5-21-2008

A Statistical Look at Maps of the Discrete Logarithm

Nathan Lindle
Rose-Hulman Institute of Technology

Advisors:
Joshua Holden

MSTR 08-06

A Statistical Look at Maps of the Discrete Logarithm

Nathan Lindle

Adviser: Joshua Holden

Mathematical Sciences Technical Report Series
MSTR 08-06

May 21, 2008

Department of Mathematics
Rose-Hulman Institute of Technology
http://www.rose-hulman.edu/math

Fax (812)-877-8333 **Phone (812)-877-8193**

A Statistical Look at Maps of the Discrete Logarithm

Nathan Lindle
Rose-Hulman Institute of Technology
Terre Haute, IN 47803
lindlenw@rose-hulman.edu

May 21, 2008

1 Introduction

Cryptography is being used today more than it ever has in the past. Millions of transactions are being conducted every hour using encrypted channels, most of which use the Internet as their medium. It is taken for granted by the average user that these transaction are secure, but mathematicians and computer scientists alike are constantly testing the algorithms being used. Several of these cryptosystems use the transformation

$$g^x \equiv y \pmod{n} \tag{1}$$

The appeal of this transformation is that it is quite simple to calculate $g^x \bmod n$; exponentiation by squaring is fairly simple and quick even using very large numbers. However there is no known algorithm to compute the inverse of the transformation (that is given y, g, and n, find x) in a comparable amount of time. This is called the discrete log problem, and Diffie-Hellman key exchange, RSA and the Blum-Micali pseudorandom bit generator all rely on its inherent difficulty. This paper explores the maps generated by this transformation with the hope of gaining some insight into its structure and similarity to random mappings.

2 Functional Graphs

2.1 Terminology

A directed functional graph is a set of nodes S and a set of directed edges where (a, b) denotes an edge from node a to node b on the graph. Also, every node must have exactly one edge coming out of it (this is where the "functional" part of the name comes from). Using the transformation from (1), define a transition function φ where $\varphi(x) = g^x \pmod{p}$. Given any prime modulus p and some base g, construct a functional graph using φ where $S = \{1, 2, ..., p-1\}$ and the directed edges are $(a, \varphi(a))$ for every $a \in S$.

Now take any random starting node u_0, construct the sequence $u_1 = \varphi(u_0)$, $u_2 = \varphi(u_1)$, ..., and note that there are $p-1$ nodes in the graph. After at most p iterations, it must be the case that $u_i = u_j$ where $i < j$. j is called the rho length of u_0 (because when you graph the sequence it looks like a ρ), i

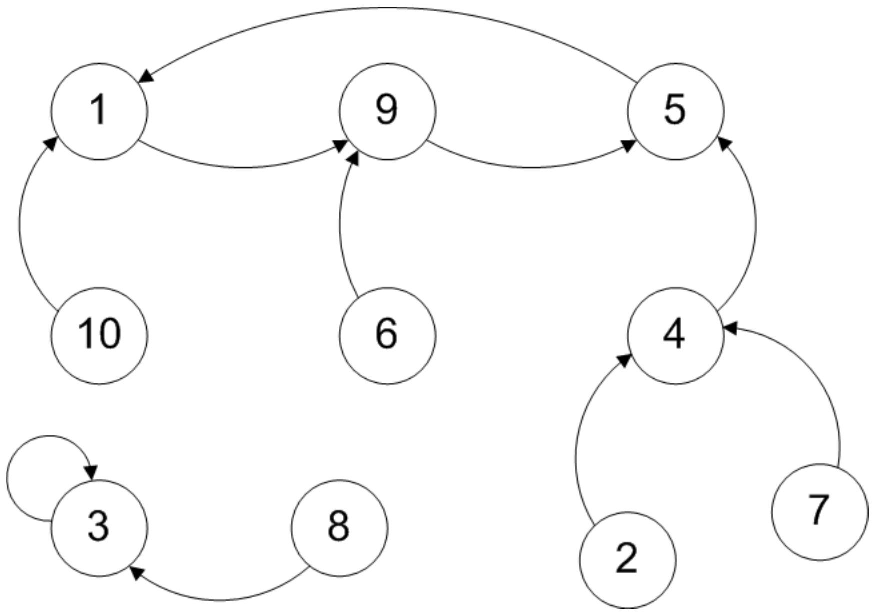

Figure 1: A graph constructed using $\varphi(x) \to 9^x \bmod 11$

is the tail length of u_0 and $j - i$ is the cycle length of u_0. Note that it is possible for u_0 to be part of a cycle, so the tail length could be 0. For example, nodes 3 and 8 in Figure 1 have a cycle length of 1, and the rest of the nodes have a cycle length of 3. Nodes 2 and 7 both have a tail length of 2, and nodes 10, 6, 4 and 8 have tail length 1, and the rest have a length of 0. The average tail length of a graph is the sum of the tail lengths of each node divided by the number of nodes. Similarly, the average cycle length is the sum of the cycle lengths of each node divided by the number of nodes. Figure 1 has an average tail length of 0.8 and an average cycle length of 2.6, for instance.

A node x is an image node if there is a transition (a, x) for some $a \in S$. Otherwise x is called a terminal node. A cycle and all the nodes included in tails which lead to that cycle is called a connected component. Since the out-degree of every node is exactly one, no component is connected to any other, and each component has exactly 1 cycle.

The maximum cycle length for a graph is the size of the largest cycle (which may or may not be a part of the largest component). The maximum tail length is the maximum number of transitions any node is from its cycle, or equivalently the maximum tail length of any node.

The in-degree of a node b is the number of edges of the form (a, b) for some node a in the graph. A functional graph is said to be m-ary if the in-degree of each node is either 0 or m. 1-ary graphs are referred to as permutations, and 2-ary graphs are referred to as binary functional graphs. Therefore the graph in Figure 1 can be more specifically labeled as a binary functional graph.

2.2 Previous work

Daniel Cloutier did a lot of work analyzing functional graphs using this transition function in [2]. Theorem 1 from his paper proves an important result needed to practically analyze these graphs, and it is restated here:

Theorem 1 (Theorem 1 of [2]) *Let p be fixed and let m be any positive integer that divides $p-1$. Then as g ranges over all integers, there are $\phi(\frac{p-1}{m})$ different functional graphs which are m-ary produced by maps of the form $f : x \mapsto g^x \bmod p$. Furthermore, if r is any primitive root modulo p, and $g \equiv r^a \bmod p$, then the values of g that produce an m-ary graph are precisely those for which $\gcd(a, p-1) = m$.*

This theorem allows a person to not only know how many m-ary graphs can be generated from a given prime p, but allows them to figure out which bases (values of g) will generate such graphs. A generator r is an element of the residue class of p such that any element g can be written as $r^a \equiv g \bmod p$. One only needs to pick a generator and do a simple check on each possible a to see if $\gcd(a, p-1) = m$, and use r^a as the base if this is the case.

This result should also make it evident that it makes sense to separate our analysis of the graphs based on what type of m-ary graph it is. The author of [2] showed that this is indeed the case; considering all possible graphs for a given p does not produce results similar to a random functional graph, even if there are many possible types of graph (a result that arises when $p-1$ is very composite). When permutations and binary functional graphs, which correspond to $m = 1$ and $m = 2$, were considered separately however, a more satisfying analysis could be performed.

The author of [2] extended a technique from [5] which uses generating functions to determine the expected values for some parameters of interest in random binary functional graphs. The techniques used will be explored more in depth in the Theoretical Values section, but the parameters analyzed in each graph were number of components, number of cyclic nodes, number of tail nodes, number of terminal nodes, average cycle length, average tail length, maximum cycle length and maximum tail length. For a given prime p, the author of [2] generated as many binary functional graphs as possible and collected data on these parameters for each one. Since each graph had the same number of nodes, these results could then be combined and compared to the theoretical parameter values random binary functional graphs have. The results for the 3 primes he collected data on indicated that these graphs indeed had a similar structure to random binary functional graphs.

The work on binary functional graphs in [2] has been extended further to take an even closer look at the graphs generated. Principally, this paper seeks to analyze the variance in each of the parameters mentioned before, as well as the maximum tail length, and also seeks an exact value for many of the theoretical parameters instead of an asymptotic one. Many more tests have also been run over a much wider range of primes in order to increase the confidence in the accuracy of the results.

3 Theoretical Values

3.1 Techniques

The first step to creating the generating functions needed is to describe the structure of the graphs, as in [5]. A binary functional graph will be a set of components, each of which is made up of a cycle of nodes with a binary tree attached to each node. A binary tree is a node with up to 2 binary trees attached to it. Converting this English description to the same notation as [5] yields

BinFunGraph	=	set(Components)
Component	=	cycle(Node*BinaryTree)
BinaryTree	=	Node + Node*set(BinaryTree, cardinality = 2)
Node	=	Atomic Unit

Using the techniques of [5], this can then be converted to the following generating functions:

$$f(z) = e^{c(z)} = \frac{1}{1 - zb(z)} \tag{2}$$

$$c(z) = \log \frac{1}{1 - zb(z)} \tag{3}$$

$$b(z) = z + \frac{1}{2} zb^2(z) \tag{4}$$

f here generates functional graphs, so the coefficient of z^n in the Taylor expansion of $f(z)$ will be $\frac{f_n}{n!}$ where f_n is the number of binary functional graphs of size n. Similarly, the coefficients of z^n in $c(z)$ and $b(z)$ will tell us about the number of components and the number of binary trees. Solving the quadratic equation for $b(z)$, and using the only answer that makes sense, gives a simpler representation of f and c:

$$f^*(z) = \frac{1}{\sqrt{1 - 2z^2}} \tag{5}$$

$$c^*(z) = \log \frac{1}{\sqrt{1 - 2z^2}} \tag{6}$$

In [2], the author uses singularity analysis on these generating functions to get an asymptotic form for the coefficients. This paper takes a different route, seeking to solve for the coefficient directly where possible. The first step is to create a differential equation $y(z)$ with a set of initial conditions which is satisfied by the function. Applying this process to f^* yields

$$y(0) = 1$$

$$2zy(z) + (-1 + 2z^2)\left(\frac{d}{dz} y(z)\right) = 0$$

One can then find the power series solution to this differential equation and use it to create the recurrence $u(n)$, which will give the nth Taylor coefficient for the generating function around 0. The initial condition for the differential equation is used to create the base case for the recurrence. In this case, solving for this recurrence gives

$$u(0) = 1$$

$$u(1) = 0$$

$$(2n + 2)u(n) - (n + 2)u(n + 2) = 0$$

After studying this result briefly, one can see that $u(2n + 1)$ will be 0 for any positive n. This follows directly from the description of binary functional graphs, since such a graph cannot be created with an odd number of nodes. Both the conversion to a differential equation and the conversion to a recurrence relation were done using functions from [6]; more documentation can be found there. Solving this recurrence yields

$$g(n) = \frac{2^{\frac{n}{2}}\Gamma(n/2 + 1/2)}{\sqrt{\pi}\Gamma(n/2 + 1)} \tag{7}$$

where $\Gamma(n) = \int_{t=0}^{\infty} e^{-t} t^{n-1}$. $g(n)$ therefore will be the number of possible binary functional graphs of size n divided by $n!$. In the following sections, a similar technique will be used to get functions for parameters of interest.

3.2 Means

Theorem 2 *The expected number of components, number of cyclic nodes, average cycle length and average tail length in a random binary functional graph of size n are*

$$\text{Number of Components} \quad \left(\frac{1}{2}\right)\Psi\left(\frac{n}{2}+\frac{1}{2}\right)+\left(\frac{1}{2}\right)\gamma+\ln(2) \tag{8}$$

$$\text{Number of Cyclic Nodes} \quad \frac{\sqrt{\pi}\Gamma(\frac{n}{2}+1)-\Gamma(\frac{n}{2}+\frac{1}{2})}{\Gamma(\frac{n}{2}+\frac{1}{2})} \tag{9}$$

$$\text{Average Cycle Length} \quad \frac{(\frac{1}{2})\sqrt{\pi}\Gamma(\frac{n}{2}+1)}{\Gamma(\frac{n}{2}+\frac{1}{2})} \tag{10}$$

$$\text{Average Tail Length} \quad \frac{\sqrt{\pi}\Gamma(2+\frac{n}{2})-n\Gamma(\frac{n}{2}+\frac{1}{2})-\Gamma(\frac{n}{2}+\frac{1}{2})}{n\Gamma(\frac{n}{2}+\frac{1}{2})} \tag{11}$$

Here, $\Psi(n) = (\frac{d}{dx}\Gamma(x))/\Gamma(x)$ and γ represents the Euler constant, which is approximately 0.57721566.

Proof. The generating functions for these parameters were developed in [2], and in following with the notation from [5], they are called $\Xi(z)$. Since the equations can get quite large, only the process for computing the average cycle length is reproduced here. The other functions are created in exactly the same manner, but can get quite messy. The generating function $\Xi(z)$ for the average tail length is:

$$\Xi(z) = \frac{2z^2(1-\sqrt{1-2z^2})}{(1-2z^2)^2} + \frac{2z^2}{(1-2z^2)^{3/2}} \tag{12}$$

Using the same techniques as earlier, a differential equation can be found that is satisfied by $\Xi(z)$, which can then be used to get the following recurrence:

$$\begin{aligned} 4u(n) - 4u(n+2) + u(n+4) \\ u(1) = 0 \\ u(0) = 0 \\ u(2) = 2 \\ u(3) = 0 \end{aligned} \tag{13}$$

This recurrence can then be solved to give the surprisingly simple solution of

$$[z^n]\Xi(z) = 2^{n/2}(n/2) \tag{14}$$

This is of course only true when n is even. To find the expected value for the cycle length for a given graph, this result needs to be normalized. First it must be multiplied by $n!$, since c is the actual parameter of interest and $[z^n]\Xi(z) = c/n!$. The result should also be divided by the total number of binary functional graphs (to get an expected value), and finally by n because this parameter (as well as average tail) is taking results as seen from a random node in the functional graph. This final division is not performed for the calculation of the expected number of components and cyclic nodes. Since $g(n)$

actually equals the number of binary functional graphs divided by $n!$, the result for Theorem 2 can be obtained by expanding

$$\frac{[z^n]\Xi(z)}{n(g(n))} \tag{15}$$

□

Unfortunately the techniques to calculate maximum tail length and maximum cycle length expected values uses other approximations which prevent a direct recursive calculation at this time. However, since much more data has been collected for this paper, it will be interesting to look at the previously determined asymptotic estimations of these parameters. They are reproduced here for reference:

Theorem 3 (Theorem 7 of [3]) *The asymptotic forms for the expected sizes of the largest cycle and the largest tail in a random binary functional graph of size n, as $n \to \infty$, are*

$$\text{Largest Cycle} \quad \sqrt{\frac{\pi n}{2}} \int_0^\infty \left[1 - \exp\left(-\int_v^\infty e^{-u} \frac{du}{u}\right)\right] dv \approx 0.78248\sqrt{n} \tag{16}$$

$$\text{Largest Tail} \quad \sqrt{2\pi n}\ln(2) - 3 + 2\ln(2) \approx 1.73746\sqrt{n} + 1.61371 \tag{17}$$

3.3 Variances

Theorem 4 *The variance in number of components, number of cyclic nodes, average cycle length and average tail length between binary functional graphs is*

Number of Components

$$\frac{1}{2}\Psi\left(\frac{n}{2}+\frac{1}{2}\right) + \left(\frac{1}{2}\right)\gamma + \ln(2) + \frac{1}{4}\gamma^2 + \gamma\ln(2) + \ln(2)^2 + \sum_{l=0}^{n/2-1} \frac{\Psi(l+\frac{1}{2})}{(2l+1)} - \frac{1}{4}\Psi\left(\frac{n}{2}+\frac{1}{2}\right)^2 \tag{18}$$

Number of Cyclic Nodes

$$-\frac{(-2\Gamma(\frac{n}{2}+\frac{1}{2})^2 - 4\frac{n}{2}\Gamma(\frac{n}{2}+\frac{1}{2})^2 + \sqrt{\pi}\Gamma(\frac{n}{2}+1)\Gamma(\frac{n}{2}+\frac{1}{2}) + \pi\Gamma(\frac{n}{2}+1)^2)}{\Gamma(\frac{n}{2}+\frac{1}{2})^2} \tag{19}$$

Average Cycle Length

$$\frac{1}{12}\frac{(-3\pi\Gamma(\frac{n}{2}+1)^2 - 6\sqrt{\pi}\Gamma(\frac{n}{2}+1)\Gamma(\frac{n}{2}+\frac{1}{2}) + 16\frac{n}{2}\Gamma(\frac{n}{2}+\frac{1}{2})^2 + 8\Gamma(\frac{n}{2}+\frac{1}{2})^2)}{\Gamma(\frac{n}{2}+\frac{1}{2})^2} \tag{20}$$

Average Tail Length

$$\begin{aligned} &-\frac{1}{6}\frac{(-18\Gamma(\frac{n}{2}+\frac{3}{2}) - 8\Gamma(\frac{n}{2}+\frac{3}{2})\frac{n}{2} + 9\frac{n}{2}\sqrt{\pi}\Gamma(\frac{n}{2}+1) + 9\sqrt{\pi}\Gamma(\frac{n}{2}+1))}{(\frac{n}{2})\Gamma(\frac{n}{2}+\frac{1}{2})} \\ &\qquad -\frac{1}{4}\frac{(\sqrt{\pi}\Gamma(\frac{n}{2}+1) + \frac{n}{2}\sqrt{\pi}\Gamma(\frac{n}{2}+1) - 2\Gamma(\frac{n}{2}+\frac{3}{2}))^2}{(\frac{n}{2})^2\Gamma(\frac{n}{2}+\frac{1}{2})^2} \end{aligned} \tag{21}$$

Proof. As the reader can undoubtedly see, these equations can become quite large. For this reason, the proof will again be limited to determining the average cycle value. The variance for a set of data is $\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2$ where N is the number of data points, $\overline{x}$ is the mean, and the x_i are the individual data points. Using some simple algebra this is equivalent to $\frac{1}{N}\left(\sum_{i=1}^{N} x_i^2\right) - \overline{x}^2$, so a generating function for $\left(\sum_{i=1}^{N} x_i^2\right)$ would be very handy.

Using the techniques from [1], a doubly marked function for the average cycle length in a random binary functional graph is

$$\xi(z) = \frac{\ln\left(\frac{1}{(1-u(1-\sqrt{1-2z^2w^2}))}\right)}{\sqrt{1-2z^2}} \tag{22}$$

To compute Ξ, which was used to find the expected average cycle length, the authors of [1] took $\Xi(z) = \frac{\partial}{\partial w \partial u}\xi(z) \mid_{u=1,w=1}$ to eventually obtain $[z^n]\Xi(z)$ (this was actually done for tertiary graphs, but the idea is the same). Since u is marking the parameter of interest, differentiating with respect to u then substituting $u = 1$ is like summing up the parameter for each graph of size n then putting it in front of z^n. A technique to get $\left(\sum_{i=1}^{N} x_i^2\right)$ would be to differentiate with respect to u, then multiply by u to correct for the power of u, then differentiate with respect to u again. Now the coefficient of the u terms is the parameter squared, so substituting in $u = 1$ gives the needed generating function. Doing this with a doubly marked generating function does not introduce any problems, the w was only introduced to account for the fact that the average cycle length is a statistic taken from every node. Therefore the generating function for the total cycle length squared is

$$\Xi^*(z) = \frac{d}{du}\left(u\left(\frac{\partial}{\partial w \partial u}\xi(z) \mid_{w=1}\right)\right) \mid_{u=1} \tag{23}$$

which turns out to be

$$\Xi^*(z) = -\frac{6z^2(-1+\sqrt{1-2z^2})}{(1-2z^2)^2} + \frac{2z^2}{(1-2z^2)^{3/2}} + \frac{4z^2(-1+\sqrt{1-2z^2})^2}{(1-2z^2)^{5/2}} \tag{24}$$

Turning this into a differential equation, then a recurrence relation, and solving for the recurrence relation (as demonstrated previously in the paper) gives

$$[z^n]\Xi^*(z) = \frac{1}{3}\frac{2^{\frac{n}{2}}(\frac{3n}{2}\sqrt{\pi}\Gamma(\frac{n}{2}) - 8\Gamma(\frac{n}{2}+\frac{3}{2}))}{\sqrt{\pi}\Gamma(\frac{n}{2})} \tag{25}$$

Now, let $v(z)$ be the variance in the average cycle length of graphs of size z. The mean has already been described as $\frac{[z^n]\Xi(z)}{n(g(n))}$, so combining this result with the one just obtained, and using them with the simplified equation for variance computed at the beginning of the proof, yields

$$v(z) = \frac{[z^n]\Xi^*(z)g(n)n - ([z^n]\Xi(z))^2}{n^2 g(n)^2} \tag{26}$$

When expanded, $v(z)$ is the same value as in the theorem. The rest of the variances in the parameters (components, cyclic nodes and average tail) are computed similarly with the only difference from the computation of the average cycle being in the normalization, just like the mean.

The key to the success of this technique is having a marked generating function. Since the generating functions for the maximum cycle length and the maximum tail length were not computed using marked generating functions, and no marked generating functions have been found yet, a theoretical value for the variance in these two statistics does not yet exist. Observed values for the variance were collected though in order to better compare the means to the expected values.

4 Observed Values

The same techniques as [2] were used to gather data on functional graphs which use (1) as the transition function. A large prime p is chosen, and a generator r is chosen for the integers modulo p, then a graph is generated using $g = r^a$ for each possible $a \in \{1, 2, 3, 4, ..., p-1\}$ where $\gcd(a, p-1) = 2$. For each prime, the results are then combined to get the average number of components, cyclic nodes, cycle length, tail length, maximum cycle and maximum tail. Data for the variances of these parameters is collected as well. The code to generate these graphs is based on the C++ code written by the author of [2], but it has been converted to C, optimized in places, parallelized and altered to collect data for the computation of the variance.

The total number of graphs generated depends on $\phi(p-1)$. The primes to test were chosen in three sets of 10. The first set is around 100,000, the second set is all safe primes (primes which can be written as $2q+1$ where q is prime) which are basically spread evenly between 110,000 and 200,000, and the last set is primes which are spread over the same interval and have a very composite $p-1$. For a list of the number of graphs generated for each prime, see Appendix A.

4.1 Means

For a given p, $\phi((p-1)/2)$ graphs can be generated. In order to compare the average values of the parameters in these graphs with the values that Theorems 2 and 3 predict, this paper uses a t-test on the means. The idea here is to ascribe some statistical significance to the results, instead of just saying that they look similar. When performing a t-test, the experimenter has a hypothesis in mind which he would like to verify, H_0, and an alternative hypothesis H_a. The t-test on the data produces a t-statistic, which can then be matched with a table to get a P-value. The P-value is the probability of obtaining a test statistic which is stranger, or more extreme, than the one that was observed, given that the null hypothesis is false. So low P-values indicate that the data collected may be unusual, and high P-values give no indication that the null hypothesis is true. In this case, an observed mean (for components for instance) μ_{obs} is compared to a predicted mean μ_{pred} (from Theorem 2 or 3). The null hypothesis is $H_0 : \mu_{pred} = \mu_{obs}$ and the alternative hypothesis is $H_a : \mu_{pred} \neq \mu_{obs}$. The t-test is

$$\frac{(\mu_{pred} - \mu_{obs})\sqrt{N}}{\sigma_{obs}}$$

where $N = p-1$ in this case and σ_{obs} is the standard deviation (which is the square root of the variance) in the observed samples. The number of degrees of freedom here is very large ($df >> 1000$), so the t-statistic is actually the same as a z-statistic and we can use a z table to get the corresponding P-value. Since our null hypothesis allows for $\mu_{pred} > \mu_{obs}$ or $\mu_{pred} < \mu_{obs}$, the value obtained from the table is doubled to get the actual P-value (this is called a 2-tailed t-test). For more information on these tests see [4]. Table 1 gives an example of the results of this test for a single prime and its associated graphs.

	Predicted	Observed	t-value	P-value
Components	6.392	6.389	0.266	0.842
Cyclic Nodes	395.417	395.303	0.123	0.920
Average Cycle	198.208	198.319	-0.173	0.920
Average Tail	197.212	197.178	0.088	1.000
Maximum Cycle	247.495	247.261	0.339	0.764
Maximum Tail	547.935	541.827	8.359	0.000

Table 1: Observed means, t-values and corresponding P-values for the prime 100043

All of the P-values obtained for the maximum tail statistic were very low, and in general the observed values are a bit lower than the theoretical means. Some other statistics have very low P-values for some primes, but have very high P-values for other primes. Based on the definition of a P-value, a uniform distribution is expected when looking at the same statistic over all of the p which have data. Therefore some low values are expected along with some high ones (it would actually be unusual to have P-values which are all very high). To tell if this is actually what's going on for these P-values, one could perform some sort of uniformity test on them. Alternatively, one could perform a normality test (of which there are a large number to choose from) on the test statistics since the t-test uses so many degrees of freedom that it is like a z-test, which uses a normal curve. Table 2 shows the P-value obtained from an Anderson-Darling normality test on each statistic where the 33 t-test results were used as input.

	P-value
Components	0.632
Cyclic Nodes	0.069
Average Cycle	0.483
Average Tail	0.084
Maximum Cycle	0.102

Table 2: P-values obtained by doing Anderson-Darling normality tests on the t-statistics from the mean tests. The maximum tail length statistic is excluded because the P-values from the t-test were consistently very low. Tests were run using Minitab.

4.2 Variance

Techniques similar to those used to compare means will also be useful to take a statistical look at the observed variance compared to the predicted variance. Here the null hypothesis is $H_0 : \sigma^2_{obs} = \sigma^2_{pred}$ and the alternative hypothesis is $H_a : \sigma^2_{obs} \neq \sigma^2_{pred}$. The test used to get the P-value when comparing these is very similar to the one used to compare means; it is

$$\frac{(\sigma^2_{pred} - \sigma^2_{obs})\sqrt{N}}{\tau_{obs}}$$

where τ_{obs} is used to denote $\mathrm{Var}(\{(x_i - \overline{x}_{obs})^2\})$, the variance between the square of the difference between an individual data point (the number of components in a certain functional graph for instance) and the mean value for that statistic. This produces another statistic which can be used to find a P-value in a z-table. Since no theoretical variances have been obtained for the maximum tail and maximum cycle, this test cannot be performed on those statistics. Table 3 gives an example of the results of this test on data from a single prime.

	Predicted	Observed	t-value	P-value
Components	5.158	5.117	1.227	0.230
Cyclic Nodes	42543.192	42227.348	1.123	0.272
Average Cycle	27210.527	20392.727	35.112	0.000
Average Tail	27210.956	7362.882	302.252	0.000
Maximum Cycle		23806.218		
Maximum Tail		26709.198		

Table 3: Predicted and observed variances, t-values and corresponding P-values for the prime 100043. Theoretical values for the maximum cycle and maximum tail variances do not exist

The observed variances in the average tail length and the average cycle length are quite a bit lower than expected. The P-values for the other statistics look like those obtained for the means, so it useful to perform a normality test on these test statistics as well. Table 4 shows the results from these tests.

	P-value
Components	0.485
Cyclic Nodes	0.487

Table 4: P-values obtained by doing Anderson-Darling normality tests on the t-statistics from the variance tests. The average tail length and average cycle length statistics are excluded because the P-values from these t-tests were consistently very low. Test statistics are unavailable for maximum cycle and maximum tail variance. Tests were run using Minitab.

5 Conclusions and Future Work

The results produced by these tests lend some statistical backing to the statement that the average number of components, cyclic nodes, and max cycle length in functional graphs generated using (1) are similar to the expected statistics for random binary functional graphs. The results also let one say with some degree of confidence that the average cycle length and the average tail length of graphs made with (1) are similar to those random graphs. The most interesting result, however, is the extremely low variance in these two statistics when compared to the expected variance. Unfortunately there is not an immediately evident reason for this, so future work could include an investigation as to why the variance is so much lower, but still somewhat consistent for these statistics. Regression tests could be performed to try and match some sort of function which relates the size of a graph and its observed variance. An initial look at the data, however, makes it appear that such a function would have to be very complicated in order to account for the large jumps that can be seen in the observed results. It is still possible that a relatively simple asymptotic formula exists though.

Intuition seems to indicate that while the choice of the average tail/cycle length for some graph generated by the code may be like what a random binary functional graph is like, it may be that (1) imposes some structure that keeps the tail/cycle length from varying too much between the graphs for a given prime. Additional research could also be done to attempt formulate an attack on some system that uses (1) based on the knowledge that the average tail or cycle variance will be lower than random graphs. Perhaps, for instance, if one could figure out the prime a pseudorandom bit generator based on (1) is using, he/she could gain more knowledge about how long the generator will take to enter a cycle and repeat than should be available.

While less dramatic, the lower than average maximum tail length is also curious. While a recursive formula was not found for this statistic, the asymptotic analysis does not change significantly when additional terms are added, so it appears that the difference is not due to the asymptotic estimation. Besides being an asymptotic formula, the computation also uses an integral estimate which could also be off slightly. Additional research could be done here to analyze the lower value, but the reason is probably more subtle and does not appear to be as significant.

Theoretical values for the variance in the maximum tail length and maximum cycle length also remain unsolved. These statistics could give further insight into how closely graphs generated with (1) exhibit characteristics of random graphs. Results in this area might also serve to motivate or direct research into why the variance is so low for average cycle/tail lengths.

A number of open problems from [2] also remain unsolved. It is possible to extend this work to deal with any m-ary graph, but the methods would need to be altered somewhat in order to account for the problems the new variable m would introduce. It was also pointed out that in practice (for the RSA encryption standard, for instance), sometimes composites of large primes are used instead of primes. Additional research could be done to investigate what graphs generated using this composite modulus look like.

Acknowledgments The author would like to thank Joshua Holden for all his great advice and help with the project, and Mark Inlow for all the help with the statistical tests and interpreting the results.

A Number of Graphs

Prime	Number of Graphs
99923	48852
99961	10752
99971	36864
99989	21420
99991	24000
100003	28560
100019	48804
100043	50020
100049	22464
100057	15120
100069	16080
100103	50050
106261	10560
110017	18240
110459	55228
120041	24000
120167	60082
130021	15680
130127	65062
140053	21200
140123	70060
150001	20000
150083	75040
160009	25984
160079	80038
170099	85048
170101	19440
180001	24000
180023	90010
190093	25920
190523	95260
200041	26656
200087	100042

Table 5: Number of binary functional graphs generated for each prime.

B Components

Prime	Mean				Variance			
	Predicted	Observed	t-value	P-value	Predicted	Observed	t-value	P-value
99923	6.391	6.375	1.559	0.134	5.158	5.170	-0.344	0.764
99961	6.391	6.384	0.329	0.764	5.158	5.210	-0.702	0.484
99971	6.391	6.391	0.059	1.000	5.158	5.219	-1.513	0.134
99989	6.392	6.388	0.261	0.842	5.158	5.147	0.213	0.842
99991	6.392	6.411	-1.330	0.194	5.158	5.131	0.567	0.618
100003	6.392	6.415	-1.746	0.090	5.158	5.208	-1.111	0.272
100019	6.392	6.393	-0.160	0.920	5.158	5.198	-1.136	0.272
100043	6.392	6.389	0.266	0.842	5.158	5.117	1.227	0.230
100049	6.392	6.356	2.342	0.022	5.158	5.144	0.272	0.842
100057	6.392	6.364	1.505	0.134	5.158	5.098	0.981	0.368
100069	6.392	6.380	0.661	0.548	5.158	5.139	0.316	0.764
100103	6.392	6.386	0.637	0.548	5.158	5.182	-0.683	0.548
106261	6.422	6.370	2.350	0.022	5.188	5.176	0.169	0.920
110017	6.439	6.478	-2.269	0.028	5.206	5.171	0.611	0.548
110459	6.441	6.453	-1.201	0.230	5.208	5.184	0.738	0.484
120041	6.483	6.480	0.232	0.842	5.249	5.266	-0.328	0.764
120167	6.483	6.480	0.324	0.764	5.250	5.207	1.371	0.194
130021	6.523	6.508	0.801	0.424	5.289	5.170	1.974	0.058
130127	6.523	6.528	-0.515	0.618	5.290	5.300	-0.352	0.764
140053	6.560	6.576	-1.031	0.318	5.326	5.298	0.530	0.618
140123	6.560	6.545	1.745	0.090	5.327	5.324	0.101	0.920
150001	6.594	6.546	2.978	0.004	5.361	5.354	0.118	0.920
150083	6.595	6.609	-1.756	0.090	5.361	5.360	0.023	1.000
160009	6.627	6.600	1.866	0.072	5.393	5.376	0.335	0.764
160079	6.627	6.637	-1.242	0.230	5.393	5.380	0.472	0.690
170099	6.657	6.644	1.609	0.110	5.424	5.407	0.596	0.618
170101	6.657	6.632	1.521	0.134	5.424	5.381	0.738	0.484
180001	6.686	6.627	3.908	0.000	5.452	5.338	2.247	0.028
180023	6.686	6.685	0.098	1.000	5.452	5.486	-1.278	0.230
190093	6.713	6.699	0.985	0.368	5.479	5.444	0.722	0.484
190523	6.714	6.723	-1.203	0.230	5.480	5.515	-1.339	0.194
200041	6.738	6.704	2.423	0.016	5.505	5.462	0.872	0.424
200087	6.738	6.745	-0.919	0.368	5.505	5.517	-0.477	0.690

Table 6: Observed and theoretical mean number of components and variance in number of components for 33 primes. t-statistics are from applying a 2-tail t-test, and were used to obtain the P-value.

C Cyclic Nodes

Prime	Mean				Variance			
	Predicted	Observed	t-value	P-value	Predicted	Observed	t-value	P-value
99923	395.179	395.621	-0.472	0.690	42491.925	42783.286	-1.010	0.318
99961	395.254	392.525	1.372	0.194	42508.159	42563.770	-0.093	1.000
99971	395.274	396.225	-0.878	0.424	42512.432	43214.886	-2.030	0.046
99989	395.310	395.114	0.137	0.920	42520.122	43451.612	-2.129	0.036
99991	395.314	394.853	0.348	0.764	42520.976	42190.923	0.804	0.424
100003	395.338	396.658	-1.079	0.318	42526.103	42733.314	-0.551	0.618
100019	395.369	395.179	0.204	0.842	42532.938	42587.935	-0.189	0.920
100043	395.417	395.303	0.123	0.920	42543.192	42227.348	1.123	0.272
100049	395.429	395.312	0.085	1.000	42545.755	43234.898	-1.573	0.134
100057	395.445	395.858	-0.246	0.842	42549.173	42781.153	-0.454	0.690
100069	395.468	395.211	0.157	0.920	42554.299	43343.408	-1.590	0.134
100103	395.536	394.245	1.407	0.162	42568.825	42136.548	1.536	0.134
106261	407.551	408.433	-0.430	0.690	45199.846	44488.375	1.102	0.272
110017	414.708	414.603	0.066	1.000	46804.777	46880.846	-0.146	0.920
110459	415.543	417.077	-1.663	0.110	46993.651	47011.789	-0.061	1.000
120041	433.234	433.334	-0.069	1.000	51088.590	50971.173	0.242	0.842
120167	433.462	433.045	0.452	0.690	51142.441	51228.171	-0.275	0.842
130021	450.924	453.199	-1.198	0.272	55354.352	56512.667	-1.780	0.090
130127	451.109	452.989	-2.033	0.046	55399.663	55614.724	-0.659	0.548
140053	468.035	467.198	0.504	0.618	59643.012	58631.769	1.676	0.110
140123	468.152	467.555	0.647	0.548	59672.939	59757.006	-0.250	0.842
150001	484.407	483.525	0.494	0.690	63896.358	63807.597	0.130	0.920
150083	484.540	487.669	-3.386	0.000	63931.420	64096.713	-0.468	0.690
160009	500.339	500.424	-0.053	1.000	68175.897	67755.971	0.674	0.548
160079	500.449	500.681	-0.252	0.842	68205.832	67829.395	1.047	0.318
170099	515.904	512.754	3.425	0.000	72490.997	71981.329	1.373	0.194
170101	515.907	514.844	0.555	0.618	72491.852	71368.015	1.459	0.162
180001	530.737	529.276	0.823	0.424	76726.139	75536.984	1.640	0.110
180023	530.769	531.465	-0.753	0.484	76735.549	76786.860	-0.134	0.920
190093	545.440	542.939	1.416	0.162	81042.959	80875.321	0.222	0.842
190523	546.058	547.976	-2.079	0.046	81226.899	81064.068	0.416	0.690
200041	559.556	557.622	1.083	0.318	85298.562	84995.931	0.388	0.764
200087	559.620	562.252	-2.841	0.006	85318.241	85825.849	-1.258	0.230

Table 7: Observed and theoretical mean number of cyclic nodes and variance in number of cyclic nodes for 33 primes. t-statistics are from applying a 2-tail t-test, and were used to obtain the P-value.

D Average Cycle Length

Prime	Mean				Variance			
	Predicted	Observed	t-value	P-value	Predicted	Observed	t-value	P-value
99923	198.090	198.892	-1.236	0.230	27177.770	20631.551	32.900	0.000
99961	198.127	195.151	2.190	0.036	27188.143	19855.515	18.353	0.000
99971	198.137	197.991	0.195	0.920	27190.873	20575.488	28.372	0.000
99989	198.155	198.157	-0.002	1.000	27195.786	20983.340	20.199	0.000
99991	198.157	197.564	0.639	0.548	27196.332	20643.351	22.748	0.000
100003	198.169	198.300	-0.154	0.920	27199.608	20645.738	25.561	0.000
100019	198.185	197.841	0.529	0.618	27203.976	20630.259	32.799	0.000
100043	198.208	198.319	-0.173	0.920	27210.527	20392.727	35.112	0.000
100049	198.214	197.505	0.748	0.484	27212.165	20198.512	24.740	0.000
100057	198.222	198.215	0.006	1.000	27214.349	20680.648	17.942	0.000
100069	198.234	197.509	0.640	0.548	27217.624	20624.273	18.798	0.000
100103	198.268	197.704	0.883	0.424	27226.906	20389.102	35.060	0.000
106261	204.275	206.612	-1.619	0.110	28907.991	22003.465	15.180	0.000
110017	207.854	206.185	1.514	0.134	29933.434	22186.572	22.445	0.000
110459	208.271	209.063	-1.228	0.230	30054.110	22921.145	33.981	0.000
120041	217.117	216.709	0.399	0.764	32670.422	25058.237	21.701	0.000
120167	217.231	216.963	0.418	0.690	32704.828	24617.970	37.913	0.000
130021	225.962	227.173	-0.925	0.368	35395.773	26875.914	19.489	0.000
130127	226.054	227.404	-2.090	0.046	35424.722	27112.730	36.694	0.000
140053	234.518	235.155	-0.546	0.618	38135.661	28874.204	21.605	0.000
140123	234.576	234.477	0.154	0.920	38154.780	28860.896	40.497	0.000
150001	242.704	244.271	-1.255	0.230	40852.904	31212.642	20.560	0.000
150083	242.770	244.601	-2.846	0.006	40875.303	31056.500	41.103	0.000
160009	250.669	253.280	-2.292	0.028	43586.806	33696.107	22.438	0.000
160079	250.724	250.808	-0.130	0.920	43605.929	32881.529	42.924	0.000
170099	258.452	256.815	2.574	0.012	46343.356	34436.466	47.313	0.000
170101	258.454	258.444	0.007	1.000	46343.903	35361.541	19.892	0.000
180001	265.868	264.323	1.252	0.230	49048.767	36552.212	24.427	0.000
180023	265.885	266.585	-1.089	0.318	49054.778	37210.351	44.854	0.000
190093	273.220	273.708	-0.395	0.764	51806.297	39567.572	22.938	0.000
190523	273.529	274.813	-1.995	0.058	51923.794	39467.061	45.135	0.000
200041	280.278	279.706	0.460	0.690	54524.668	41233.270	24.104	0.000
200087	280.310	281.659	-2.097	0.046	54537.238	41358.233	47.640	0.000

Table 8: Observed and theoretical mean average cycle length and variance in average cycle length for 33 primes. t-statistics are from applying a 2-tail t-test, and were used to obtain the P-value.

E Average Tail Length

Prime	Mean				Variance			
	Predicted	Observed	t-value	P-value	Predicted	Observed	t-value	P-value
99923	197.094	197.552	-1.177	0.272	27178.199	7400.315	289.163	0.000
99961	197.131	196.782	0.424	0.690	27188.572	7279.408	135.068	0.000
99971	197.141	198.754	-3.563	0.000	27191.302	7554.003	242.988	0.000
99989	197.159	197.091	0.116	0.920	27196.215	7337.799	194.463	0.000
99991	197.161	196.741	0.763	0.484	27196.761	7263.687	216.992	0.000
100003	197.173	197.926	-1.492	0.162	27200.037	7269.436	228.496	0.000
100019	197.189	197.071	0.303	0.764	27204.405	7401.001	289.771	0.000
100043	197.212	197.178	0.088	1.000	27210.956	7362.882	302.252	0.000
100049	197.218	198.290	-1.859	0.072	27212.594	7464.627	194.361	0.000
100057	197.226	196.768	0.657	0.548	27214.778	7335.733	166.087	0.000
100069	197.238	200.522	-4.739	0.000	27218.054	7722.150	162.246	0.000
100103	197.272	196.897	0.972	0.368	27227.335	7436.388	289.012	0.000
106261	203.279	201.644	1.930	0.058	28908.420	7578.376	141.245	0.000
110017	206.858	205.797	1.602	0.110	29933.863	8004.221	177.592	0.000
110459	207.275	207.261	0.036	1.000	30054.540	8165.605	310.309	0.000
120041	216.121	216.655	-0.877	0.424	32670.851	8910.985	203.307	0.000
120167	216.235	215.620	1.602	0.110	32705.257	8845.946	325.515	0.000
130021	224.966	224.368	0.759	0.484	35396.203	9728.814	160.217	0.000
130127	225.058	226.124	-2.752	0.006	35425.151	9765.134	328.879	0.000
140053	233.521	234.855	-1.907	0.058	38136.090	10380.536	193.514	0.000
140123	233.580	233.598	-0.047	1.000	38155.209	10318.550	351.600	0.000
150001	241.707	242.031	-0.433	0.690	40853.334	11220.811	185.284	0.000
150083	241.773	241.350	1.107	0.272	40875.733	10941.976	374.663	0.000
160009	249.673	248.871	1.206	0.230	43587.235	11474.843	221.910	0.000
160079	249.727	249.619	0.283	0.842	43606.358	11726.168	379.344	0.000
170099	257.455	257.153	0.791	0.484	46343.786	12382.481	390.537	0.000
170101	257.457	257.594	-0.172	0.920	46344.332	12394.279	187.779	0.000
180001	264.871	265.199	-0.440	0.690	49049.196	13336.878	202.746	0.000
180023	264.888	265.843	-2.468	0.016	49055.207	13477.276	385.937	0.000
190093	272.223	273.566	-1.822	0.072	51806.726	14080.980	213.220	0.000
190523	272.532	273.337	-2.094	0.046	51924.223	14072.768	412.570	0.000
200041	279.281	280.272	-1.338	0.194	54525.097	14637.299	224.727	0.000
200087	279.313	278.974	0.883	0.424	54537.668	14731.689	418.880	0.000

Table 9: Observed and theoretical mean average tail length and variance in average tail length for 33 primes. t-statistics are from applying a 2-tail t-test, and were used to obtain the P-value.

F Maximum Cycle

Prime	Mean				Variance
	Predicted	Observed	t-value	P-value	Observed
99923	247.347	247.597	-0.358	0.764	24009.258
99961	247.394	244.195	2.176	0.036	23231.014
99971	247.406	247.104	0.375	0.764	24003.558
99989	247.428	246.834	0.557	0.618	24354.597
99991	247.431	246.591	0.841	0.424	23880.666
100003	247.446	247.461	-0.017	1.000	23985.838
100019	247.465	246.537	1.327	0.194	23881.212
100043	247.495	247.261	0.339	0.764	23806.218
100049	247.502	246.768	0.716	0.484	23652.104
100057	247.512	247.302	0.167	0.920	23985.249
100069	247.527	246.502	0.839	0.424	24030.513
100103	247.569	246.561	1.464	0.162	23746.781
106261	255.070	256.986	-1.229	0.230	25629.420
110017	259.539	257.916	1.362	0.194	25908.517
110459	260.060	260.661	-0.867	0.424	26503.298
120041	271.105	270.519	0.534	0.618	28918.058
120167	271.248	270.751	0.718	0.484	28760.455
130021	282.150	283.448	-0.917	0.368	31463.383
130127	282.265	283.567	-1.873	0.072	31441.827
140053	292.833	292.879	-0.037	1.000	33562.989
140123	292.906	292.427	0.691	0.548	33621.248
150001	303.054	303.790	-0.547	0.618	36165.692
150083	303.137	304.979	-2.659	0.010	35994.045
160009	313.001	315.267	-1.855	0.072	38751.775
160079	313.069	312.615	0.659	0.548	38105.498
170099	322.719	320.296	3.525	0.000	40178.418
170101	322.721	321.416	0.901	0.368	40801.664
180001	331.979	330.325	1.242	0.230	42612.976
180023	332.000	332.393	-0.567	0.618	43236.317
190093	341.159	340.368	0.595	0.618	45900.280
190523	341.544	342.689	-1.650	0.110	45840.435
200041	349.972	348.661	0.978	0.368	47826.847
200087	350.012	351.356	-1.939	0.058	48068.838

Table 10: Observed and theoretical mean maximum cycle and observed variance in maximum cycle length for 33 primes. t-statistics are from applying a 2-tail t-test, and were used to obtain the P-value. No theoretical values for maximum cycle variance exist yet.

G Maximum Tail

Prime	Mean				Variance
	Predicted	Observed	t-value	P-value	Observed
99923	547.606	543.281	5.835	0.000	26833.550
99961	547.710	541.005	4.253	0.000	26730.420
99971	547.738	544.967	3.219	0.002	27307.394
99989	547.787	542.476	4.746	0.000	26833.394
99991	547.793	541.265	6.173	0.000	26832.102
100003	547.826	543.877	4.087	0.000	26665.133
100019	547.870	542.008	7.905	0.000	26827.293
100043	547.935	541.827	8.359	0.000	26709.198
100049	547.952	544.386	3.226	0.002	27440.183
100057	547.974	541.701	4.707	0.000	26848.354
100069	548.007	549.379	-1.049	0.318	27531.440
100103	548.100	540.967	9.702	0.000	27059.016
106261	564.756	554.905	6.093	0.000	27602.038
110017	574.679	565.605	7.192	0.000	29039.726
110459	575.836	569.769	8.243	0.000	29920.698
120041	600.361	595.029	4.580	0.000	32531.293
120167	600.677	592.276	11.482	0.000	32159.836
130021	624.885	616.087	5.877	0.000	35141.251
130127	625.141	621.785	4.553	0.000	35347.570
140053	648.606	645.702	2.169	0.036	38004.996
140123	648.768	642.440	8.629	0.000	37674.203
150001	671.303	665.275	4.243	0.000	40375.416
150083	671.486	665.402	8.280	0.000	40529.603
160009	693.389	685.142	6.479	0.000	42098.894
160079	693.541	686.769	9.240	0.000	42989.106
170099	714.967	707.318	10.494	0.000	45190.679
170101	714.971	709.991	3.269	0.002	45110.111
180001	735.529	730.899	3.243	0.002	48941.776
180023	735.574	731.320	5.766	0.000	48996.888
190093	755.912	752.173	2.656	0.010	51345.311
190523	756.768	752.829	5.365	0.000	51364.029
200041	775.481	770.913	3.207	0.002	54093.098
200087	775.570	769.207	8.657	0.000	54039.057

Table 11: Observed and theoretical mean maximum tail and observed variance in maximum tail length for 33 primes. t-statistics are from applying a 2-tail t-test, and were used to obtain the P-value. No theoretical values for maximum tail variance exist yet.

H Code

H.1 bnprime.h

```
/* Auto generated by bn_prime.pl */
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

```
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */

#ifndef EIGHT_BIT
#define NUMPRIMES 2048
#else
#define NUMPRIMES 54
#endif
static const unsigned int primes[NUMPRIMES]=
{
   2,   3,   5,   7,  11,  13,  17,  19,
  23,  29,  31,  37,  41,  43,  47,  53,
  59,  61,  67,  71,  73,  79,  83,  89,
  97, 101, 103, 107, 109, 113, 127, 131,
 137, 139, 149, 151, 157, 163, 167, 173,
 179, 181, 191, 193, 197, 199, 211, 223,
 227, 229, 233, 239, 241, 251,
#ifndef EIGHT_BIT
 257, 263,
 269, 271, 277, 281, 283, 293, 307, 311,
 313, 317, 331, 337, 347, 349, 353, 359,
 367, 373, 379, 383, 389, 397, 401, 409,
 419, 421, 431, 433, 439, 443, 449, 457,
 461, 463, 467, 479, 487, 491, 499, 503,
 509, 521, 523, 541, 547, 557, 563, 569,
 571, 577, 587, 593, 599, 601, 607, 613,
 617, 619, 631, 641, 643, 647, 653, 659,
 661, 673, 677, 683, 691, 701, 709, 719,
 727, 733, 739, 743, 751, 757, 761, 769,
 773, 787, 797, 809, 811, 821, 823, 827,
 829, 839, 853, 857, 859, 863, 877, 881,
 883, 887, 907, 911, 919, 929, 937, 941,
 947, 953, 967, 971, 977, 983, 991, 997,
```

1009,1013,1019,1021,1031,1033,1039,1049,
1051,1061,1063,1069,1087,1091,1093,1097,
1103,1109,1117,1123,1129,1151,1153,1163,
1171,1181,1187,1193,1201,1213,1217,1223,
1229,1231,1237,1249,1259,1277,1279,1283,
1289,1291,1297,1301,1303,1307,1319,1321,
1327,1361,1367,1373,1381,1399,1409,1423,
1427,1429,1433,1439,1447,1451,1453,1459,
1471,1481,1483,1487,1489,1493,1499,1511,
1523,1531,1543,1549,1553,1559,1567,1571,
1579,1583,1597,1601,1607,1609,1613,1619,
1621,1627,1637,1657,1663,1667,1669,1693,
1697,1699,1709,1721,1723,1733,1741,1747,
1753,1759,1777,1783,1787,1789,1801,1811,
1823,1831,1847,1861,1867,1871,1873,1877,
1879,1889,1901,1907,1913,1931,1933,1949,
1951,1973,1979,1987,1993,1997,1999,2003,
2011,2017,2027,2029,2039,2053,2063,2069,
2081,2083,2087,2089,2099,2111,2113,2129,
2131,2137,2141,2143,2153,2161,2179,2203,
2207,2213,2221,2237,2239,2243,2251,2267,
2269,2273,2281,2287,2293,2297,2309,2311,
2333,2339,2341,2347,2351,2357,2371,2377,
2381,2383,2389,2393,2399,2411,2417,2423,
2437,2441,2447,2459,2467,2473,2477,2503,
2521,2531,2539,2543,2549,2551,2557,2579,
2591,2593,2609,2617,2621,2633,2647,2657,
2659,2663,2671,2677,2683,2687,2689,2693,
2699,2707,2711,2713,2719,2729,2731,2741,
2749,2753,2767,2777,2789,2791,2797,2801,
2803,2819,2833,2837,2843,2851,2857,2861,
2879,2887,2897,2903,2909,2917,2927,2939,
2953,2957,2963,2969,2971,2999,3001,3011,
3019,3023,3037,3041,3049,3061,3067,3079,
3083,3089,3109,3119,3121,3137,3163,3167,
3169,3181,3187,3191,3203,3209,3217,3221,
3229,3251,3253,3257,3259,3271,3299,3301,
3307,3313,3319,3323,3329,3331,3343,3347,
3359,3361,3371,3373,3389,3391,3407,3413,
3433,3449,3457,3461,3463,3467,3469,3491,
3499,3511,3517,3527,3529,3533,3539,3541,
3547,3557,3559,3571,3581,3583,3593,3607,
3613,3617,3623,3631,3637,3643,3659,3671,
3673,3677,3691,3697,3701,3709,3719,3727,
3733,3739,3761,3767,3769,3779,3793,3797,
3803,3821,3823,3833,3847,3851,3853,3863,

3877,3881,3889,3907,3911,3917,3919,3923,
3929,3931,3943,3947,3967,3989,4001,4003,
4007,4013,4019,4021,4027,4049,4051,4057,
4073,4079,4091,4093,4099,4111,4127,4129,
4133,4139,4153,4157,4159,4177,4201,4211,
4217,4219,4229,4231,4241,4243,4253,4259,
4261,4271,4273,4283,4289,4297,4327,4337,
4339,4349,4357,4363,4373,4391,4397,4409,
4421,4423,4441,4447,4451,4457,4463,4481,
4483,4493,4507,4513,4517,4519,4523,4547,
4549,4561,4567,4583,4591,4597,4603,4621,
4637,4639,4643,4649,4651,4657,4663,4673,
4679,4691,4703,4721,4723,4729,4733,4751,
4759,4783,4787,4789,4793,4799,4801,4813,
4817,4831,4861,4871,4877,4889,4903,4909,
4919,4931,4933,4937,4943,4951,4957,4967,
4969,4973,4987,4993,4999,5003,5009,5011,
5021,5023,5039,5051,5059,5077,5081,5087,
5099,5101,5107,5113,5119,5147,5153,5167,
5171,5179,5189,5197,5209,5227,5231,5233,
5237,5261,5273,5279,5281,5297,5303,5309,
5323,5333,5347,5351,5381,5387,5393,5399,
5407,5413,5417,5419,5431,5437,5441,5443,
5449,5471,5477,5479,5483,5501,5503,5507,
5519,5521,5527,5531,5557,5563,5569,5573,
5581,5591,5623,5639,5641,5647,5651,5653,
5657,5659,5669,5683,5689,5693,5701,5711,
5717,5737,5741,5743,5749,5779,5783,5791,
5801,5807,5813,5821,5827,5839,5843,5849,
5851,5857,5861,5867,5869,5879,5881,5897,
5903,5923,5927,5939,5953,5981,5987,6007,
6011,6029,6037,6043,6047,6053,6067,6073,
6079,6089,6091,6101,6113,6121,6131,6133,
6143,6151,6163,6173,6197,6199,6203,6211,
6217,6221,6229,6247,6257,6263,6269,6271,
6277,6287,6299,6301,6311,6317,6323,6329,
6337,6343,6353,6359,6361,6367,6373,6379,
6389,6397,6421,6427,6449,6451,6469,6473,
6481,6491,6521,6529,6547,6551,6553,6563,
6569,6571,6577,6581,6599,6607,6619,6637,
6653,6659,6661,6673,6679,6689,6691,6701,
6703,6709,6719,6733,6737,6761,6763,6779,
6781,6791,6793,6803,6823,6827,6829,6833,
6841,6857,6863,6869,6871,6883,6899,6907,
6911,6917,6947,6949,6959,6961,6967,6971,
6977,6983,6991,6997,7001,7013,7019,7027,

7039,7043,7057,7069,7079,7103,7109,7121,
7127,7129,7151,7159,7177,7187,7193,7207,
7211,7213,7219,7229,7237,7243,7247,7253,
7283,7297,7307,7309,7321,7331,7333,7349,
7351,7369,7393,7411,7417,7433,7451,7457,
7459,7477,7481,7487,7489,7499,7507,7517,
7523,7529,7537,7541,7547,7549,7559,7561,
7573,7577,7583,7589,7591,7603,7607,7621,
7639,7643,7649,7669,7673,7681,7687,7691,
7699,7703,7717,7723,7727,7741,7753,7757,
7759,7789,7793,7817,7823,7829,7841,7853,
7867,7873,7877,7879,7883,7901,7907,7919,
7927,7933,7937,7949,7951,7963,7993,8009,
8011,8017,8039,8053,8059,8069,8081,8087,
8089,8093,8101,8111,8117,8123,8147,8161,
8167,8171,8179,8191,8209,8219,8221,8231,
8233,8237,8243,8263,8269,8273,8287,8291,
8293,8297,8311,8317,8329,8353,8363,8369,
8377,8387,8389,8419,8423,8429,8431,8443,
8447,8461,8467,8501,8513,8521,8527,8537,
8539,8543,8563,8573,8581,8597,8599,8609,
8623,8627,8629,8641,8647,8663,8669,8677,
8681,8689,8693,8699,8707,8713,8719,8731,
8737,8741,8747,8753,8761,8779,8783,8803,
8807,8819,8821,8831,8837,8839,8849,8861,
8863,8867,8887,8893,8923,8929,8933,8941,
8951,8963,8969,8971,8999,9001,9007,9011,
9013,9029,9041,9043,9049,9059,9067,9091,
9103,9109,9127,9133,9137,9151,9157,9161,
9173,9181,9187,9199,9203,9209,9221,9227,
9239,9241,9257,9277,9281,9283,9293,9311,
9319,9323,9337,9341,9343,9349,9371,9377,
9391,9397,9403,9413,9419,9421,9431,9433,
9437,9439,9461,9463,9467,9473,9479,9491,
9497,9511,9521,9533,9539,9547,9551,9587,
9601,9613,9619,9623,9629,9631,9643,9649,
9661,9677,9679,9689,9697,9719,9721,9733,
9739,9743,9749,9767,9769,9781,9787,9791,
9803,9811,9817,9829,9833,9839,9851,9857,
9859,9871,9883,9887,9901,9907,9923,9929,
9931,9941,9949,9967,9973,10007,10009,10037,
10039,10061,10067,10069,10079,10091,10093,10099,
10103,10111,10133,10139,10141,10151,10159,10163,
10169,10177,10181,10193,10211,10223,10243,10247,
10253,10259,10267,10271,10273,10289,10301,10303,
10313,10321,10331,10333,10337,10343,10357,10369,

10391,10399,10427,10429,10433,10453,10457,10459,
10463,10477,10487,10499,10501,10513,10529,10531,
10559,10567,10589,10597,10601,10607,10613,10627,
10631,10639,10651,10657,10663,10667,10687,10691,
10709,10711,10723,10729,10733,10739,10753,10771,
10781,10789,10799,10831,10837,10847,10853,10859,
10861,10867,10883,10889,10891,10903,10909,10937,
10939,10949,10957,10973,10979,10987,10993,11003,
11027,11047,11057,11059,11069,11071,11083,11087,
11093,11113,11117,11119,11131,11149,11159,11161,
11171,11173,11177,11197,11213,11239,11243,11251,
11257,11261,11273,11279,11287,11299,11311,11317,
11321,11329,11351,11353,11369,11383,11393,11399,
11411,11423,11437,11443,11447,11467,11471,11483,
11489,11491,11497,11503,11519,11527,11549,11551,
11579,11587,11593,11597,11617,11621,11633,11657,
11677,11681,11689,11699,11701,11717,11719,11731,
11743,11777,11779,11783,11789,11801,11807,11813,
11821,11827,11831,11833,11839,11863,11867,11887,
11897,11903,11909,11923,11927,11933,11939,11941,
11953,11959,11969,11971,11981,11987,12007,12011,
12037,12041,12043,12049,12071,12073,12097,12101,
12107,12109,12113,12119,12143,12149,12157,12161,
12163,12197,12203,12211,12227,12239,12241,12251,
12253,12263,12269,12277,12281,12289,12301,12323,
12329,12343,12347,12373,12377,12379,12391,12401,
12409,12413,12421,12433,12437,12451,12457,12473,
12479,12487,12491,12497,12503,12511,12517,12527,
12539,12541,12547,12553,12569,12577,12583,12589,
12601,12611,12613,12619,12637,12641,12647,12653,
12659,12671,12689,12697,12703,12713,12721,12739,
12743,12757,12763,12781,12791,12799,12809,12821,
12823,12829,12841,12853,12889,12893,12899,12907,
12911,12917,12919,12923,12941,12953,12959,12967,
12973,12979,12983,13001,13003,13007,13009,13033,
13037,13043,13049,13063,13093,13099,13103,13109,
13121,13127,13147,13151,13159,13163,13171,13177,
13183,13187,13217,13219,13229,13241,13249,13259,
13267,13291,13297,13309,13313,13327,13331,13337,
13339,13367,13381,13397,13399,13411,13417,13421,
13441,13451,13457,13463,13469,13477,13487,13499,
13513,13523,13537,13553,13567,13577,13591,13597,
13613,13619,13627,13633,13649,13669,13679,13681,
13687,13691,13693,13697,13709,13711,13721,13723,
13729,13751,13757,13759,13763,13781,13789,13799,
13807,13829,13831,13841,13859,13873,13877,13879,

13883,13901,13903,13907,13913,13921,13931,13933,
13963,13967,13997,13999,14009,14011,14029,14033,
14051,14057,14071,14081,14083,14087,14107,14143,
14149,14153,14159,14173,14177,14197,14207,14221,
14243,14249,14251,14281,14293,14303,14321,14323,
14327,14341,14347,14369,14387,14389,14401,14407,
14411,14419,14423,14431,14437,14447,14449,14461,
14479,14489,14503,14519,14533,14537,14543,14549,
14551,14557,14561,14563,14591,14593,14621,14627,
14629,14633,14639,14653,14657,14669,14683,14699,
14713,14717,14723,14731,14737,14741,14747,14753,
14759,14767,14771,14779,14783,14797,14813,14821,
14827,14831,14843,14851,14867,14869,14879,14887,
14891,14897,14923,14929,14939,14947,14951,14957,
14969,14983,15013,15017,15031,15053,15061,15073,
15077,15083,15091,15101,15107,15121,15131,15137,
15139,15149,15161,15173,15187,15193,15199,15217,
15227,15233,15241,15259,15263,15269,15271,15277,
15287,15289,15299,15307,15313,15319,15329,15331,
15349,15359,15361,15373,15377,15383,15391,15401,
15413,15427,15439,15443,15451,15461,15467,15473,
15493,15497,15511,15527,15541,15551,15559,15569,
15581,15583,15601,15607,15619,15629,15641,15643,
15647,15649,15661,15667,15671,15679,15683,15727,
15731,15733,15737,15739,15749,15761,15767,15773,
15787,15791,15797,15803,15809,15817,15823,15859,
15877,15881,15887,15889,15901,15907,15913,15919,
15923,15937,15959,15971,15973,15991,16001,16007,
16033,16057,16061,16063,16067,16069,16073,16087,
16091,16097,16103,16111,16127,16139,16141,16183,
16187,16189,16193,16217,16223,16229,16231,16249,
16253,16267,16273,16301,16319,16333,16339,16349,
16361,16363,16369,16381,16411,16417,16421,16427,
16433,16447,16451,16453,16477,16481,16487,16493,
16519,16529,16547,16553,16561,16567,16573,16603,
16607,16619,16631,16633,16649,16651,16657,16661,
16673,16691,16693,16699,16703,16729,16741,16747,
16759,16763,16787,16811,16823,16829,16831,16843,
16871,16879,16883,16889,16901,16903,16921,16927,
16931,16937,16943,16963,16979,16981,16987,16993,
17011,17021,17027,17029,17033,17041,17047,17053,
17077,17093,17099,17107,17117,17123,17137,17159,
17167,17183,17189,17191,17203,17207,17209,17231,
17239,17257,17291,17293,17299,17317,17321,17327,
17333,17341,17351,17359,17377,17383,17387,17389,
17393,17401,17417,17419,17431,17443,17449,17467,

```
17471,17477,17483,17489,17491,17497,17509,17519,
17539,17551,17569,17573,17579,17581,17597,17599,
17609,17623,17627,17657,17659,17669,17681,17683,
17707,17713,17729,17737,17747,17749,17761,17783,
17789,17791,17807,17827,17837,17839,17851,17863,
#endif
};
```

H.2 ca3.h

```
#ifndef CA3
#define CA3

#include <stdio.h>
#include<math.h>
#include "bn_prime.h"
#include "mpi.h"


#define STATUS "status.txt"
#define n 10069
#define trials 1676.0
#define M_ARY 2

/*
7 -- 2
11 -- 4
1019 -- 508
2579 -- 1288
10069 -- 1676

99923  -- 48852
99961  -- 10752
99971  -- 36864
99989  -- 21420
99991  -- 24000
100003 -- 28560
100019 -- 48840
100049 -- 22464
100069 -- 16080
100103 -- 50050


100057 -- 15120
100043 -- 50020
106261 -- 10560
```

```
250007 -- 125002
300023 -- 150010
500231 -- 200088
*/


#define bool char
#define false 0
#define true 1

void run();
void run2();
int m_expn(int b, int r, int num);
int m_exp(int b, int r);
long long ml_exp(long long, int, long long);
void computeResults( const int*, const int*, int*, int* );
void setArrays(int*,bool*, int*, bool*);
bool isPrimRoot(int);
bool isRelPrime(int);
void writeTotalResults(int*, int*, int*, int*, int*, int*, int*);

int gcd(int, int);
bool isPrime(int);
bool MillerRabin(int, int, int, int);


#endif
```

H.3 Lindle2.c

```
#include "ca3.h"



int main(int argc, char* argv[]) {


  MPI_Init(&argc, &argv);

run();

  MPI_Finalize();

  return 0;
}
```

H.4 Lindle2ca.c

```
#include "ca3.h"

int mynode, totalnodes;

int m_expn(int b, int r, int num) {
return (int) ml_exp((long long) b, r, (long long) num);
}

int m_exp(int b, int r) {
return m_expn(b, r, n);
}

/*
long long ml_exp(long long b, int r, long long num) {
if (r == 0) return 1;
if(r % 2 == 0) {
long long result = ml_exp(b,r/2,num);
return result * result % num;
}
long long result = ml_exp(b,r/2,num);
return (b * result % num) * result % num;
}
*/

/*taken from wikipedia page*/
long long ml_exp(long long b, int e, long long m) {

   long long result = 1;

   while (e > 0) {
      /*// multiply in this bits' contribution while using modulus to keep result small*/
      if ((e & 1) == 1) result = (result * b) % m;
      e >>= 1;
      b = (b * b) % m;
   }
   return result;
}

void computeResults(const int* distToCycle, const int* cycleSize,
int* allToCycleSum, int* allCycleLengthSum) {

int sumDistToCycle = 0;
int sumCycleSize = 0;
int i = 0;
```

```
for(i = 0; i < n; i++) {
sumDistToCycle += distToCycle[i];
sumCycleSize += cycleSize[i];
}

*allToCycleSum = sumDistToCycle;
*allCycleLengthSum = sumCycleSize;
}


bool isPrimRoot(int base) {
if(!isPrime(n)) return false;

int n_1 = n-1;

if ((unsigned)n_1 > (primes[NUMPRIMES-1]*primes[NUMPRIMES-1]))
printf("Error in Primitive Root Testing, n could have prime factor too large for testing\n");

int n1 = n_1;
int index = 0;
int p;

while(n1 > 1 && index < NUMPRIMES) {
/*find the primes that divide phi(n)*/
if((n1 % primes[index]) == 0) {
p = primes[index];
/* divide out that prime all the way so it isn't tested again */
while((n1 % primes[index] == 0)) n1/=primes[index];
/*if base^phi(n)/p is 1, not a prim root */
if(m_exp(base,n_1/p) == 1) return false;
/*if(isPrime(n1)) return m_exp(base,n_1/n1) == 1; */
if(n1 == 50021) return (m_exp(base,n_1/50021) != 1);
}
index++;
}
return true;
}

bool isPrime(int num) {
int i = 0;
for(i = 0; i < 50;i++) {
if(primes[i] > (unsigned)num) return true;
if((num % primes[i] == 0) && (num!=primes[i])) return false;
}
```

```
int k = 0;
int q = num-1;
while(q % 2 == 0) {
k++;
q >>= 1;
}
srand(time(0));
int a;
for(i = 0; i < 10; i++) {
a = (rand() % (num-2)) + 1;
if(!MillerRabin(num, k, q, a)) return false;
}

return true;
}


bool MillerRabin(int num, int k, int q, int a) {
int n1 = num-1;
if(m_expn(a,q, num) == 1) return true;
int i = 0;
for(i = 0; i < k; i++)
if(m_expn(a,(int)pow(2,i)*q, num) == n1) return true;
return false;
}


bool isRelPrime(int base) {
return gcd(base, n-1) == 1;
}

int gcd(int a, int b) {
  if(a== 0) return b;
  if(b==0) return a;
  int r = a % b;
int d = b;
int c;
while (r > 0) {
c = d;
d = r;
r = c % d;
}
return d;
}

void setArrays(int * cycleSize, bool* visit, int* distToCycle, bool* image){
```

```
int i = 0;
for(i = 0; i < n; i++){
visit[i] = false;
cycleSize[i] = 0;
distToCycle[i] = 0;
image[i] = false;
}
}

void zeroList(int * listArray) {
int i = 0;
for(i = 0; i < n; i++)
listArray[i] = 0;
}


void writeTotalResults(
        int* maxTAll,
        int* maxCAll,
        int* terminalAll,
int* allComponents,
int* allCyclicNodes,
int* allToCycleSum,
int* allCycleLengthSum) {


char fileStr[20];
sprintf(fileStr, "%d_%d_%d.dat", n, M_ARY, mynode);
FILE * out = fopen(fileStr, "w");
/*# cycles base i
//sum of cycle size seen from nodes in base i
//sum of distance to cycle from nodes in base i
//terminal nodes for base i
//max cycle for base i
//max tail for base i
//cyclic nodes for base i */
int i = 2;
for(i = 2; i < n; i++) { /*0 and 1 not considered*/
fprintf(out, "%d %d %d %d %d %d %d\n", allComponents[i], allCycleLengthSum[i],
allToCycleSum[i], terminalAll[i], maxCAll[i], maxTAll[i], allCyclicNodes[i]);
}
fclose(out);

double cComponents = 0;
double cComponentsSquared = 0;
```

```
double cCyclicNodes = 0;
double cCyclicNodesSquared = 0;

double cImageNodes = 0;
/*variance = 0*/

double cMaxCycle = 0;
double cMaxCycleSquared = 0;

double cMaxTail = 0;
double cMaxTailSquared = 0;

double cWeightedCycle = 0;
double cWeightedCycleSquared = 0;

double cWeightedTail = 0;
double cWeightedTailSquared = 0;

for (i = 2; i < n; i++)
{
cComponents += ((double)allComponents[i]) / trials;
cComponentsSquared += (double)allComponents[i] * (double)allComponents[i] / trials;

cCyclicNodes += (double)allCyclicNodes[i] / trials;
cCyclicNodesSquared += (double)allCyclicNodes[i] * (double)allCyclicNodes[i] / trials;

double cycle = (double)allCycleLengthSum[i] / (double)(n-1);
cWeightedCycle += (double)(cycle) / trials;
cWeightedCycleSquared += (double)(cycle*cycle) / (trials);

double tail = (double)allToCycleSum[i] / (double)(n-1);
cWeightedTail += tail / trials;
cWeightedTailSquared += (double)(tail*tail) / (trials);

if (i != n)
{
if (terminalAll[i] > 0)
cImageNodes += ((double)(n-1) - terminalAll[i]) / trials;
cMaxCycle += (double)maxCAll[i] / trials;
cMaxCycleSquared += (double)maxCAll[i]*(double)maxCAll[i] / trials;

cMaxTail += (double)maxTAll[i] / trials;
cMaxTailSquared += (double)maxTAll[i]*(double)maxTAll[i] / trials;
}
}
```

```
double cComponentsTot = 0;
double cComponentsSquaredTot = 0;

double cCyclicNodesTot = 0;
double cCyclicNodesSquaredTot = 0;

double cImageNodesTot = 0;
double cMaxCycleTot = 0;
double cMaxCycleSquaredTot = 0;

double cMaxTailTot = 0;
double cMaxTailSquaredTot = 0;

double cWeightedCycleTot = 0;
double cWeightedCycleSquaredTot = 0;

double cWeightedTailTot = 0;
double cWeightedTailSquaredTot = 0;



MPI_Reduce( &cComponents, &cComponentsTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce( &cComponentsSquared, &cComponentsSquaredTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Reduce( &cCyclicNodes, &cCyclicNodesTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce( &cCyclicNodesSquared, &cCyclicNodesSquaredTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Reduce( &cImageNodes, &cImageNodesTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Reduce( &cMaxCycle, &cMaxCycleTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce( &cMaxCycleSquared, &cMaxCycleSquaredTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Reduce( &cMaxTail, &cMaxTailTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce( &cMaxTailSquared, &cMaxTailSquaredTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Reduce( &cWeightedCycle, &cWeightedCycleTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce( &cWeightedCycleSquared, &cWeightedCycleSquaredTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WOR

MPI_Reduce( &cWeightedTail, &cWeightedTailTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce( &cWeightedTailSquared, &cWeightedTailSquaredTot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD


if (mynode == 0)
{
char res[20];
sprintf(res, "results_%d.dat", n);
```

```
FILE * r = fopen(res, "w");

double ComponentsVariance = cComponentsSquaredTot - cComponentsTot*cComponentsTot;
double CyclicNodesVariance = cCyclicNodesSquaredTot - cCyclicNodesTot*cCyclicNodesTot;
double WeightedCycleVariance = cWeightedCycleSquaredTot - cWeightedCycleTot*cWeightedCycleTot;
double WeightedTailVariance = cWeightedTailSquaredTot - cWeightedTailTot*cWeightedTailTot;
double MaxCycleVariance = cMaxCycleSquaredTot - cMaxCycleTot*cMaxCycleTot;
double MaxTailVariance = cMaxTailSquaredTot - cMaxTailTot*cMaxTailTot


fprintf(r, "components: %lf \n", cComponentsTot);
fprintf(r, "components variance: %lf \n", ComponentsVariance);

fprintf(r, "cyclic nodes: %lf \n", cCyclicNodesTot);
fprintf(r, "cyclic nodes variance: %lf\n", CyclicNodesVariance);

fprintf(r, "avg cycle: %lf\n", cWeightedCycleTot);
fprintf(r, "avg cycle variance: %lf\n", WeightedCycleVariance);

fprintf(r, "avg tail: %lf\n", cWeightedTailTot);
fprintf(r, "avg tail variance: %lf\n", WeightedTailVariance);

fprintf(r, "image nodes: %lf\n", cImageNodesTot);
fprintf(r, "max cycle: %lf\n", cMaxCycleTot);
fprintf(r, "max cycle variance: %lf\n", MaxCycleVariance);

fprintf(r, "max tail: %lf\n", cMaxTailTot);
fprintf(r, "max tail variance: %lf\n", MaxTailVariance);

fclose(r);
}
}


void run() {


MPI_Comm_size(MPI_COMM_WORLD, &totalnodes);
MPI_Comm_rank(MPI_COMM_WORLD, &mynode);

FILE * s;
if (mynode == 0)
{
s = fopen(STATUS, "w");
fprintf(s, "Allocating...\n");
fclose(s);
```

```
}

/*status << "Allocating...\n";
status.close();*/

bool visit[n];
bool image[n];
/*maximum tail length for base [i]*/
int maxTAll[n];
/*maximum cycle lenghth for base [i]*/
int maxCAll[n];
/*terminal nodes for base [i]*/
int terminalAll[n];

/*size of cycle for the component this node is a part of*/
int cycleSize[n];
/*distance to cycle from node n (0 if node n is cyclic)*/
int distToCycle[n];

/*The number of nodes -in- cycles of size [i] for current n
//int *allCResults = new int[n+1];
//The number of nodes which are [i] away from their cycle
//int *allTResults = new int[n+1];
//The number of tail nodes that lead to a cycle of size [i]
//int *allToCycleResults = new int[n+1];*/

/*Number of components for base i*/
int allComponents[n+1];
/*Number of image nodes for base i*/
int allCyclicNodes[n+1];
/*Sum of all nodes' distance to cycle for base i */
int allToCycleSum[n+1];
/*Sum of each node's cycle length for base i */
int allCycleLengthSum[n+1];

/*max cycle, max tail */
int mC, mT;
int next, loc, baseTail, cycleLength, terminal;
int root, exp, base;

int listArray[n];
int listSize = 0;


if (mynode == 0)
{
```

```
s = fopen(STATUS, "w");
fprintf(s, "zeroing...\n");
fclose(s);
}


/*initialize arrays to 0 */
int i = 0;
for(i = 0; i < n; i++){
if(i < n) {
  maxTAll[i] = 0;
  maxCAll[i] = 0;
  terminalAll[i] = 0;
}
}
allComponents[n] = 0;
allCyclicNodes[n] = 0;
allToCycleSum[n] = 0;
allCycleLengthSum[n] = 0;

double t;
if (mynode == 0)
t = MPI_Wtime();

double tt;
double expTime = 0;
double tailTime = 0;
double intoCycleTime = 0;
double cycleTime = 0;
double resultsTime = 0;

if (mynode == 0)
{
s = fopen(STATUS, "w");
fprintf(s, "Finding a PR...\n");
fclose(s);
}

/*find the smallest primitive root*/
for(root = 1; !isPrimRoot(root); root++);

if (mynode == 0)
{
s = fopen(STATUS, "a");
fprintf(s, "Prim root is %d...\n", root);
fclose(s);
```

```
}

int count = -1;

for(exp = 0; exp < n; exp ++) {
  if(exp % 100 == 0 && mynode == 0) {
    s = fopen(STATUS, "w");
    fprintf(s, "Exp is %d\n", exp);
    fclose(s);
  }

  /*discard all but the bases which will make the target M-ARY graphs*/
  if(gcd(exp,n-1) != M_ARY) continue;
  count++;
  if(count % totalnodes != mynode) continue;

  base = m_exp(root,exp);
  mC = 0;
  mT = 0;
  /*0 out everything*/
  setArrays(cycleSize, visit, distToCycle, image);

  /*begin making graph, using gamma(i) = base^i mod n*/
for(i = 1; i < n; i++) {
if (visit[i])
continue;

next = i;
listArray[0] = next;
listSize = 1;

tt = MPI_Wtime();
while(!visit[next]){
visit[next] = true;
next = m_exp(base,next);
image[next] = true;
listArray[listSize] = next;
listSize++;
}
expTime += MPI_Wtime() - tt;

int j = 0;
if(cycleSize[next] != 0) {
if(distToCycle[next] == 0) {/*all tail into cycle*/

tt = MPI_Wtime();
```

```
cycleLength = cycleSize[listArray[listSize-1]];
if(listSize - 1 > mT) mT = listSize - 1;
for(j = 0; j < listSize-1; j++){
distToCycle[listArray[j]] = listSize - 1 - j;
cycleSize[listArray[j]] = cycleLength;
}
intoCycleTime += MPI_Wtime() - tt;

} else {/*extension of tail*/

tt = MPI_Wtime();
baseTail = distToCycle[listArray[listSize-1]];
cycleLength = cycleSize[listArray[listSize-1]];
if(listSize-1 + baseTail > mT) mT = listSize-1 + baseTail;
for(j = 0; j < listSize-1; j++) {
distToCycle[listArray[j]] = baseTail + listSize - 1 - j;
cycleSize[listArray[j]] = cycleLength;
}
tailTime += MPI_Wtime() - tt;
}
} else {/*new cycle found*/

tt = MPI_Wtime();
/*loc will be the first node in the cycle we ran in to*/
int repeat = listArray[listSize-1];
for(j = 0; listArray[j] != repeat; j++);

int firstCycle = j;
cycleLength = listSize - (j+1);
if(cycleLength > mC) mC = cycleLength;

if(firstCycle > mT) mT = firstCycle;
/*mark each tail node along the way with how far it is to*/
/*the cycle (marked as a negative number)*/
for(j = 0; j < firstCycle; j++) {
distToCycle[listArray[j]] = firstCycle - j;
cycleSize[listArray[j]] = cycleLength;
}
/*mark each cycle nodes with how big the cycle is*/
for(j = firstCycle; j < listSize - 1; j++)
cycleSize[listArray[j]] = cycleLength;

allComponents[base]++;
allCyclicNodes[base] += cycleLength;

cycleTime += MPI_Wtime() - tt;
```

```
}
}

tt = MPI_Wtime();

terminal=0;
for(i = 1; i < n; i++)
  if(!image[i]) terminal++;
maxTAll[base] = mT;
maxCAll[base] = mC;
terminalAll[base] = terminal;
computeResults(distToCycle, cycleSize, &allToCycleSum[base], &allCycleLengthSum[base]);

resultsTime += MPI_Wtime() - tt;
}
if (mynode == 0)
{
s = fopen(STATUS, "w");
fprintf(s, "Writing Results...\n");
fclose(s);
}
writeTotalResults(
  maxTAll,
  maxCAll,
  terminalAll,
  allComponents,
  allCyclicNodes,
  allToCycleSum,
  allCycleLengthSum);


if (mynode == 0)
{
s = fopen(STATUS, "w");
fprintf(s, "%lf minutes...\n Exiting...\n%lf %lf %lf %lf %lf\n", (MPI_Wtime() - t)/60, expTime, tailTi
fclose(s);
}

printf("%lf\n", expTime);

}
```

References

[1] Max Brugger and Christina Frederick. The discrete logarithm problem and ternary functional graphs. *Rose-Hulman Undergraduate Mathematics Journal*, 8(2), 2007. http://www.rose-hulman.edu/mathjournal/archives/2007/vol8-n2/paper8/v8n2-8pd.pdf.

[2] Daniel R. Cloutier. Mapping the discrete logarithm. Senior thesis, Rose-Hulman Institute of Technology, 2005. http://www.rose-hulman.edu/ holden/Preprints/MapDisLog.pdf.

[3] Daniel R. Cloutier and Joshua Holden. Mapping the discrete logarithm. 2006. http://xxx.lanl.gov/abs/math.NT/0605024.

[4] Jay Devore and Nicholas Farnum. *Applied Statistics for Enginrs and Scientists*, chapter 8. Curt Hinrichs, 2005.

[5] Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. In *Advances in cryptology—EUROCRYPT '89 (Houthalen, 1989)*, volume 434 of *Lecture Notes in Comput. Sci.*, pages 329–354. Springer, Berlin, 1990.

[6] B. Salvy and P. Zimmermann. Gfun: a maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 1994.