try {
            // FileReader reads text files in the default encoding.
            FileReader fileReader = new FileReader(fileName);

            // Always wrap FileReader in BufferedReader.
            BufferedReader bufferedReader = new
BufferedReader(fileReader);
            String line[] = bufferedReader.readLine().split(" ");
            V = Integer.parseInt(line[1]);
            int E = Integer.parseInt(line[2]);

            for (int i = 0; i < V; ++i)
                graph.put(i, new HashSet());
            int v1,v2;
            for (int i = 0; i < E; ++i){
                line = bufferedReader.readLine().split(" ");
                if(line[0].equals("e")) {
                    v1 = Integer.parseInt(line[1]);
                    v2 = Integer.parseInt(line[2]);
                    graph.get(v1).add(v2);
                    graph.get(v2).add(v1);
                }
            }
            // Always close files.
            bufferedReader.close();
        }
        catch(FileNotFoundException ex) {
            System.out.println("Unable to open file: " + fileName);
            throw new FileNotFoundException();
        }
        catch(IOException ex) {
            System.out.println("Error reading file: " + fileName);
            throw new IOException();
        }
    }

    /**
```

```
     * Breadth first search for a given graph anc compute the average
distance.
     * @param src - What node to start from
     * @return  double : the average distance for a given node.
     */
    double BFS_avgDist(int src) {
        HashSet<Integer> seen = new HashSet();
        LinkedList<Path> queue = new LinkedList();
        Path A = new Path(src);
        queue.addLast(A);
        seen.add(A.id);
        int total_dist = 0;
        int count = 0;
        while (!queue.isEmpty()) {
            A = queue.poll();
            for (Integer B : graph.get(A.id)) {
                if (!seen.contains(B)) {
                    seen.add(B);
                    queue.addLast(new Path(B, A));
                    total_dist += A.dist;
                    count += 1;
                }
            }
        }
        return (double)total_dist/count;
    }

    /**
     * Breadth first search for a given graph.
     * @param start - What node to start from
     * @param dest - What node we are looking for.
     * @return
     */
    int BFS(int start, int dest) {
        if(graph.get(start).contains(dest)) return 1;
        HashSet<Integer> seen = new HashSet();
        LinkedList<Path> queue = new LinkedList();
        Path A = new Path(start);
        queue.addLast(A);
        seen.add(A.id);
        while (!queue.isEmpty()) {
            A = queue.poll();
            for (Integer B : graph.get(A.id)) {
                if (graph.get(B).contains(dest)) return A.dist + 1;
                if (!seen.contains(B)) {
                    seen.add(B);
                    queue.addLast(new Path(B, A));
                }
            }
        }
        return 0;
    }


    /**
     * BFS to find a component for a given node.
     * @param start - What node to start from
     * @return
     */
    HashSet<Integer> BFS_component(int start) {
        HashSet<Integer> seen = new HashSet();
        LinkedList<Path> queue = new LinkedList();
        Path A = new Path(start);
        queue.addLast(A);
        seen.add(A.id);
        while (!queue.isEmpty()) {
            A = queue.poll();
            for (Integer B : graph.get(A.id)) {
                if (!seen.contains(B)) {
                    seen.add(B);
                    queue.addLast(new Path(B, A));
                }
            }
        }
        return seen;
    }

    // Class used in the breadth first search.
    class Path {
        int dist, id;
        Path parent;

        //constructor, keeps tract of the distance by adding one to
the previous.
        Path(int _id, Path _parent) {
            id = _id;
            parent = _parent;
            dist = parent.dist + 1;
        }
        //initial constructor
        Path(int _id) {
            id = _id;
            dist = 1;
        }
    }

    /**
     * compute the degree centrality of a graph.
     */
    public void degreeCentrality(){
        // < vert id, degree >
        TreeMap<Integer,LinkedList<Integer>> results = new
TreeMap<>(Collections.reverseOrder());
        LinkedList<Integer> vertices = largestComponent();
        int src,degree;
        for (int i  = 0; i < vertices.size(); ++i){
            src = vertices.get(i);
            degree = graph.get(src).size();
            if (results.containsKey(degree))
                results.get(degree).add(src);
            else {
                LinkedList<Integer> v = new LinkedList();
                v.add(src);
                results.put(degree,v);
            }
        }

        System.out.println("Rank\tVertex\tDegCen");
        int rank = 1;
        for(Map.Entry<Integer,LinkedList<Integer>> entry :
results.entrySet()) {
            for ( Integer vertex : entry.getValue())
                System.out.printf("%d\t%d\t%d%n", rank++, vertex,
entry.getKey()) ;
            if(rank > 40) break;
        }
    }
```

```
   /**
    * compute the closeness centrality for the largest component in
the graph.
    */
   public void closenessCentrality(){
       TreeMap<Double,LinkedList<Integer>> results = new TreeMap<>();
       LinkedList<Integer> vertices = largestComponent();
       int src;
       double avg;
       for (int i  = 0; i < vertices.size(); ++i){
           src = vertices.get(i);
           avg = BFS_avgDist(src);
           if (results.containsKey(avg))
               results.get(avg).add(src);
           else {
               LinkedList<Integer> v = new LinkedList();
               v.add(src);
               results.put(avg,v);
           }
       }
       System.out.println("Rank\tVertex\tCloCen");
       int rank = 1;
       for(Map.Entry<Double,LinkedList<Integer>> entry :
results.entrySet()) {
           for ( Integer vertex : entry.getValue())
               System.out.printf("%d\t%d\t%f%n", rank++, vertex,
entry.getKey());
           if(rank > 40) break;
       }
   }

   /**
    * Find the number of calculated components for a given graph.
    */
   public void connectedComponents(){
       System.out.println("Comp\tSize");
       int comp = 0;
       int largest = -1;
       int smallest = V;
       LinkedList<Integer> vertices= new
LinkedList<>(graph.keySet());

       HashSet<Integer> seen;
```

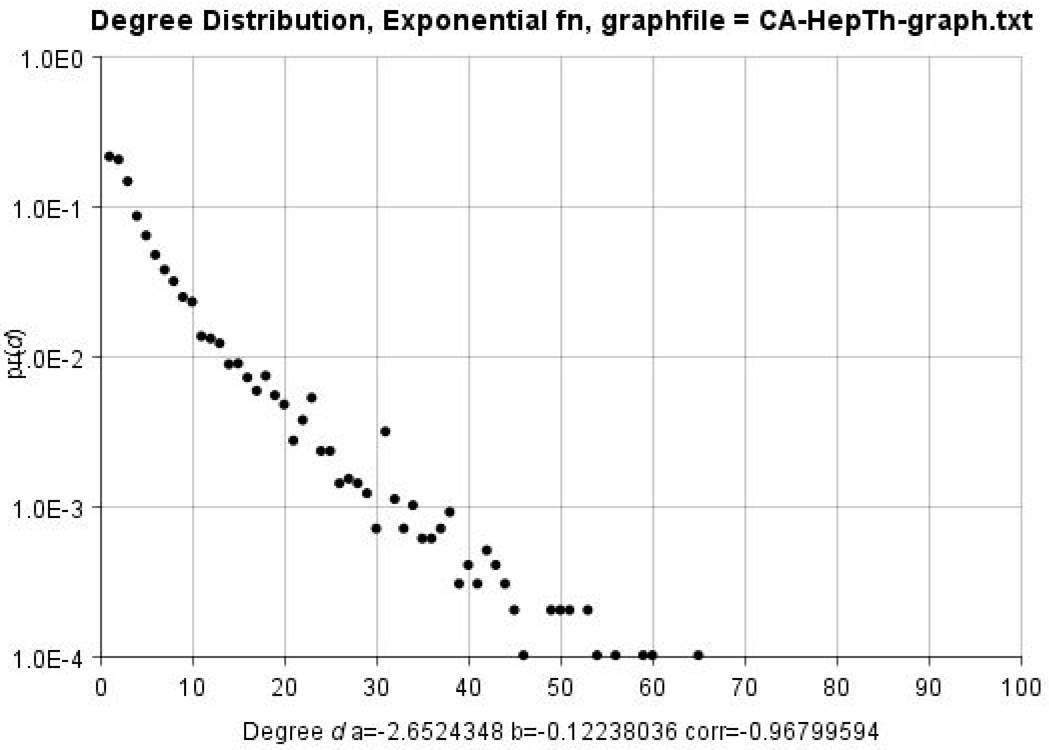
```
        while (!vertices.isEmpty()) {
            int src = vertices.poll();
            seen = BFS_component(src);
            vertices.removeAll(seen);
            int size = seen.size();
            System.out.printf("%d\t%d%n", comp++, size);
            if ( size < smallest) smallest =  size;
            if ( size  > largest ) largest = size;
        }
        System.out.println("Largest Component: " + largest);
        System.out.println("Smallest Component: " + smallest);
    }

    /**
     * get the largest component of the graph
     * @return List of vertices in the largest component.
     */
    public LinkedList<Integer> largestComponent(){
        LinkedList<Integer> vertices= new
LinkedList<>(graph.keySet());
        HashSet<Integer> largest = new HashSet<>();

        HashSet<Integer> seen;
        while (!vertices.isEmpty()) {
            int src = vertices.poll();
            seen = BFS_component(src);
            vertices.removeAll(seen);
            if ( largest.size() < seen.size() ) largest = seen;
        }
        return new LinkedList<Integer>(largest);
    }

    /**
     * calculate the degree distribution, and produce two plots of the
outcome -- a Power Fn, and an Exponential Fn plot
     * @throws IOException
     */
    public void degreeDistribution() throws IOException{
        //set up histogram

        Histogram hist = new Histogram(V);
        // Update histogram.
        for(int i = 0; i < V; ++i ) {
                hist.accumulate(graph.get(i).size());
        }

        // Print results, gather plot data.
        ListXYSeries results = new ListXYSeries();
        ListXYSeries exponential = new ListXYSeries();
        ListXYSeries power = new ListXYSeries();
        System.out.printf("d\tcount\tpr\texpect%n");
        for (int d = 0; d < hist.size(); ++d) {
            long count = hist.count(d);
            if (count == 0) continue;
            double pr = hist.prob(d);
            double expect = hist.expectedProb(d);
            results.add(d, pr);
            exponential.add(d, Math.log(pr));
            power.add(Math.log(d), Math.log(pr));
            System.out.printf("%d\t%d\t%.4e\t%.4e%n", d, count, pr,
expect);
        }

        XYSeries.Regression r = exponential.linearRegression();
        System.out.printf("exponential%na=%f\tb=%f\tcorr=%f%n", r.a,
r.b, r.corr);
        Plot exponentialplot = new Plot()
            .plotTitle(String.format("Degree Distribution, Exponential
fn, graphfile = %s", graphfile))
            .xAxisTitle("Degree <I>d</I> a=" + (float) r.a + " b=" +
(float) r.b + " corr=" + (float) r.corr)
            .xAxisLength(460)
            .yAxisTitle("pr(<I>d</I>)")
            .yAxisKind(Plot.LOGARITHMIC)
            .yAxisTickFormat(new DecimalFormat("0.0E0"))
            .yAxisLength(300)
            .seriesDots(Dots.circle(5))
            .seriesStroke(null)
            .xySeries(results);
        exponentialplot.getFrame().setVisible(true);
        r = power.linearRegression();
        System.out.printf("Power%na=%f\tb=%f\tcorr=%f%n", r.a, r.b,
r.corr);
        Plot powerplot = new Plot()
                .plotTitle(String.format("Degree Distribution, Power
fn, graphfile = %s", graphfile))
```

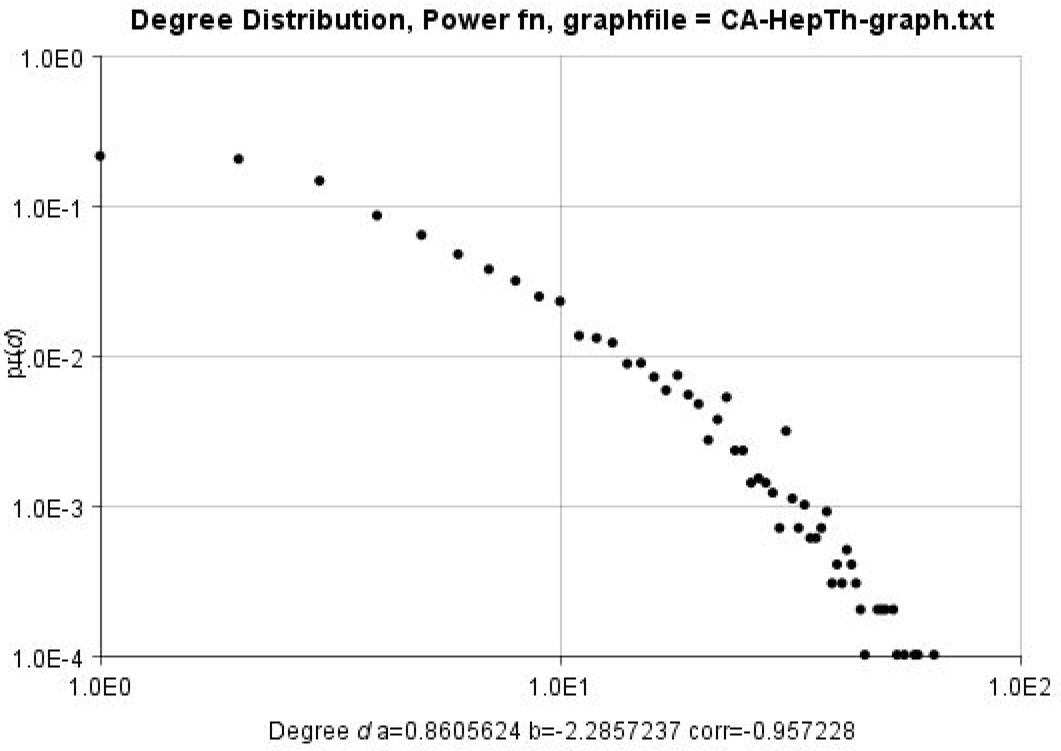
```
                .xAxisTitle("Degree <I>d</I> a="+(float)r.a + " b=" +
(float)r.b + " corr="+(float)r.corr)
                .xAxisLength(460)
                .yAxisTitle("pr(<I>d</I>)")
                .yAxisKind(Plot.LOGARITHMIC)
                .xAxisKind(Plot.LOGARITHMIC)
                .yAxisLength(300)
                .yAxisTickFormat(new DecimalFormat("0.0E0"))
                .xAxisTickFormat(new DecimalFormat("0.0E0"))
                .seriesDots(Dots.circle(5))
                .seriesStroke(null)
                .xySeries(results);

        powerplot.getFrame().setVisible(true);
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println ("Usage: java pj2 GraphAnalysis <graphfile>
<graph measurement>");
        System.err.println ("<graphfile> = graph file name");
        System.err.println ("<graph measurement> = [
degreeDistribution | connectedComponents | " +
                "closenessCentrality | degreeCentrality");
        throw new IllegalArgumentException();
    }

}
```

4. Does the HEP-TH graph more closely resemble a small-world graph or a scale-free graph? Include a mathematical justification for your answer, with data and plots. Include the exact command line(s) for your program(s) that you ran to answer this question.

Command Line: java GraphAnalysis CA-HepTh-graph.txt degreeDistribution

The graphs below show that the HEP-TH graph resembles a Small world graph due to the higher correlation coefficient on the exponential function.





5. How many connected components are there in the HEP-TH graph? Does the graph have a giant component? What is the size of the largest component? What is the size of the

smallest component? Include the exact command line(s) for your program(s) that you ran to answer this question, and include the actual output of your program(s).

Command Line: java GraphAnalysis CA-HepTh-graph.txt connectedComponents

There are 429 connected components in the graph with one giant component (87%) the largest component has 8638, and the smallest has 1.

Component	Size	Component	Size	Component	Size	Component	Size	Component	Size
0	8638	98	2	196	21	294	2	392	2
1	2	99	2	197	2	295	4	393	2
2	2	100	6	198	2	296	3	394	2
3	2	101	2	199	2	297	2	395	2
4	7	102	2	200	2	298	2	396	3
5	2	103	3	201	2	299	4	397	2
6	2	104	2	202	2	300	5	398	2
7	2	105	2	203	2	301	2	399	3
8	2	106	7	204	2	302	2	400	2
9	2	107	2	205	2	303	3	401	3
10	2	108	3	206	2	304	9	402	2
11	4	109	2	207	3	305	2	403	2
12	3	110	2	208	2	306	2	404	2
13	2	111	3	209	2	307	5	405	4
14	2	112	5	210	2	308	2	406	2
15	5	113	6	211	2	309	4	407	3
16	7	114	2	212	5	310	2	408	2
17	2	115	3	213	7	311	4	409	2
18	4	116	3	214	2	312	2	410	2
19	2	117	2	215	4	313	4	411	2
20	2	118	3	216	2	314	2	412	2
21	4	119	3	217	3	315	2	413	2
22	2	120	2	218	2	316	2	414	4
23	3	121	4	219	4	317	4	415	4
24	11	122	2	220	3	318	4	416	2

25	2	123	4	221	2	319	2	417	3
26	3	124	3	222	4	320	2	418	2
27	2	125	2	223	4	321	2	419	2
28	2	126	2	224	2	322	1	420	2
29	5	127	5	225	2	323	2	421	2
30	4	128	2	226	2	324	3	422	3
31	6	129	3	227	2	325	2	423	2
32	7	130	2	228	2	326	2	424	2
33	2	131	4	229	2	327	2	425	2
34	2	132	2	230	3	328	3	426	2
35	3	133	3	231	2	329	2	427	2
36	2	134	2	232	3	330	1	428	3
37	3	135	2	233	2	331	2		
38	8	136	2	234	3	332	3		
39	2	137	5	235	4	333	2		
40	2	138	3	236	3	334	2		
41	2	139	4	237	2	335	2		
42	2	140	2	238	3	336	2		
43	2	141	9	239	2	337	2		
44	7	142	4	240	5	338	2		
45	2	143	5	241	5	339	3		
46	2	144	7	242	2	340	2		
47	6	145	2	243	2	341	2		
48	5	146	2	244	2	342	2		
49	4	147	5	245	2	343	3		
50	2	148	4	246	2	344	3		
51	2	149	2	247	4	345	2		
52	5	150	7	248	2	346	2		
53	4	151	2	249	4	347	2		
54	4	152	2	250	4	348	2		
55	5	153	2	251	2	349	2		
56	2	154	3	252	4	350	2		
57	2	155	2	253	2	351	2		
58	2	156	4	254	2	352	2		

59	2	157	2	255	2	353	2		
60	2	158	3	256	2	354	3		
61	3	159	2	257	2	355	4		
62	2	160	6	258	2	356	2		
63	2	161	2	259	3	357	2		
64	2	162	2	260	2	358	2		
65	2	163	3	261	3	359	3		
66	8	164	2	262	2	360	3		
67	5	165	2	263	3	361	3		
68	5	166	2	264	2	362	2		
69	2	167	2	265	8	363	2		
70	2	168	2	266	4	364	2		
71	5	169	2	267	2	365	2		
72	3	170	2	268	2	366	2		
73	2	171	4	269	3	367	2		
74	4	172	2	270	2	368	2		
75	2	173	3	271	3	369	2		
76	2	174	3	272	7	370	2		
77	8	175	2	273	6	371	2		
78	2	176	2	274	2	372	3		
79	4	177	2	275	3	373	2		
80	5	178	2	276	2	374	4		
81	2	179	2	277	4	375	2		
82	2	180	2	278	2	376	2		
83	2	181	2	279	2	377	2		
84	2	182	3	280	5	378	3		
85	2	183	2	281	2	379	2		
86	2	184	3	282	3	380	2		
87	3	185	2	283	2	381	4		
88	2	186	5	284	3	382	2		
89	3	187	2	285	3	383	2		
90	7	188	7	286	4	384	2		
91	2	189	2	287	2	385	2		
92	2	190	4	288	2	386	2		

93	2	191	2	289	6	387	2		
94	8	192	4	290	2	388	2		
95	2	193	3	291	2	389	2		
96	4	194	2	292	6	390	3		
97	8	195	2	293	2	391	2		

6. What are the top-40-ranked vertices in the largest connected component of the HEP-TH graph with respect to degree centrality? Include a table of data to justify your answer. Include the exact command line(s) for your program(s) that you ran to answer this question.

Command: java GraphAnalysis CA-HepTh-graph.txt degreeCentrality

Rank	Vertex	DegCen
1	86	65
2	15	60
3	54	59
4	920	56
5	163	54
6	35	53
7	1546	53
8	1420	51
9	1873	51
10	8	50
11	38	50
12	2	49
13	277	49
14	67	46
15	40	45
16	1907	45
17	220	44
18	252	44

19	1305	44
20	32	43
21	386	43
22	442	43
23	590	43
24	1	42
25	62	42
26	141	42
27	1094	42
28	1332	42
29	26	41
30	76	41
31	1904	41
32	13	40
33	527	40
34	1461	40
35	1687	40
36	582	39
37	851	39
38	1231	39
39	36	38
40	173	38
41	224	38
42	600	38
43	806	38
44	889	38
45	1068	38
46	1256	38
47	3494	38

7. What are the top-40-ranked vertices in the largest connected component of the HEP-TH graph with respect to closeness centrality? Include a table of data to justify your answer.

Include the exact command line(s) for your program(s) that you ran to answer this question.

Command: java GraphAnalysis CA-HepTh-graph.txt closenessCentrality

Rank	Vertex	CloCen
1	15	4.045849
2	920	4.074216
3	40	4.095751
4	38	4.121107
5	1	4.147042
6	1461	4.154799
7	1094	4.163946
8	54	4.168924
9	1332	4.169851
10	442	4.183281
11	13	4.190228
12	8	4.194859
13	163	4.195322
14	30	4.211879
15	17	4.213268
16	62	4.221605
17	35	4.222184
18	277	4.222415
19	1420	4.228552
20	429	4.229015
21	14	4.240245
22	67	4.242445
23	1873	4.247771
24	582	4.248582
25	386	4.249508
26	31	4.250434
27	26	4.253676
28	252	4.259697

29	1443	4.264675
30	141	4.269191
31	851	4.272548
32	171	4.274517
33	618	4.278685
34	76	4.2832
35	173	4.291884
36	413	4.296515
37	590	4.298483
38	1546	4.300104
39	24	4.310293
40	806	4.313072

8. Based on the answers to the previous questions, discuss what your analysis reveals about collaborations between high energy physics theory researchers.

Since this is a small world graph, a few researches have ties to many different papers. The connected components shows that there is a large percentage of researchers who collaborate together (87.46%), and few researchers are not able to be a part of this group (~12%). The degree distribution helps reinforce the theory that few researches contribute to many of the papers with the top 40 having over 38 papers each. The closeness centrality tends to make me believe that there is a group effort among the top 40 researchers since they all have a 4.0 - 4.3 average distance. The close average distance shows that the top 40 researchers they can be linked to other authors on average within 4 papers.

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

Breadth first search is not just for a path. I was having a tough time making the runtime of the closeness centrality to run in a reasonable amount of time until I was pointed in the right direction by the professor. Most of the functions I previously wrote could be changed for a huge speed up using a different type of BFS.

The analysis about the graph showed me how to make different assumptions based on the calculations output. The analysis helped me able to visualize how a community acts without much information. Would be curious to see the metrics larger companies (facebook, google, etc) use to analyze their graphs.