STANFORD UNIVERSITY
Stanford Center for Professional Development

HOMEWORK ROUTE FORM

Stanford Center for Professional Development Student Information

Course No.	CS224W	Faculty / Instructor Name	Jure Leskovec	Date	11/11/2012
Student Name	Philip Scuderi			Phone	413-262-0693
Company		Email	pscuderi@gmail.com		
City.	Boston	State	MA	Country	USA

Check One: (●) Homework #: 3 ( ) Midterm ( ) Other

The email address provided on this form will be used to return homework, exams, and other documents and correspondence that require routing.

Total number of pages faxed including cover sheet 31

	For Stanford Use Only	
Date Received by the Stanford Center for Professional Development	Date Instructor returned graded project Score/Grade: (to be completed by instructor or by teaching assistant)	Date the Stanford Center for Professional Development returned graded project:

Please attach this route form to ALL MATERIALS and submit ALL to:

Stanford Center for Professional Development
496 Lomita Mall, Durand Building, Rm 410, Stanford, CA 94305-4036
Office 650.725.3015 | Fax 650.736.1266 or 650.725.4138
For homework confirmation, email scpd-distribution@lists.stanford.edu
http://scpd.stanford.edu

Last modified October 27, 2008

CS224w:
Social and Information Network Analysis

Assignment number: Homework 3

Submission time: 11:00 **and date:** 11/11/2012

Fill in and include this cover sheet with each of your assignments. It is an honor code violation to write down the wrong time. Assignments are due at 9:30 am, either handed in at the beginning of class or left in the submission box on the 1^{st} floor of the Gates building, near the east entrance.
Each student will have a total of *two* free late days. *One late day expires at the start of each class.* (Homeworks are usually due on Thursdays, which means the first late day expires on the following Tuesday at 9:30am.) Once these late days are exhausted, any assignments turned in late will be penalized 50% per late day. However, no assignment will be accepted more than *one* late day after its due date.

Your name: Philip Scuderi
Email: pscuderi@gmail.com **SUID:** 05855811

Collaborators: Olga Kapralova, Valentina Kroshlina

I acknowledge and accept the Honor Code.

(Signed) /Philip Scuderi/

(For CS224w staff only)

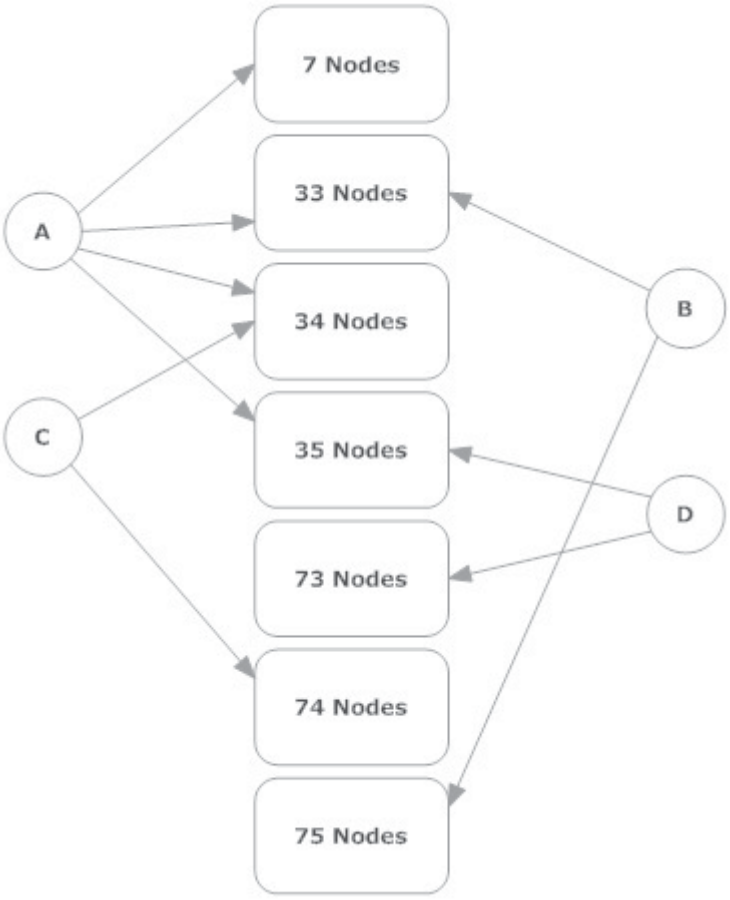
Late days: 1 2

Section	Score
1	
2	
3	
4	
5	
6	
Total	

Comments:

1(a) and 1(b)

In the following graph, each of the rectangular boxes indicates a group of n nodes, where n = the number displayed on the box. Each arrow from the circled nodes (A-D) to each rectangular box represents a directed edge from the circled node to each of the nodes in the group of nodes in the box. For example, node A has 7+33+34+35 = 109 directed edges (one directed edge to each of the nodes in the top 4 rectangular boxes). Each directed edge from any node n to any node m indicates that node n influences node m with probabiltiy 1.



Step 1 (i = 1) of Greedy Algorithm

Marginal gain of selecting A = 1 + 7 + 33 + 34 + 35 = 110

Marginal gain of selecting B = 1 + 33 + 75 = 109

Marginal gain of selecting C = 1 + 34 + 74 = 109

Marginal gain of selecting D = 1 + 35 + 73 = 109

Result: Greedy Algorithm selects node A

$S_1 = \{A, all\ nodes\ influenced\ by\ A\}$

Step 2 (i = 2) of Greedy Algorithm

Marginal gain of selecting B = 1+ 75 = 76

Marginal gain of selecting C = 1 + 74 = 75

Marginal gain of selecting D = 1 + 73 = 74

Result: Greedy Algorithm selects node B

$S_2 = \{A, B, all\ nodes\ influenced\ by\ A\ or\ B\}$

Step 3 (i = 3) of Greedy Algorithm

Marginal gain of selecting C = 1 + 74 = 75

Marginal gain of selecting D = 1 + 73 = 74

Result: Greedy Algorithm selects node C

$S_3 = \{A, B, C, all\ nodes\ influenced\ by\ A, B, or\ C\}$

1(a)

At after step 2 in the Greedy Algorithm detailed above (i = 2), we have the following:

$S_2 = \{A, B, all\ nodes\ influenced\ by\ A\ or\ B\}$

$\boldsymbol{f(S_2)} = 2 + 7 + 33 + 34 + 35 + 75 = \mathbf{186}$

We can select set T as follows:

$T = \{C, D, all\ nodes\ influenced\ by\ C\ or\ D\}$

$\boldsymbol{f(T)} = 2 + 34 + 35 + 73 + 74 = \mathbf{218}$

Thus,

$f(S_2) < f(T)$

1(b)

At after step 3 in the Greedy Algorithm detailed above (i = 3), we have the following:

$S_3 = \{A, B, C, all\ nodes\ influenced\ by\ A, B, or\ C\}$

$\boldsymbol{f(S_3)} = 3 + 7 + 33 + 34 + 35 + 74 + 75 = \mathbf{261.0}$

We can select set T as follows:

$T = \{B, C, D, all\ nodes\ influenced\ by\ B, C\ or\ D\}$

$\boldsymbol{(0.8)f(T)} = (0.8)(3 + 33 + 34 + 35 + 73 + 74 + 75) = (0.8)(327) = \mathbf{261.6}$

Thus,

$f(S_3) < (0.8)f(T)$

1(c)

The greedy hill climbing algorithm always outputs the optimal solution when:

At each step of the algorithm, the most influential node (u) selected does not influence any node from the previous steps (i.e., any node in S_{i-1}). In other words, the influence sets are all disjoint.

When this is the case at every step, at each step the greedy algorithm necessarily always has to select the next node with the largest influence set. Additionally, the selected node's influence set cannot destroy the greedy algorithm's ability to select the optimal node at subsequent steps.

1(d)

$f(S_k) + \sum_{i=1}^{k} \delta_i - f(T)$ (1.d.1)

Given equation (1.d.1) can be simplified as:

$f(S_k) + k\delta_i - f(T)$ (1.d.2)

Equation (1.d.2) can become arbitrarily large if we can increase the value of δ_i at each step. For example, take the graph used in problems 1(a) and 1(b).

$Let\ V = \{A, B, C, D\}$

Let N be all the nodes in each of the groups of nodes represented by the rectangular boxes in the graph (see the Figure presented in 1a and 1b). However, consider the number of nodes in each of the boxes to be a multiple of the number of nodes displayed on the box. For example, the topmost box contains 7*x

nodes, the second topmost box contains 33*x nodes, and so on. When x = 1, we have the example presented in problems 1(a) and 1(b).

Now, as we increase the value of x, we increase the value of δ_i at each step in the greedy algorithm. Thus, for a fixed value of k, we can increase the vlaue of given equation (1.d.1).

2(a)

$$f(x) = \left(\frac{\alpha-1}{x_{min}}\right)\left(\frac{x}{x_{min}}\right)^{-\alpha}$$

$$f(x) = \frac{\alpha-1}{(x_{min})^{1-\alpha}} x^{-\alpha}$$

$$F(x) = \frac{\alpha-1}{(x_{min})^{1-\alpha}} \int_{x_{min}}^{x} x^{-\alpha} dx$$

$$F(x) = \frac{\alpha-1}{(x_{min})^{1-\alpha}} \left(\frac{-x^{1-\alpha}}{\alpha-1} \Big|_{x_{min}}^{x} \right)$$

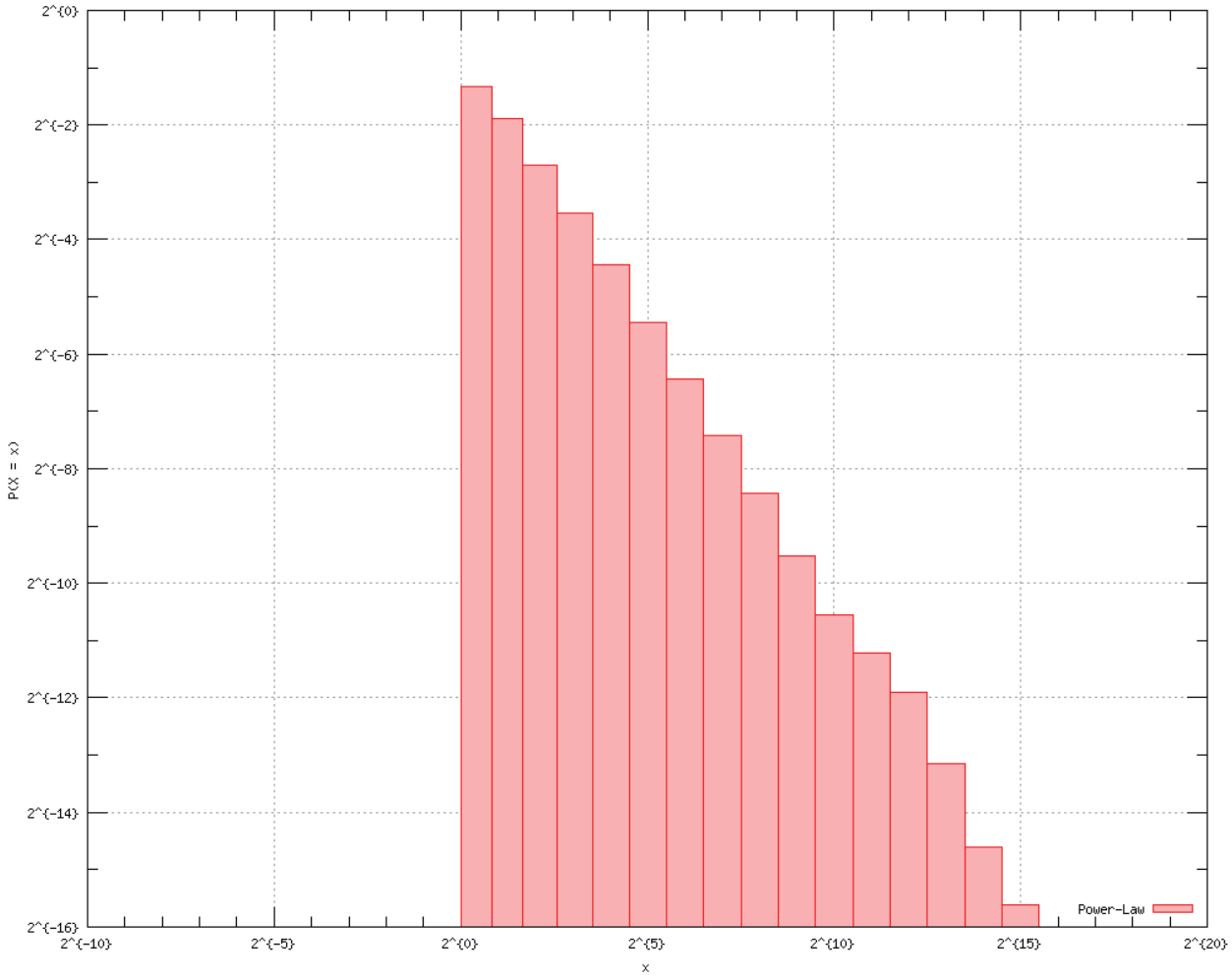
$$F(x) = \frac{1}{(x_{min})^{1-\alpha}} \left((x_{min})^{1-\alpha} - x^{1-\alpha} \right)$$

$$F(x) = 1 - \frac{x^{1-\alpha}}{(x_{min})^{1-\alpha}} = 1 - \left(\frac{x}{x_{min}}\right)^{1-\alpha}$$

$$\bar{F}(x) = 1 - F(x) = 1 - \left(1 - \left(\frac{x}{x_{min}}\right)^{1-\alpha}\right)$$

$$\bar{F}(x) = \left(\frac{x}{x_{min}}\right)^{1-\alpha}$$

2(b)



The above histogram was obtained using C++ and logarithmic binning. See Homework3.cpp, and in particular the function Q2().

2(c)

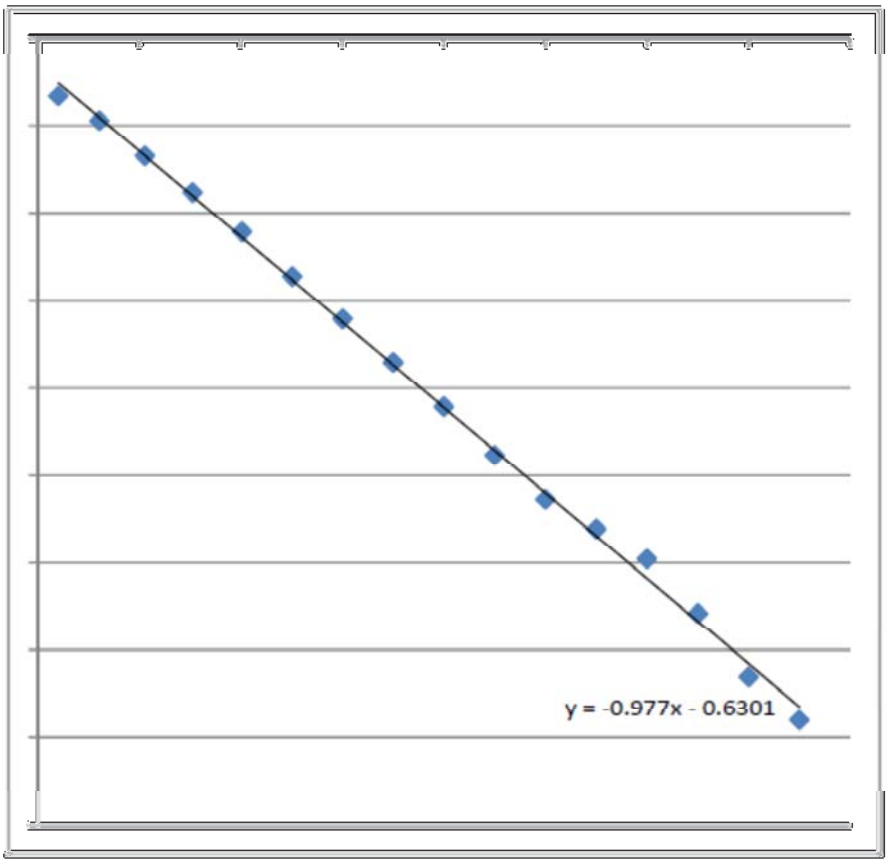


Figure 1: Least Squares Linear Regression on Log-Log Histogram Data

As you can see in Figure 1, $\alpha' \approx 0.977$.

We know that $\alpha = 1 + \alpha'$.

Therefore, $\alpha \approx 1.977$.

The Q2_Normalized_Pdf_Histogram.xml file was produced using C++ (see Homework3.cpp, and in particular the function Q2()). The above results were obtained by opening Q2_Normalized_Pdf_Histogram.xml in Excel, translating the Histogram data to a log-log scale, plotting the translated data, and running a least squares linear regression on the translated data.

2(d)

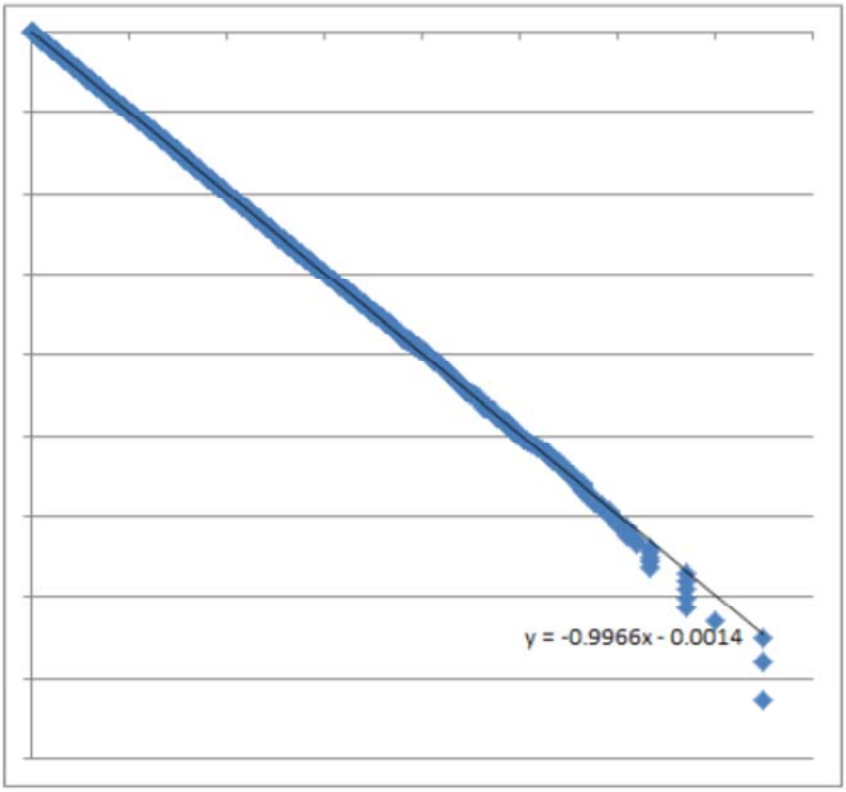


Figure 2: Least Squares Linear Regression on CCDF

As you can see in Figure 2, $\alpha' \approx 0.9966$.

We know that $\alpha = 1 + \alpha'$.

Therefore, $\alpha \approx 1.9966$.

The Q2_Normalized_Pdf_Histogram.xml file was produced using C++ (see Homework3.cpp, and in particular the function Q2()). The above results were obtained by opening Q2_Ccdf.xml in Excel, translating the CCDF data to a log-log scale, plotting the translated data, and running a least squares linear regression on the translated data.

2(e)

$$\hat{\alpha} = 2.00137$$

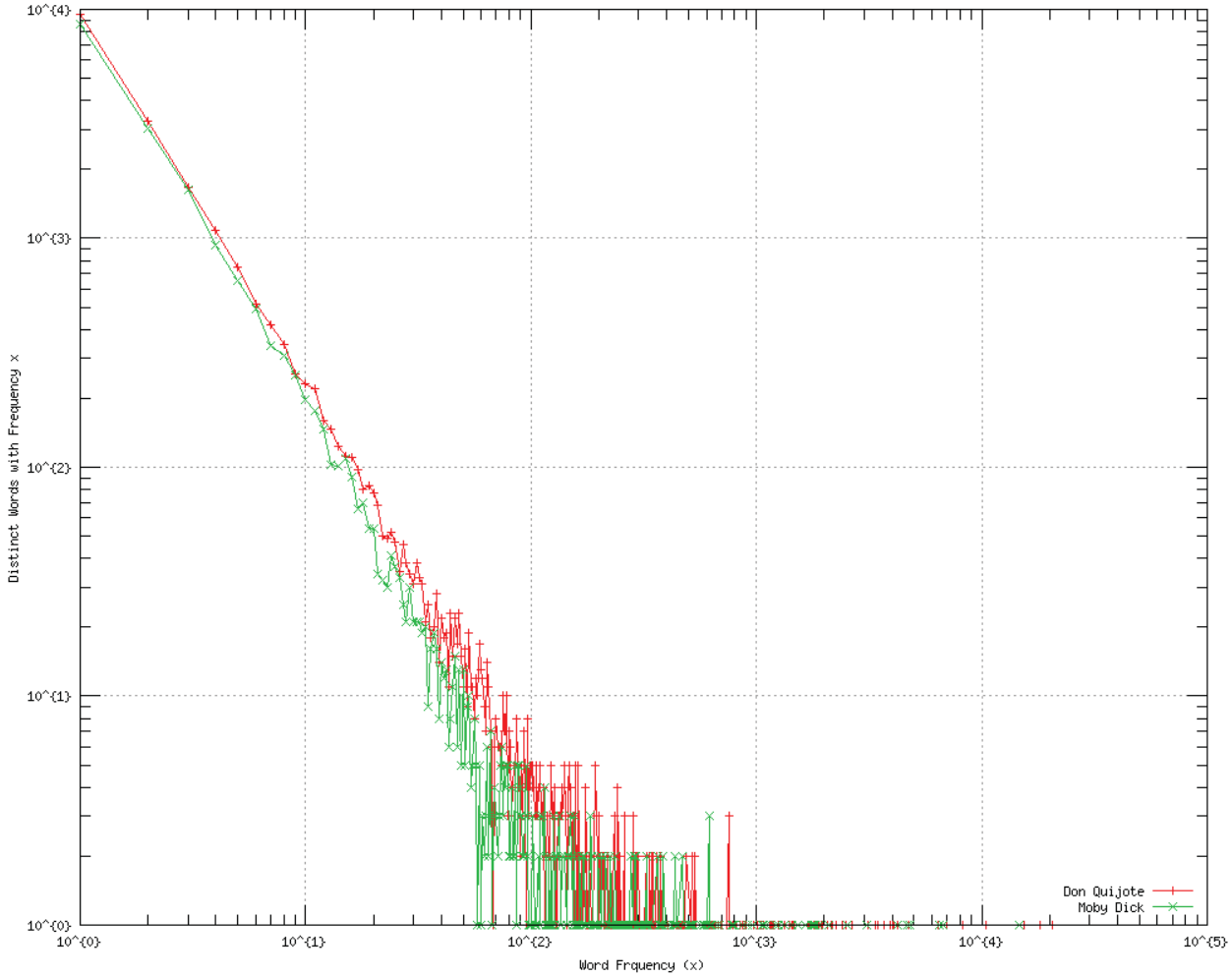
The above result was obtained using C++. See Homework3.cpp, and in particular the function Q2().

2(f)

Maximum likelihood estimation (e) gives the best estimate.

Least squares linear regression (c) gives the worst estimate.

3(a)



The above plot was generated using C++. See Homework3.cpp (function Q3()), WordDistribution.h, and WordDistribution.cpp.

3(b)

Don Quijote:

α = 1.85

x_{min} = 196

The above results were obtained by using C++ to write the Word Frequencies to a text file (see Homework3.cpp (function Q3()) and WordDistribution.cpp (GetDistinctWordFrequency())). The text file was then used by Matlab to produce the results by using the following Matlab function:

EDU>> [alpha, xMin, l] = plfit(dlmread('DonQuijoteDistinctWordFrequency.txt'))

Moby Dick:

α = 2.01

x_{min} = 212

The above results were obtained by using C++ to write the Word Frequencies to a text file (see Homework3.cpp (function Q3()) and WordDistribution.cpp (GetDistinctWordFrequency())). The text file was then used by Matlab to produce the results by using the following Matlab function:

EDU>> [alpha, xMin, l] = plfit(dlmread('MobyDickDistinctWordFrequency.txt'))

The above results appear to indicate that the word frequency distribution of human language roughly follows a power law distribution having $\propto \approx [1.85, 2.01]$ for $x \geq \approx [196, 212]$ For almost all $\propto$ values in this approximated range ($where \propto \leq 2$), $E(x) = \infty$. For all $\propto$ values in and near this range ($where \propto \leq 3$), $Var(x) = \infty$.

What this means is that there is a large number of words that occur in human language with a high frequency. There are also a significant number of words (in a heavy tail) that are somewhat rare but occur often enough to (1) make the expected value of the PDF extremely high or infinite and (2) make the variance infinite.

3(c)

Where m is the # of letters, the # of distinct words of length y is $\boldsymbol{m^y}$

Let W = the number of distinct words in the monkey's alphabet.

Let $p(y) =$ the probability of choosing a word of length y when we choose a word uniformly at random from among all the distinct words in the monkey's alphabet.

It follows that $p(y) = \frac{m^y}{W}$

Given the simple identity: $m = e^{\ln m}$, we can rewrite $p(y)$ as follows.

$$p(y) = \frac{m^y}{W} = \frac{e^{\ln(m)^y}}{W} = \frac{e^{\ln(m)\, y}}{W}$$

Therefore, $\boldsymbol{a = \ln m}$

3(d)

$$\boldsymbol{x = \left(\frac{1 - q_s}{m}\right)^y q_s}$$

Given the simple identity: $m = e^{\ln m}$, we can rewrite x as follows

$$x = q_s\, e^{\ln\left(\frac{1-q_s}{m}\right)^y}$$

$$x = q_s\, e^{y \ln\left(\frac{1-q_s}{m}\right)}$$

x is proportional to e^{by}, thus:

$$\boldsymbol{b = ln\left(\frac{1 - q_s}{m}\right) = ln(1 - q_s) - ln(m)}$$

3(e)

We are given that:

$|p(y)dy| = |p(x)dx|$

Therefore:

$p(x) = \pm p(y)\frac{dy}{dx}$ (3.e.1)

From 3(c), we know:

$p(y) = e^{\ln(m)\, y}$ (3.e.2)

From 3(d), we know:

$$x = e^{y \ln\left(\frac{1-q_s}{m}\right)}$$

Therefore:

$$\ln(x) = \ln\left(e^{y \ln\left(\frac{1-q_s}{m}\right)}\right)$$

$$\ln(x) = y \ln\left(\frac{1-q_s}{m}\right)$$

$$y = \frac{\ln(x)}{\ln\left(\frac{1-q_s}{m}\right)} \quad \text{(3.e.3)}$$

$$\frac{dy}{dx} = \frac{1}{x \ln\left(\frac{1-q_s}{m}\right)} \quad \text{(3.e.4)}$$

From (3.e.1) we know:

$$p(x) \propto p(y)\frac{dy}{dx}$$

Substituting the value of p(y) from (3.e.2), we obtain:

$$p(x) \propto \left(e^{\ln(m)\, y}\right)\frac{dy}{dx}$$

Substituting the value of y from (3.e.3), we obtain:

$$p(x) \propto \left(e^{\ln(m)\left(\frac{\ln(x)}{\ln\left(\frac{1-q_s}{m}\right)}\right)} \right)\frac{dy}{dx}$$

Substituting the value of $\frac{dy}{dx}$ from (3.e.4), we obtain:

$$p(x) \propto \left(e^{\ln(m)\left(\frac{\ln(x)}{\ln\left(\frac{1-q_s}{m}\right)}\right)} \right)\left(\frac{1}{x\ln\left(\frac{1-q_s}{m}\right)}\right) = \frac{x^{\frac{\ln(e)\ln(m)}{ln\left(\frac{1-q_s}{m}\right)}-1}}{ln\left(\frac{1-q_s}{m}\right)}$$

Thus:

$$p(x) \propto x^{\frac{\ln(m)}{ln\left(\frac{1-q_s}{m}\right)}-1} = x^{\left(\frac{-\ln(m)}{\ln(m)-\ln(1-q_s)}-1\right)} \qquad \text{(3.e.5)}$$

Because of (3.e.5), we know:

$$-\alpha = -\left(\frac{-\ln(m)}{\ln(m)-\ln(1-q_s)}-1\right)$$

Therefore:

$$\boldsymbol{\alpha = \frac{\ln(m)}{\ln(m)-\ln(1-q_s)}+1} \qquad \text{(3.e.6)}$$

Constraints on q_s and m are the following:

$0 \leq q_s \leq 1$

$m \in$ [redacted]

$\ln(1-q_s)$ approaches 0 as q_s approaches 0. Therefore, and $\frac{\ln(m)}{\ln(m)}$ is 1 for all values of m.

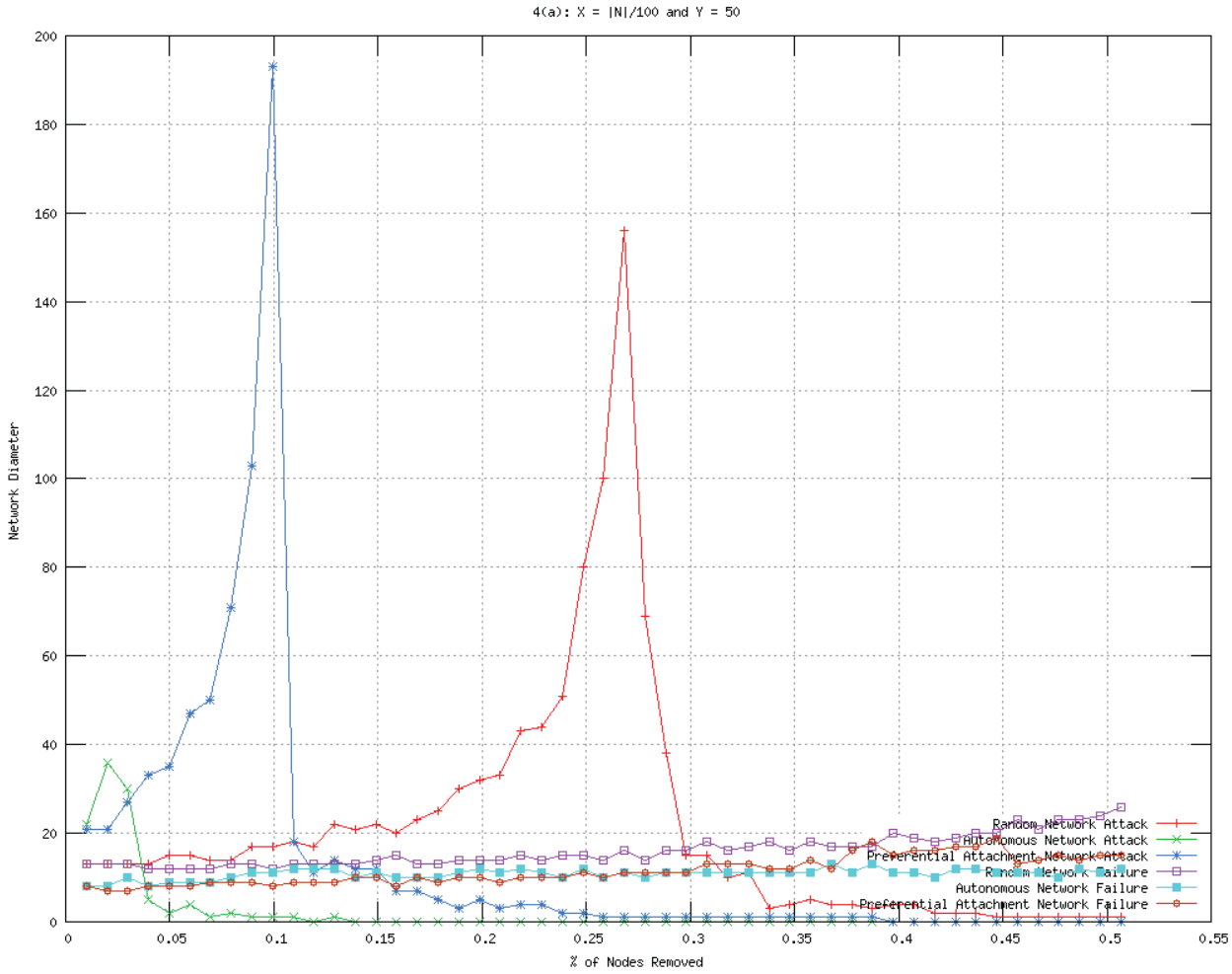
Thus, α converges to 2 for small q_s and large m.

Also, $1 \leq \alpha \leq 2 \; for \; all \; q_s \; where \; 0 \leq q_s \leq 1.$

The value of α converging to 2 resembles our results in 3(b) because both of our estimates of α values are near 2.

4(a)

The plot of the diameter of each of the networks against the percentage of nodes removed from each of the networks for Scenario 1 (X = |N|/100 and Y = 50), is shown below.



All three networks are highly resilient to random failures. As you can see, for each of the random failure scenarios the network diameter remains relatively stagnant.

All three networks are susceptible to targeted attacks, but some are more susceptible than others. In each of the targeted attack scenarios, the network diameter peaks and then drops off. The reason for the drop off after the peaks is that as we remove high-degree nodes we get more small disconnected components that contain nodes with small average path lengths between them.

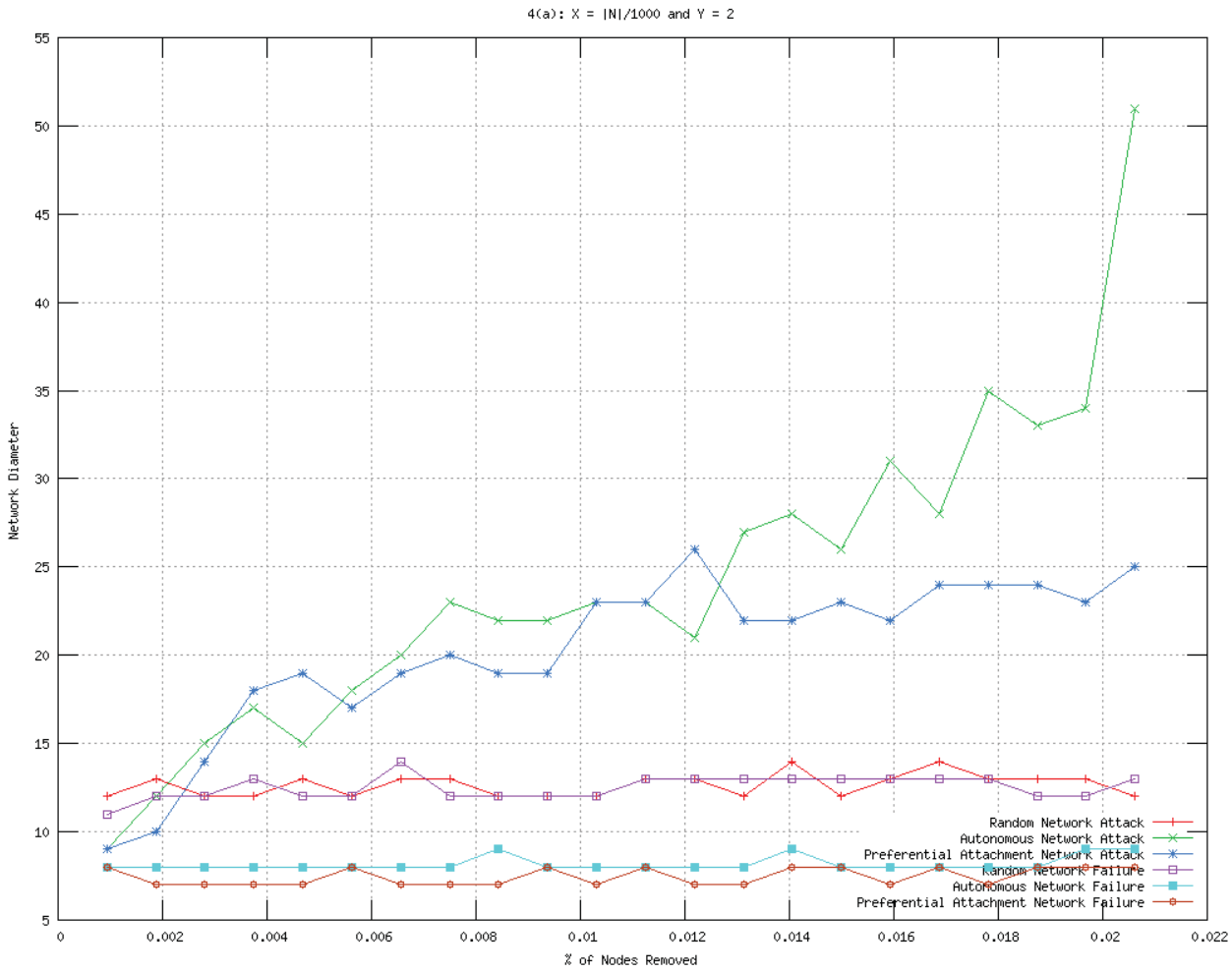
The Real World Autonomous System Network is the most susceptible to targeted attacks. As you can see, only approximately 2% of the high-degree nodes in this network need to be targeted before the network starts to break up into smaller disconnected components.

The Preferential Attachment Network is less susceptible to targeted attacks than the Real World Autonomous System Network but more susceptible to attacks than the Random Network. As you can

see, when approximately 10% of the high-degree nodes in the Preferential Attachment Network are removed the network begins to break up into smaller disconnected components.

Not surprisingly, the Random Network is the most resilient to targeted attacks. As you can see, approximately 27% of the high-degree nodes in this network need to be targeted before the network starts to break up into smaller disconnected components.

The plot of the diameter of each of the networks against the percentage of nodes removed from each of the networks for Scenario 2 (X = |N|/1000 and Y = 2), is shown below.



Scenario 2 (X = |N|/1000 and Y = 2) examines the resilience of the three networks when up to 2% of their nodes are removed through targeted and random attacks. Once again, we find that all three networks are highly resilient to random failures. As you can see, in each of the random failure scenarios the network diameter remains relatively stagnant.

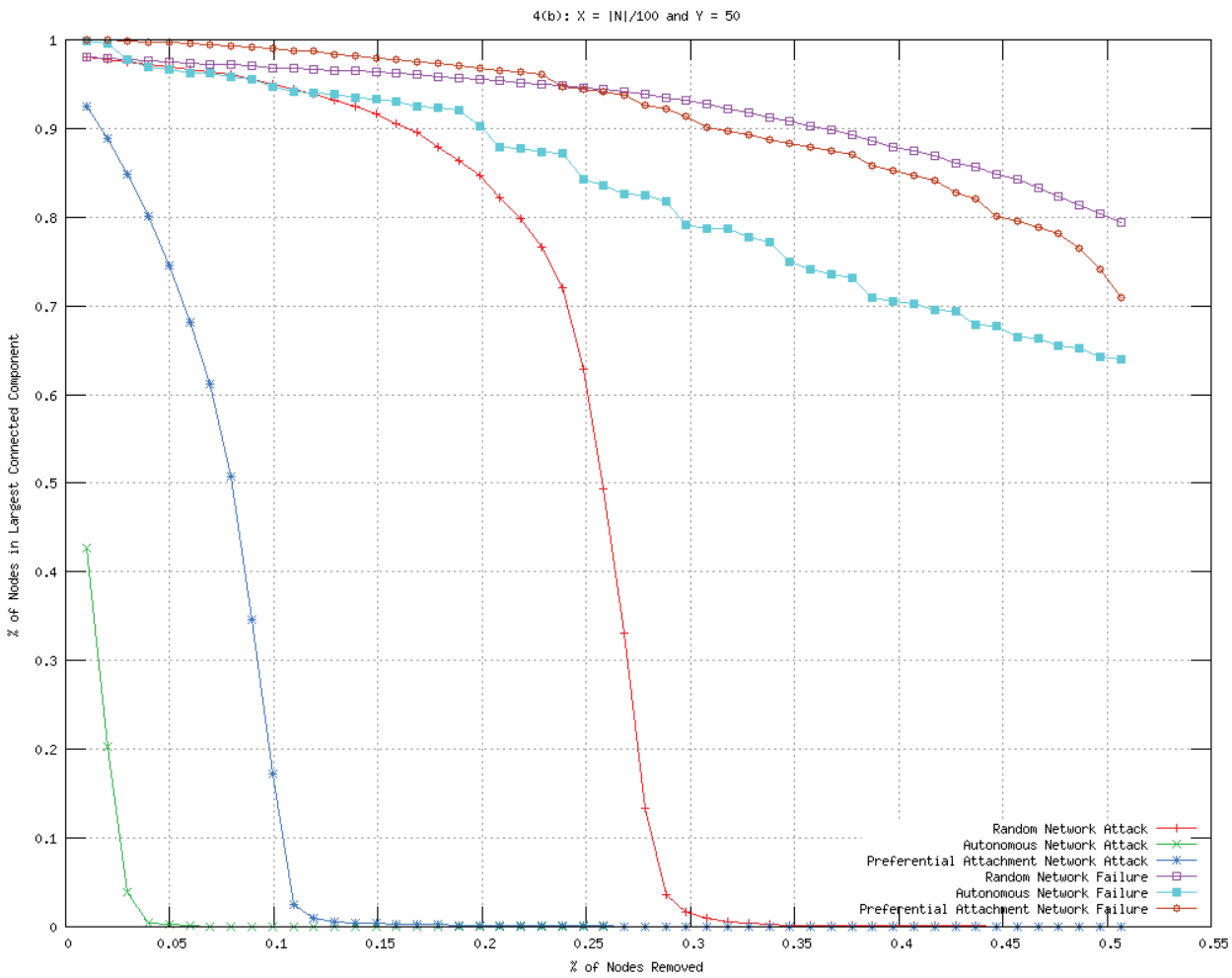
Looking at the failure rate under targeted attack of the Autonomous System Network and the Preferential Attachment Network, we can see that these networks exhibit similar rates of network diameter growth until approximately 12% to 13% of the high-degree nodes have been removed. After 12% to 13% of the high-degree nodes have been removed, the network diameter of the Autonomous System Network quickly spikes.

Under targeted attack, the Random Network's network diameter is very constant within the confines of Scenario 2. This is not surprising because the Random Network is much less likely to have nodes with very high degree than the other two networks.

The above results were obtained using C++. See Homework3.cpp, procedure Q4(). See also NetworkRobustnessModel.h, NetworkRobustnessModel.cpp, PreferentialAttachmentModel.h, and PreferentialAttachmentModel.cpp.

4(b)

A plot of the fraction of nodes making up the largest connected component against the percentage of nodes removed from each of the networks where X = |N|/100 and Y = 50 is shown below.



As you can see, the Real World Autonomous System Network is the least robust (i.e., the most susceptible to targeted attacks) as a very small fraction of high-degree nodes need to be targeted before this network's largest connected component size drops drastically. The Autonomous System Network is less robust than the Real World Autonomous System Network and more robust than the Random Network. And the Random Network is the most robust as a considerable percentage of its high-degree nodes need to be targeted before this network's largest connected component size drops drastically.

We can again see here that all three networks are highly robust under random attacks.

The data in this graph lends confirmation our analysis in 4(a) above. We can see clearly that the % of nodes removed where the drop in the number of nodes in the largest connected component occurs in the 4(b) plot in each of the attack scenarios roughly correspond to the % of nodes removed where the drop in network diameter occurs in the corresponding (Scenario 1) 4(a) plot in each of the attack scenarios. In other words, we state in our analysis of Scenario 1 in 4(a) that "[t]he reason for the drop off after the peaks is that as we remove high-degree nodes we get more small disconnected components that contain nodes with small average path lengths between them." The plot here in 4(b) demonstrates this directly.

The above results were obtained using C++. See Homework3.cpp, procedure Q4(). See also NetworkRobustnessModel.h, NetworkRobustnessModel.cpp, PreferentialAttachmentModel.h, and PreferentialAttachmentModel.cpp.

```
// Homework3.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "PreferentialAttachmentModel.h"
#include "WordDistribution.h"

#include <iostream>
#include <Snap.h>

double GenPowerLawPdf(double alpha, double xMin)
{
	return xMin * pow(1 - (double)rand()/RAND_MAX, -1.0/(alpha-1.0));
}

double GenPowerLawCcdf(double x, double alpha, double xMin)
{
	return pow(x / xMin, 1 - alpha);
}

double MaximumLiklihood(TVec<TFlt> d, double xMin)
{
	double sum = 0.0;

	for (TVec<TFlt>::TIter di = d.BegI(); di < d.EndI(); di++)
		sum += log(*di/xMin);

	return 1.0 + (double)d.Len() * pow(sum, -1.0);
}

void Q2()
{
	////////////////////////
	double multiplier = 2.0;
	double C = 1.0/3.0;
	int numSamples = 100000;
	////////////////////////

	srand((unsigned int)time(NULL));

	TVec<TFlt> pdf;
	TVec<TFltPr> normalizedPdfHistogram, ccdf;
	double alpha = 2.0;
	double xMin = 1.0;

	// create PDF
	for (int i = 0; i < numSamples; i++)
		pdf.Add(GenPowerLawPdf(alpha, xMin));


	// sort PDF
	pdf.QSort(0, pdf.Len()-1, true);


	// create CCDF
	int n = 0;
	for (TVec<TFlt>::TIter i = pdf.BegI(); i < pdf.EndI(); i++)
		ccdf.Add(TFltPr(*i, (TFlt)(numSamples-(++n))/(TFlt)numSamples));


	// create Histogram of PDF
	int iterationNum = 1;
	double x1 = xMin;
	double x2 = xMin + multiplier * C; //*iterationNum

	TVec<TFlt>::TIter iter = pdf.BegI();

	while (iter != pdf.EndI())
	{
```

```
                if (*iter <= x2)
                {
                        int numValuesInRange = 0;

                        // find the number of values between x1 and x2
                        do
                        {
                                ++numValuesInRange;
                                ++iter;
                        } while (*iter <= x2 && iter != pdf.EndI());

                        normalizedPdfHistogram.Add(TFltPr((x1 + x2)/2.0,
(TFlt)numValuesInRange/(TFlt)numSamples));
                }

                x1 = x2;
                x2 += pow(multiplier, ++iterationNum) * C;
        }

        //plot Histogram of PDF
        TGnuPlot gnuPlot1("Q2b");
        gnuPlot1.AddPlot(normalizedPdfHistogram, gpwBoxes, "Normalized PDF Histogram");
        gnuPlot1.SetXYLabel("x", "P(X = x)");
        gnuPlot1.SetScale(gpsLog2XY);
        gnuPlot1.SavePng(); // ERROR WHEN C IS LESS THAN 0.3

        //plot CCDF
        TGnuPlot gnuPlot2("Q2_CCDF");
        gnuPlot2.AddPlot(ccdf, gpwLinesPoints, "CCDF");
        gnuPlot2.SetXYLabel("x", "P(X > x)");
        gnuPlot2.SetScale(gpsLog2XY);
        gnuPlot2.SavePng();


        TFOut pdfHistogramFile("Q2_Normalized_Pdf_Histogram.xml");
        normalizedPdfHistogram.SaveXml(pdfHistogramFile, "Q2_Normalized_Pdf_Histogram");

        TFOut ccdfFile("Q2_Ccdf.xml");
        ccdf.SaveXml(ccdfFile, "Q2_Ccdf");


        std::cout << "Maximum Liklihood Alpha Estimation = " << MaximumLiklihood(pdf, xMin) << std::endl;
}

void WriteToFile(TVec<TInt> vec, TStr filename)
{
        TFOut fout(filename);

        for (TVec<TInt>::TIter i = vec.BegI(); i < vec.EndI(); i++)
        {
                fout.PutInt(*i);
                fout.PutLn();
        }

        fout.Flush();
}

void Q3()
{
        WordDistribution distDonQuijote("donquijote.txt");
        WordDistribution distMobyDick("mobydick.txt");

        TGnuPlot gnuPlot("Q3a");

        gnuPlot.AddPlot(distDonQuijote.GetWordFrequency(), gpwLinesPoints, "Don Quijote");
        gnuPlot.AddPlot(distMobyDick.GetWordFrequency(), gpwLinesPoints, "Moby Dick");

        gnuPlot.SetScale(gpsLog10XY);
        gnuPlot.SetXYLabel("Word Frquency", "Distinct Words");
```

```
	gnuPlot.SavePng();

	WriteToFile(distDonQuijote.GetDistinctWordFrequency(), "DonQuijoteDistinctWordFrequency.txt");
	WriteToFile(distMobyDick.GetDistinctWordFrequency(), "MobyDickDistinctWordFrequency.txt");
}

TVec<TFltPr> Q4a_AttackVsDiameter(NetworkRobustnessModel & network, TInt x, TFlt y)
{
	// attack in batches of x
	// until y (a percentage) of nodes have been deleted

	double initNodes = (double)network.GetNodes();
	TFlt percentRemoved;
	TVec<TFltPr> plot;

	do
	{
		network.Attack(x);
		percentRemoved = 1.0 - ((double)network.GetNodes() / initNodes);
		plot.Add( TFltPr(percentRemoved, (TFlt)network.GetDiameter(20)) );
	} while (percentRemoved < y);

	return plot;
}

TVec<TFltPr> Q4a_FailureVsDiameter(NetworkRobustnessModel & network, TInt x, TFlt y)
{
	// failures occur in batches of x
	// until y% of nodes have been deleted

	double initNodes = (double)network.GetNodes();
	TFlt percentRemoved;
	TVec<TFltPr> plot;

	do
	{
		network.Failure(x);
		percentRemoved = 1.0 - ((double)network.GetNodes() / initNodes);
		plot.Add( TFltPr(percentRemoved, (TFlt)network.GetDiameter(20)) );
	} while (percentRemoved < y);

	return plot;
}

void Q4aScenario(TInt xDenominator, TFlt y, TStr filename, TStr title)
{
	NetworkRobustnessModel randomNetwork1(TSnap::GenRndGnm<PUNGraph>(10670, 22002, false));
	NetworkRobustnessModel randomNetwork2(TSnap::GenRndGnm<PUNGraph>(10670, 22002, false));
	NetworkRobustnessModel autonomousNetwork1(TSnap::LoadEdgeList<PUNGraph>("oregon1_010331.txt", 0,
1));
	NetworkRobustnessModel autonomousNetwork2(TSnap::LoadEdgeList<PUNGraph>("oregon1_010331.txt", 0,
1));
	PreferentialAttachmentModel prefAttachNetwork1(TSnap::GenFull<PUNGraph>(40), 10670-40, 2);
	PreferentialAttachmentModel prefAttachNetwork2(TSnap::GenFull<PUNGraph>(40), 10670-40, 2);

	TGnuPlot gnuPlot(filename);

	gnuPlot.AddPlot(Q4a_AttackVsDiameter(randomNetwork1, randomNetwork1.GetNodes() / xDenominator, y),
gpwLinesPoints, "Random Network Attack");
	gnuPlot.AddPlot(Q4a_AttackVsDiameter(autonomousNetwork1, autonomousNetwork1.GetNodes() /
xDenominator, y), gpwLinesPoints, "Autonomous Network Attack");
	gnuPlot.AddPlot(Q4a_AttackVsDiameter(prefAttachNetwork1, prefAttachNetwork1.GetNodes() /
xDenominator, y), gpwLinesPoints, "Preferential Attachment Network Attack");

	gnuPlot.AddPlot(Q4a_FailureVsDiameter(randomNetwork2, randomNetwork2.GetNodes() / xDenominator,
y), gpwLinesPoints, "Random Network Failure");
	gnuPlot.AddPlot(Q4a_FailureVsDiameter(autonomousNetwork2, autonomousNetwork2.GetNodes() /
xDenominator, y), gpwLinesPoints, "Autonomous Network Failure");
```

```
        gnuPlot.AddPlot(Q4a_FailureVsDiameter(prefAttachNetwork2, prefAttachNetwork2.GetNodes() /
xDenominator, y), gpwLinesPoints, "Preferential Attachment Network Failure");

        gnuPlot.SetTitle(title);
        gnuPlot.SetXYLabel("% of Nodes Removed", "Network Diameter");
        gnuPlot.SavePng();
}

void Q4a()
{
        Q4aScenario(100, 0.50, "Q4a1", "4(a): X = |N|/100 and Y = 50");
        Q4aScenario(1000, 0.02, "Q4a2", "4(a): X = |N|/1000 and Y = 2");
}

TVec<TFltPr> Q4b_AttackVsMxScc(NetworkRobustnessModel & network, TInt x, TFlt y)
{
        // attack in batches of x
        // until y% of nodes have been deleted

        double initNodes = (double)network.GetNodes();
        TFlt percentRemoved;
        TVec<TFltPr> plot;

        do
        {
                network.Attack(x);
                percentRemoved = 1.0 - ((double)network.GetNodes() / initNodes);
                plot.Add( TFltPr(percentRemoved, (TFlt)network.GetNodesInMxScc()/(TFlt)network.GetNodes())
);
        } while (percentRemoved < y);

        return plot;
}

TVec<TFltPr> Q4b_FailureVsMxScc(NetworkRobustnessModel & network, TInt x, TFlt y)
{
        // failures occur in batches of x
        // until y% of nodes have been deleted

        double initNodes = (double)network.GetNodes();
        TFlt percentRemoved;
        TVec<TFltPr> plot;

        do
        {
                network.Failure(x);
                percentRemoved = 1.0 - ((double)network.GetNodes() / initNodes);
                plot.Add( TFltPr(percentRemoved, (TFlt)network.GetNodesInMxScc()/(TFlt)network.GetNodes())
);
        } while (percentRemoved < y);

        return plot;
}

void Q4b()
{
        NetworkRobustnessModel randomNetwork1(TSnap::GenRndGnm<PUNGraph>(10670, 22002, false));
        NetworkRobustnessModel randomNetwork2(TSnap::GenRndGnm<PUNGraph>(10670, 22002, false));
        NetworkRobustnessModel autonomousNetwork1(TSnap::LoadEdgeList<PUNGraph>("oregon1_010331.txt", 0,
1));
        NetworkRobustnessModel autonomousNetwork2(TSnap::LoadEdgeList<PUNGraph>("oregon1_010331.txt", 0,
1));
        PreferentialAttachmentModel prefAttachNetwork1(TSnap::GenFull<PUNGraph>(40), 10670-40, 2);
        PreferentialAttachmentModel prefAttachNetwork2(TSnap::GenFull<PUNGraph>(40), 10670-40, 2);

        TGnuPlot gnuPlot("Q4b");

        gnuPlot.AddPlot(Q4b_AttackVsMxScc(randomNetwork1, randomNetwork1.GetNodes() / 100, 0.50),
gpwLinesPoints, "Random Network Attack");
```

```
        gnuPlot.AddPlot(Q4b_AttackVsMxScc(autonomousNetwork1, autonomousNetwork1.GetNodes() / 100, 0.50),
gpwLinesPoints, "Autonomous Network Attack");
        gnuPlot.AddPlot(Q4b_AttackVsMxScc(prefAttachNetwork1, prefAttachNetwork1.GetNodes() / 100, 0.50),
gpwLinesPoints, "Preferential Attachment Network Attack");

        gnuPlot.AddPlot(Q4b_FailureVsMxScc(randomNetwork2, randomNetwork2.GetNodes() / 100, 0.50),
gpwLinesPoints, "Random Network Failure");
        gnuPlot.AddPlot(Q4b_FailureVsMxScc(autonomousNetwork2, autonomousNetwork2.GetNodes() / 100, 0.50),
gpwLinesPoints, "Autonomous Network Failure");
        gnuPlot.AddPlot(Q4b_FailureVsMxScc(prefAttachNetwork2, prefAttachNetwork2.GetNodes() / 100, 0.50),
gpwLinesPoints, "Preferential Attachment Network Failure");

        gnuPlot.SetTitle("4(b): X = |N|/100 and Y = 50");
        gnuPlot.SetXYLabel("% of Nodes Removed", "% of Nodes in Largest Connected Component");
        gnuPlot.SavePng();
}

void Q4()
{
        Q4a();
        Q4b();
}

void PrintSystemTime()
{
        SYSTEMTIME * st = new SYSTEMTIME;
        GetSystemTime(st);
        std::cout << st->wHour-5 << ":" << st->wMinute << ":" << st->wSecond << "." << st->wMilliseconds
<< std::endl;
        delete st;
}

int _tmain(int argc, _TCHAR* argv[])
{
        PrintSystemTime();

        try
        {
                Q2();
                Q3();
                Q4();
        }
        catch(std::exception& e)
    {
                PrintSystemTime();
        std::cerr << "ERROR: " << e.what() << "\n";
                system("pause");
        return 1;
    }

        PrintSystemTime();
        system("pause");
        return 0;
}
```

```
// WordDistribution.h

#include "stdafx.h"
#include <Snap.h>

class WordDistribution
{
private:
        THash<TStr, TInt> wordFrequencies;
        int totalWords;

public:
        WordDistribution(TStr filename);
        TVec<TIntPr> GetWordFrequency();
        TVec<TInt> GetDistinctWordFrequency();
};
```

```
// WordDistribution.cpp

#include "stdafx.h"
#include "WordDistribution.h"

#include <iostream>

WordDistribution::WordDistribution(TStr filename)
{
	TFIn fin(filename);
	TStr word;
	totalWords = 0;

	while (fin.GetNextLn(word))
	{
		++totalWords;

		if (wordFrequencies.IsKey(word))
			wordFrequencies.GetDat(word)++;
		else
			wordFrequencies.AddDat(word, 1);
	}

	wordFrequencies.SortByDat(false);
}

TVec<TIntPr> WordDistribution::GetWordFrequency()
{
	THash<TInt, TInt> wordFrequencyDistribution;

	for (THash<TStr, TInt>::TIter i = wordFrequencies.BegI(); i < wordFrequencies.EndI(); i++)
	{
		int frequency = i.GetDat();

		if (wordFrequencyDistribution.IsKey(frequency))
			wordFrequencyDistribution.GetDat(frequency)++;
		else
			wordFrequencyDistribution.AddDat(frequency, 1);

	}

	TVec<TIntPr> wordFrequency;

	for (THash<TInt, TInt>::TIter i = wordFrequencyDistribution.BegI(); i <
wordFrequencyDistribution.EndI(); i++)
		wordFrequency.Add(TIntPr(i.GetKey(), i.GetDat()));

	wordFrequency.QSort(0, wordFrequency.Len()-1, true);

	return wordFrequency;
}

TVec<TInt> WordDistribution::GetDistinctWordFrequency()
{
	THash<TInt, TInt> wordFrequencyDistribution;

	for (THash<TStr, TInt>::TIter i = wordFrequencies.BegI(); i < wordFrequencies.EndI(); i++)
	{
		int frequency = i.GetDat();

		if (wordFrequencyDistribution.IsKey(frequency))
			wordFrequencyDistribution.GetDat(frequency)++;
		else
			wordFrequencyDistribution.AddDat(frequency, 1);
	}

	TVec<TInt> wordFrequency;
```

```
        for (THash<TInt, TInt>::TIter i = wordFrequencyDistribution.BegI(); i <
wordFrequencyDistribution.EndI(); i++)
                wordFrequency.AddUnique(i.GetKey());

        wordFrequency.QSort(0, wordFrequency.Len()-1, true);

        return wordFrequency;
}
```

```
// NetworkRobustnessModel.h

#include <Snap.h>

class NetworkRobustnessModel
{
protected:
        PUNGraph graph;

        // stores the ID of each node in the graph
        TVec<TInt> allNodes;

public:
        NetworkRobustnessModel(PUNGraph graph);

        int GetDiameter(int sampleSize);
        int GetNodes();
        int GetNodesInMxScc();

        void Failure(int numNodes);
        void Attack(int numNodes);
};
```

```
// NetworkRobustnessModel.h

#include "stdafx.h"
#include "NetworkRobustnessModel.h"

NetworkRobustnessModel::NetworkRobustnessModel(PUNGraph graph)
{
	this->graph = graph;
	srand((unsigned int)time(NULL));

	for (TUNGraph::TNodeI nodeI = graph->BegNI(); nodeI < graph->EndNI(); nodeI++)
		allNodes.Add(nodeI.GetId());

	allNodes.QSort(0, allNodes.Len()-1, true);
}

void NetworkRobustnessModel::Failure(int numNodes)
{
	for (int i = 0; i < numNodes; i++)
	{
		int rndIndex = rand() % allNodes.Len();
		graph->DelNode(allNodes[rndIndex]);
		allNodes.Del(rndIndex);
	}
}

void NetworkRobustnessModel::Attack(int numNodes)
{
	for (int i = 0; i < numNodes; i++)
	{
		int nodeId = TSnap::GetMxDegNId(graph);
		graph->DelNode(nodeId);
		allNodes.DelIfIn(nodeId);
	}
}

int NetworkRobustnessModel::GetDiameter(int sampleSize)
{
	return TSnap::GetBfsFullDiam(graph, sampleSize, false);
}

int NetworkRobustnessModel::GetNodes()
{
	return graph->GetNodes();
}

int NetworkRobustnessModel::GetNodesInMxScc()
{
	return TSnap::GetMxScc(graph)->GetNodes();
}
```

```
// PreferentialAttachmentModel.h

#include "stdafx.h"
#include "NetworkRobustnessModel.h"
#include <Snap.h>

class PreferentialAttachmentModel : public NetworkRobustnessModel
{
private:
        TInt GetRandomDestinationNodeId(TVec<TIntPr> & edges);
        void AddEdge(TVec<TIntPr> & edges, int sourceNodeId, int destinationNodeId);

public:
        PreferentialAttachmentModel(PUNGraph graph, int newNodes, int edgesPerNewNode);
};
```

```
// PreferentialAttachmentModel.cpp

#include "stdafx.h"
#include "PreferentialAttachmentModel.h"

PreferentialAttachmentModel::PreferentialAttachmentModel(PUNGraph graph, int newNodes, int
edgesPerNewNode) : NetworkRobustnessModel(graph)
{
	// a list of all the edges
	// we need this to implement the preferential attachment procedure
	TVec<TIntPr> edges;

	// add all the edges to the edges TVec so that we can select them randomly later
	for (TUNGraph::TEdgeI edgeI = graph->BegEI(); edgeI < graph->EndEI(); edgeI++)
		edges.Add(TIntPr(min(edgeI.GetSrcNId(), edgeI.GetDstNId()), max(edgeI.GetSrcNId(),
edgeI.GetDstNId())));

	edges.QSort(0, edges.Len()-1, true);

	// add new nodes using the preferential attachment method
	for (int i = 0; i < newNodes; i++)
	{
		int sourceNodeId = graph->AddNode();

		allNodes.Add(sourceNodeId);

		for (int j = 0; j < edgesPerNewNode; j++)
			AddEdge(edges, sourceNodeId, GetRandomDestinationNodeId(edges));
	}

	allNodes.QSort(0, allNodes.Len()-1, true);
}

inline void PreferentialAttachmentModel::AddEdge(TVec<TIntPr> & edges, int sourceNodeId, int
destinationNodeId)
{
	if (edges.AddUnique(TIntPr(min(sourceNodeId, destinationNodeId), max(sourceNodeId,
destinationNodeId))) != -1)
		graph->AddEdge(sourceNodeId, destinationNodeId);
}

inline TInt PreferentialAttachmentModel::GetRandomDestinationNodeId(TVec<TIntPr> & edges)
{
	return (rand() % 2) ? edges[rand() % edges.Len()].Val1() : edges[rand() % edges.Len()].Val2();
}
```