The Discrete Logarithm Problem and Ternary Functional Graphs

Max F. Brugger
Oregon State University

Christina A. Frederick
University of Maryland, College Park

The Discrete Logarithm Problem and Ternary Functional Graphs

Max F. Brugger, Christina A. Frederick

Adviser: Joshua B. Holden

Mathematical Sciences Technical Report Series
MSTR 07-02

August 15, 2007

The Discrete Logarithm Problem and Ternary Functional Graphs

Max F. Brugger
Oregon State University

Christina A. Frederick
University of Maryland, College Park

The Discrete Logarithm Problem and Ternary Functional Graphs

Max F. Brugger   Christina A. Frederick

August 15, 2007

Abstract

Encryption is essential to the security of transactions and communications, but the algorithms on which they rely might not be as secure as we all assume. In this paper, we investigate the randomness of the discrete exponentiation function used frequently in encryption. We show how we used exponential generating functions to gain theoretical data for mapping statistics in ternary functional graphs. Then, we compare mapping statistics of discrete exponentiation functional graphs, for a range of primes, with mapping statistics of the respective ternary functional graphs.

1   Introduction

> "As far as the laws of mathematics refer to reality, they are not certain, as far as they are certain, they do not refer to reality." –Albert Einstein

Einstein's quote above is especially pertinent in number theory; usually it is one of the mathematical fields least connected to reality. It is a terrible irony then that one of the most applicable open problems within number theory — the Discrete Logarithm Problem — remains unsolved; its solution, though necessary to our assurance of security in Internet transactions and communications, is entirely uncertain.

1.1   Motivation

To explain the motivation behind this project and this paper, we need to explain what the Discrete Logarithm Problem is. Put simply, it is the problem of computing the discrete logarithm efficiently. Let us look at this problem in the context of Pohlig-Hellman encryption.

Pohlig-Hellman encryption is a Private Key System that enables secure communication between two parties. We name these two parties Alice and Bob, respectively. Pohlig-Hellman encryption is called a "Private Key" system because Alice and Bob both know and use a large number, denoted e, but keep it secret from everyone else. Alice uses e to encrypt her message, denoted P for plain-text, using a form of the discrete exponentiation function:

$$C \equiv P^e \bmod p$$

where C is the encoded message, p is the prime modulus, and everything, including even the message length and the length of the message blocks, into which the message is partitioned, is kept public, except for e. Let us recognize a third party, Eve, who will represent an eavesdropper or hacker, or, less maliciously, everyone else. To better understand how this encryption works, we will look at what each party knows:

Alice	Bob	Eve
$P, e, p,$ block length, C	$e, p,$ block length, C	$p,$ block length, C

Bob can decrypt Alice's message by finding the modular inverse of e, a number called d that satisfies the following equation:

$$(P^e)^d \equiv P \bmod p.$$

Dividing both sides by P, we obtain

$$P^{ed-1} \equiv 1 \bmod p,$$

which requires $(p-1)|(ed-1)$ by Fermat's Little Theorem.

Therefore, there exists some k such that

$$k(p-1) = ed - 1,$$

implying

$$p^{ed-1} \equiv p^{k(p-1)} \equiv (p^{(p-1)})^k \equiv 1^k \equiv 1 \bmod p,$$

and therefore,

$$ed - 1 \equiv 0 \bmod p - 1,$$

again, by Fermat's Little Theorem. Therefore, $d \equiv e^{-1} \bmod p - 1$. This inverse may be found quickly using the Euclidean Algorithm.

All that remains is to exponentiate C with the decryption key, d:

$$C^d \equiv (P^e)^d \equiv P \bmod p$$

yielding for Bob the plain-text of the message: P.

The strength of this encryption method relies on Eve not knowing e or d and thus not being able to use Bob's method of decryption. If we knew that there were only a finite set of methods that Eve could use, we would be able to compute the running time of each and assure the safety of the communication by, for example, changing keys (e or d) just before she discovers and uses them.

The Discrete Logarithm Problem is so titled because one of the most basic (and computationally-intensive) methods Eve has at her disposal is the computation of the Discrete Logarithm function. The computation of this function is an example of a Known Plain-text Attack.

In a Known Plain-text Attack, (where Eve knows P and wants to find e) Eve uses her knowledge of the relationship between C and P,

$$C \equiv P^e \bmod p$$

and, instead of taking the root that Bob can take: $\sqrt[e]{C} \equiv P \bmod p$, she computes the following:

$$\log_P(C) \equiv e \bmod p - 1$$

which is called the Discrete Logarithm Function, so-called "discrete" because of the integer values it takes as its domain and range.

Eve has faster methods at her disposal than guessing and checking, but these known methods are still computationally infeasible and we will not discuss them here. In the context of this paper, we will consider Eve to be using a Known Plaintext attack: finding e such that $C \equiv P^e \bmod p$, in which case the discrete logarithm will actually output something for e.In particular, this is a good reason to pick a large e, and in general this is a good reason to make sure that the Discrete Logarithm is a difficult function to compute.

1.2 History of the DLP

Since the birth of the computer era, encryption has become steadily more and more important in our everyday lives. Many of the methods used to keep our communications secret and our important information private involve the Discrete Logarithm Problem in some way. We will cover the topic briefly here by citing examples of encryption methdods that use the Discrete Exponentiation Function, $y \equiv x \bmod p$. In the Motivation section, we mentioned Pohlig-Hellman encryption and showed how its security relied on the difficulty of computing the Discrete Logarithm Function. RSA Encryption also uses the Discrete Exponentiation Function for encryption, and so the computation of the Discrete Logarithm Problem could be used in an attack on RSA Encryption.

1.2.1 RSA Encryption

RSA encryption, developed at MIT in 1977 by Ron Rivest, Adi Shimar, and Leonard Adleman, is used to securely transmit messages by way of encryption. It is similar to Pohlig-Hellman encryption, except the factorization of the modulus (this time a product of two primes, $n = pq$) is kept secret and the encryption key, e, is made public. Therefore, the table of who knows what looks like this:

Alice	Bob	Eve
$P, e, n, p, q,$ block length, C	$e, n, p, q, d,$ block length, C	$e, n,$ block length, C

where n is the modulus, and Bob can easily figure out d from his given information. What are some of the mathematical attacks on RSA that have been tried?

- Find p, q. Then $\phi(n) = (p-1)(q-1)$, and $de \equiv 1 \bmod \phi(n)$.

- Find $\phi(n)$ some other way, and then find d with e and the Euclidean Algorithm.

- Find d some other way.

- Decrypt some other way.

It turns out that if you succeed at the third option, finding d some other way, you get the first item for free. If we can find d, then, using that $de \equiv 1 \bmod \phi(n)$, we find that $de - 1 = h\phi(n)$ for some (probably) small h. This gives us quick, easily checkable options for $\phi(n)$, yielding p and q. The same is true for the second item: if we find $\phi(n)$, we may take n and derive p and q, but that is a fun exercise and so we leave it to the reader.

1.2.2 Diffie-Hellman "Key Agreement"

The Discrete Logarithm Problem applies to many other encryption protocols — not just to the methods of plain-text encryption. In "Diffie-Hellman Key Agreement", the Discrete Exponentiation Function is used to generate the key that will be used to encrypt the message, thus determining the security of the key, and not the message (as in Pohlig-Hellman and RSA).

We consider Diffie-Hellman to involve two parties: Alice and Bob, again. The only things they keep private are their respective private keys, which are never explicitly communicated to anyone that might steal them. Yet through this method they can agree on the same key and can then encrypt and decrypt messages securely. Here's how it works:

Alice comes up with some number α and Bob comes up with some number β and they keep them private from everyone, including each other. They publish two numbers completely publicly: γ and p. Alice and Bob follow the processes below:

Alice	Process	Bob	Eve
α		β	p, γ
α	$\rightarrow a \equiv \gamma^\alpha \bmod p \rightarrow$	β, a	p, γ, a
α, b	$\leftarrow b \equiv \gamma^\beta \bmod p \leftarrow$	β, a	p, γ, a, b

Alice then computes $b^\alpha \bmod p$, obtaining $\gamma^{\alpha\beta} \bmod p$. Similarly, Bob computes $a^\beta \bmod p$, obtaining $\gamma^{\alpha\beta} \bmod p$. Thus, they have a private key that they may use to encrypt their plaintext, as in Pohlig-Hellman Encryption.

If the Discrete Logarithm Function were easily computed, though, the entire security of this method would be brought down. If Eve could solve the Discrete Logarithm Problem, then, given a and g, she could solve $\alpha \equiv \log_\gamma a \bmod p$ and obtain α, and then construct the private key. Likewise for β. It is absolutely necessary, then, that the Discrete Logarithm Function be secure.

1.2.3 Previous Student Research

In 2005, Daniel Cloutier worked on a similar project to ours involving Binary Functional Graphs in [3]. For his research, he wrote two C++ programs, both of which took in a prime p and, for every base, γ, found out where each element x pointed, under the mapping: $x \rightarrow \gamma^x \bmod p$. One program found this for all of the γ's and the other paid attention only to graphs with a certain restriction. In his research, he used the program to look at only the mappings where every element had in-degree 0 or 2, meaning that every element is mapped to by exactly zero or two other elements. He then used the programs to measure certain statistics of the mapping. He then used methods from [4] to estimate what these statistics would be for binary mappings in general, where the requirement of the associated function is removed, but the requirement of in-degree remains. He then compared the two statistics and looked for discrepancies.

We can look at the data this way: if there is little relative error between the functions with the discrete exponentiation function associated and the functions with a random associated function, then the discrete exponentiation function is behaving as if there is a random function operation. This seems to imply that the computation of the inverse cannot, in general, rely on some structure imposed by the function. However, any discrepancy between the two data sets would seem to imply that the function is leaving a trace behind. Any trace of the function could lead to an exploit — a way to more quickly compute the Discrete Logarithm Function.

We decided to continue Cloutier's work by including the data from Ternary Functional Graphs and examining the data similarly to the way mentioned in the paragraph above. After the initial work of running these simulations and pulling out the observed data from Cloutier's program, and the theoretical data from our estimates for ternary functions, we could begin to ask more questions. Is this what we should expect to see for a general prime p? Is the relative error decreasing as we include more and more graphs? Does the m-aryness affect anything, i.e. does it matter if we look at binary or ternary mappings? These are interesting questions, as they have a lot to do with the security of the encryption methods mentioned in previous sections, and they present interesting directions to go with future research.

2   Background

We will use some terminology repeatedly, so it is useful to define it here.

2.1 Definitions

Definition 1. The *Discrete Exponentiation Function* is a well-defined function described by the equation below:

$$y \equiv \gamma^x \bmod p$$

where, in this paper, we will consider p to be the prime modulus of the finite field of integers mod p ($\mathbb{Z}_p$) and γ to be a base in that field. The domain is all x such that $x \in \mathbb{Z}_p$, $x \neq 0$. The co-domain is a subset of the domain.

Definition 2. A *functional graph* is a directed graph with an associated function, $f(x)$. The function is associated in two ways: there is a bijection between the set of nodes in the graph and the elements in the domain of the function, and the directed edge from node a to node b is included in the functional graph if $f(a) = b$.

Definition 3. The graph of the *Discrete Exponentiation Functional Graph* is a functional graph with the associated function: $y \equiv \gamma^x \bmod p$.

In this paper, we will examine the similarities and differences between Discrete Exponentiation Functional Graphs and Ternary Functional Graphs, so we really ought to define Ternary Functional Graphs.

Definition 4. A *Ternary Functional Graph* is a functional graph wherein each node has in-degree equal to exactly 0 or 3. In-degree, for a node b, is defined as the number of elements of the domain, a, such that $f(a) = b$. Thus, equivalently, every element of the domain is the image of exactly 0 or 3 elements.

We are interested in examining certain statistics of Ternary DEFGs and Ternary Functional Graphs. These definitions rely on graph theory concepts that can be defined with surprising rigor. We will refer to a function $\phi(x)$ as the function with which the functional graph is associated.

Definition 5. Let y be a node in a functional graph. It is considered to be an *image node* if there exists at least one x such that $y = \phi(x)$.

Definition 6. Let y be a node in a functional graph. If there exists *no* x such that $y = \phi(x)$, then it is a *terminal node*.

Definition 7. A *component* is the set of all connected tail nodes (nodes contained in the path to a cycle) and cyclic nodes (nodes contained in a cycle).

Definition 8. [4] Consider the sequence of elements $\{x_0, x_1 = \phi(x_0), x_2 = \phi(x_1), \ldots\}$ and note that we will find a value x_j equal to one of $\{x_0, x_1, \ldots x_{j-1}\}$. In graphical terms, this iteration structure is described by a simple path that connects to a cycle. The length of the path, measured by the number of edges, is called the *tail length* of x_0.

Definition 9. As determined in the previous definition, the length of the cycle is called the *cycle length* of x_0. In simpler terms, if we begin at a cyclic node, it is the number of iterations of the function required to return to said cyclic node.

Definition 10. The *cycle length as seen from a node* is calculated by adding up the sum, over each node, of the cycle length that it "sees". In this case, we consider each node to "see" a cycle and to record that cycle's length. We then take the sum over all of the cycle lengths that the nodes see. For example, consider a functional graph consisting of 2 components of size 8 and 4. Within the component of size 8 there is a cycle of length 2 and within the component of size 4 there is a cycle of length 3, then the cycle length as seen from a node is $8 \cdot 2 + 4 \cdot 3 = 28$.
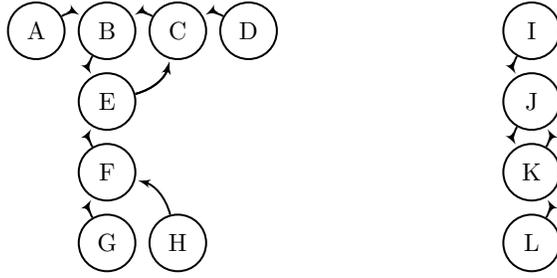
Figure 1: In this binary functional graph there are two connected components. Nodes B, E, and C form a cycle of length 3. Nodes J and K form another cycle of length two. We see tails of length 3 in E, F, G and E, F, H, and in fact they form the maximum tail length of the graph. The cycle length as seen from a node is $8 \cdot 3 + 4 \cdot 2$, since 8 nodes see a cycle of length 3 and 4 nodes see a cycle of length 2.

2.2 Comparing Ternary DEFGs and Ternary Functional Graphs

Consider a ternary functional graph. If you were to choose one at random from the set of all functional graphs and put it up next to a Discrete Exponentiation Functional Graph, what similarities would there be? What differences are there? The idea behind this comparison is that if DEFGs share a lot of characteristics with Ternary Functional Graphs, and there are no discernible differences, then that implies that DEFGs behave as if a random function were being imposed.

This implies that a hacker like Eve would be unable to use any sort of attack that guesses at the structure of the functional graph, because there would be no defining characteristics on which to rely.

To make this comparison, we had to come up with a way to collect data for Ternary Functional Graphs and for Discrete Exponentiation Functional Graphs. To collect these statistics from the Ternary Functional Graphs we used Exponential Generating Function methods to give exact expressions for, for example, the average number of cyclic nodes in a Ternary Functional Graph, over all Ternary Functional Graphs with $p - 1$ nodes.

To collect these statistics from the Discrete Exponentiation Functional Graphs we used a program written by Daniel Cloutier that yields all but one of our desired statistics for all ternary Discrete Exponentiation Functional Graphs with prime modulus p, and thus $p - 1$ nodes (since we do not include node 0).

3 Ternary Functional Graphs

3.1 Exponential Generating Functions

> "A generating function is a clothesline on which we hang up a sequence of numbers for display." –Herbert S. Wilf, [1]

In our research, we have used exponential generating functions to count certain features of Ternary Functional Graphs. To give an imprecise account before our precise definition, exponential generating functions, or EGFs, are formal Taylor series with co-efficient of the nth degree term equal to the number of things we have set up the EGF to count. If we want our EGF to count the number of components in a ternary functional graph, then the number of components over all ternary functional graphs with n nodes is given in the coefficient of the nth term. The methods we use to derive these EGFs are fun and

beautiful; some basic examples will be given in this section, and some more complicated applications will be given in the next.

3.1.1 Definitions

Exponential Generating Function Let $f(x)$ be an exponential generating function. Then $f(x)$ may be expressed as:

$$f(x) = \sum_{n=o}^{\infty} f_n \frac{x^n}{n!}$$

where the f_n are the sequential elements of some sequence. An **ordinary generating function** can be expressed as $\sum_{n=0}^{\infty} f_n x^n$.

3.1.2 Examples

Theorem 1 ([2], Theorem 3.20). *Let f_n be the number of ways to carry out a task on $\{1, \ldots, n\}$ and g_n be the number of ways to carry out another task on the same sequence. Let $F(x) = \sum_{i=0}^{\infty} f_i \frac{x^i}{i!}$ and $G(x) = \sum_{i=0}^{\infty} g_i \frac{x^i}{i!}$.*
Let h_n be the number of ways to:

- *split $\{1, \ldots, n\}$ into the disjoint union of S and T,*

- *carry out Task 1 on S,*

- *carry out Task 2 on S.*

Let $H(x) = \sum_{i=0}^{\infty} h_i \frac{x^i}{i!}$. Then $H(x) = F(x)G(x)$.

Proof. Considering $F(x)G(x)$, as defined in the theorem,

$$F(x)G(x) = \left(f_0 + f_1 x + \frac{f_2 x^2}{2!} + \cdots \right) \cdot \left(g_0 + g_1 x + \frac{g_2 x^2}{2!} + \cdots \right)$$

$$= \sum_{i=0}^{\infty} \left(\sum_{k=0}^{i} \frac{f_k}{k!} \frac{g_{i-k}}{(i-k)!} \right) x^i$$

$$= \sum_{i=0}^{\infty} \left(\sum_{k=0}^{\infty} \binom{i}{k} f_k g_{i-k} \right) \frac{x^i}{i!}$$

Therefore the coefficient of the ith term is the product of the different ways to choose k ($\sum_{k=0}^{i}$), the number of ways to choose k objects from i, for Task 1 ($\binom{i}{k}$), the number of ways to perform Task 1 (f_k) and the number of ways to perform Task 2 (g_{i-k}),

$$= \sum_{i=0}^{\infty} h_i \frac{x^i}{i!} = H(x). \qquad \boxtimes$$

$\square$

We would like to first use these EGFs to give an expression for the number of Ternary Functional Graphs with n nodes. To do this, we will build up to Functional Graphs by first considering Ternary Trees, then components, and then using these to describe Ternary Functional Graphs.

Making a Ternary Tree   We will use Theorem 1 to demonstrate how to use EGFs to make a ternary tree on $\{1, \ldots, n\}$.

First, we split $\{1, \ldots, n\}$ into S=root and T=everything else.

Let us think of this in terms of recursive tasks. Task 0 would be the splitting of the sequence into a root and everything else. The real meat comes afterward when we split up the elements of T into three trees (since we are making a ternary tree) and then, within each tree, repeat the process until we run out of elements.

Task 1 is the task of making S into a root. Since there is only one element S, we consider the case of 0 nodes and 1 node:

$$\text{EGF} = \frac{1x}{1!} = x$$

Task 2 is the task of breaking T into three pieces, T_1, T_2, and T_3. We consider this to be three subtasks. Task 2a is the task of assigning T_1 to the left ternary tree with EGF= $t(x)$, Task 2b is the task of assigning T_2 to the middle ternary tree, with EGF= $t(x)$, and Task 2c is the task of assigning T_3 to the right ternary tree, with EGF= $t(x)$.

Therefore, we want the number of ways to make a ternary tree, $t(x)$. Note that, since we do not care about which "sub"-tree is the left, middle, or right, we must divide the number of ways to perform Task 2 by 3!, as that is the number of ways of permuting 3 things.

By Theorem 1,

$$t(x) = x \left(\frac{t(x)t(x)t(x)}{3!} + 1 \right)$$

$$\implies xt(x)^3 - 6t(x) + 6x = 0.$$

This cubic polynomial has one real and two complex solutions, and we are better suited to keep the expression of t(x) in the above form.

Theorem 2 ([2] Theorem 3.27). *Let $A(x)$ be the EGF for some task:*

$$A(x) = \sum_{i-0}^{\infty} a_i \frac{x^i}{i!}$$

with $a_0 = 0$.

Let $H(x) = \sum_{i=0}^{\infty} h_i \frac{x^i}{i!}$ be the EGF for the number of ways to partition $\{1, \ldots, n\}$ into subsets and carry out Task 1 on each subset.

Set $h_0 = 1$. Then $H(x) = e^{A(x)}$.

Proof. An excellent and accessible discussion of the topic may be found in [2], Theorem 3.27. □

Making cycles   We want to count the number of ways to make the sequence $\{1, \ldots, n\}$ into cycles. This is a combination of:

- the EGF for the number of ways to partition $\{1, \ldots, n\}$ into subsets, AND

- the EGF for the number of ways to make each subset into a cycle.

Let $c(x)$ be the EGF for the number of ways to make the sequence $\{1, \ldots, n\}$ into cycles,

$$c(x) = \sum_{n=1}^{\infty} a_n \frac{x^n}{n!}$$

$$a_n = \text{number of ways to make } n \text{ things into a cycle}$$

which is the number of ways to place n things into n slots ($n!$), without any concept of starting points (so we divide by n, since in a cycle of n elements there are n possible starting points). Therefore,

$$a_n = \frac{n!}{n} = (n-1)!,$$

implying,

$$c(x) = \sum_{n=1}^{\infty} \frac{x^n}{n} = \log\left(\frac{1}{1-x}\right) = -\log(1-x).$$

Consider the EGF for the number of ways to make a sequence into cycles to be Task 1 from Theorem 2, and, letting $H(x)$ be the desired EGF for the number of ways to partition $\{1, \ldots, n\}$ into cycles, we obtain:

$$H(x) = e^{\log\left(\frac{1}{1-x}\right)} = \frac{1}{1-x} = 1 + x + x^2 + \cdots$$

$$= 1 + x + 2!\frac{x^2}{2!} + 3!\frac{x^3}{3!} + \cdots,$$

which implies that there are $n!$ ways to partition $\{1, \ldots, n\}$ into cycles, which is equivalent to the number of ways to permute n things, which is of course equal to $(n!)$.

In this case, the exponential generating function argument was more complicated than the combinatorics of making permutations, but in this way we support the exponential generating function method. In future examples, the derivation of an EGF will, though complicated, be far simpler than any sort of counting argument.

4   Counting Mapping Statistics

Methods in [4] can be used to "mark" nodes of interest in exponential generating functions for ternary functional graphs. These markings can allow us to calculate total mapping statistics for ternary functional graphs with n nodes.

Ternary functional graphs can be constructed as follows:

$$
\begin{array}{lcl}
\text{TernFunGraph} & = & \text{set(Components)} \\
\text{Component} & = & \text{cycle(Node*(TernTree cardinality=2))} \\
\text{TernTree} & = & \text{Node+Node*set(TernTree cardinality=3)} \\
\text{Node} & = & \text{Atomic Unit}
\end{array}
$$

A node is a simple atomic unit. Each ternary tree consists of either a single node or a node that is the root of another ternary tree. A component is a cycle of nodes, where each node is the root of two ternary trees. Ternary functional graphs are sets of components.

The generating functions representing total number of graphs, components, and trees in ternary functional graphs are:

$$
\begin{align}
f(x) &= e^{c(x)} \tag{1} \\
c(x) &= \ln\left(\frac{1}{1 - \frac{x}{2}t(x)^2}\right) \tag{2} \\
t(x) &= x + \frac{1}{3!}t(x)^2 \tag{3}
\end{align}
$$

4.1 Number of Components and Number of Cyclic Nodes

Theorem 3. *The exponential generating functions for the total number of components and total number of cyclic nodes in a ternary functional graph of size n are*

$$\text{Number of components} = \left[\frac{\partial}{\partial u}e^{uc(x)}\right]_{u=1}$$

$$= \frac{1}{2}x^3 + \frac{13}{24}x^6 + \frac{83}{144}x^9 + \frac{355}{576}x^{12} + O\left(x^{15}\right) \qquad \text{(i)}$$

$$\text{Number of cyclic nodes} = \left[\frac{\partial}{\partial u}e^{\ln\frac{1}{1-u\frac{x}{2}t(x)^2}}\right]_{u=1}$$

$$= \frac{1}{2}x^3 + \frac{2}{3}x^6 + \frac{29}{36}x^9 + \frac{17}{18}x^{12} + O\left(x^{15}\right) \qquad \text{(ii)}$$

Proof. (i) The exponential generating function (1) counts the number of ways to partition the nodes in a ternary functional graph into components. Inserting a u marks each component in all of the graphs. The marked generating function results in a Taylor series:

$$1 + \frac{1}{2}ux^3 + \left(\frac{7}{24}u + \frac{1}{8}u^2\right)x^6 + \left(\frac{2}{9}u + \frac{7}{48}u^2 + \frac{1}{48}u^3\right)x^9 + O\left(x^{12}\right).$$

The coefficients of u and u^2 in the term corresponding to $n = 6$ tells us that $\frac{7}{24}\cdot 6!$ possible ternary functional graphs with 6 nodes have 1 component and $\frac{1}{8}\cdot 6!$ have 2 components. Differentiating with respect to u and evaluating at $u = 1$ yields the exponential generating function for the total number of components in all ternary functional graphs of size n.

(ii) Within each component, (2) is an exponential generating function for the number of ways to make a cycle out of ternary trees. Inserting the u marks the root of each tree in the cycle, therefore counting the number of cyclic nodes. Exponentiating this function accounts for the number of possible components. The marked generating function results in a Taylor series:

$$1 + \frac{1}{2}ux^3 + \left(\frac{1}{6}u + \frac{1}{4}u^2\right)x^6 + O\left(x^9\right).$$

Again, the coefficients of u and u^2 in the term corresponding to $n = 6$ tells us $\frac{1}{6}\cdot 6!$ possible ternary functional graphs with 6 nodes have 1 cyclic node and $\frac{1}{4}\cdot 6!$ have a 2 cyclic nodes. Differentiating with respect to u and evaluating at $u = 1$ yields the exponential generating function for the total number of cyclic nodes in all ternary functional graphs of size n. $\square$

In order to calculate average mapping statistics from the total mapping statistics, there needs to be a "normalizer". In fact, for number of components and cycle length, the exponential generating function for the number of graphs acts as this "normalizer":

$$\text{avg. number of components} = \frac{total\ number\ of\ components}{total\ number\ of\ graphs}$$

$$\text{avg. cycle length} = \frac{total\ number\ of\ cyclic\ nodes}{total\ number\ of\ graphs}$$

Now, the theoretical values for average number of components and average cycle length can be computed. They can then be compared with actual values generated from discrete exponentiation functional graphs (see section 6).

4.2 "As Seen from a Node" Mapping Statistics

The double marking process is used to count mapping statistics "as seen from a node". The two markings, u and w mark the statistic of interest and the number of nodes that "see" the count respectively.
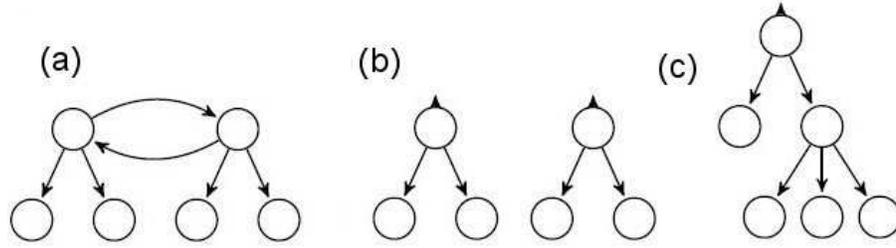

Figure 2: All of the possible ternary functional graphs with 6 nodes. In "as seen from a node" mapping statistics, the count is weighted by the number of nodes that can "see". For example, in (a), all 6 nodes "see" a cycle length of 2. In (b), 3 nodes "see" a cycle length of 1 and another 3 nodes "see" a cycle length of 1. In (c), all 6 nodes "see" a cycle length of 1. The 6th term in the Taylor series for (ii) reflect this.

Theorem 4. *The exponential generating functions for the component size and cycle length as seen from a node in a random ternary functional graph are*

$$
\text{Component size} \quad = \left[\frac{\partial^2}{\partial u \partial w} e^{c(x)} \ln \frac{1}{1 - uw\frac{x}{2}t(uwx)^2} \right]_{u=1,w=1}
$$
$$
= \frac{9}{2}x^3 + \frac{51}{4}x^6 + \frac{201}{8}x^9 + \frac{339}{8}x^{12} + O\left(x^{15}\right) \qquad \text{(iii)}
$$

$$
\text{Cycle length} \quad = \left[\frac{\partial^2}{\partial u \partial w} e^{c(x)} \ln \frac{1}{1 - uw\frac{x}{2}t(wx)^2} \right]_{u=1,w=1}
$$
$$
= \frac{3}{2}x^3 + \frac{13}{4}x^6 + \frac{43}{8}x^9 + \frac{191}{24}x^{12} + O\left(x^{15}\right) \qquad \text{(iv)}
$$

Proof. (iii) The exponential generating function (1) counts the number of ways to partition the nodes in a ternary functional graph into components. For all of the unmarked components, there is a marked component of connected ternary trees:

$$
\ln \frac{1}{1 - uw\frac{x}{2}t(uwx)^2}.
$$

Within the component, u acts as a counter for the size of the component by marking all of the nodes in the tree for each tree. Then, w acts as a counter for the number

11

of nodes total in the component that "see" the component size. The combined marked generating functions result in a Taylor series:

$$\frac{1}{2}u^3w^3x^3 + \left(\frac{7}{24}u^6w^6 + \frac{1}{4}u^3w^3\right)x^6 + O\left(x^9\right).$$

The coefficient of the term corresponding to $n = 3$ tells us that there are $\frac{1}{2} \cdot 3!$ possible ternary functional graphs with 3 nodes, and in all of them the three nodes "see" a component size of three. In the case of $n = 6$, the coefficients mean that $\frac{7}{24} \cdot 6!$ possible ternary functional graph with 6 nodes have 6 nodes that "see" a component size of 6 ((a) and (c) in Figure 2), and $\frac{1}{4} \cdot 6!$ have 3 nodes that see a component size of 3 ((b) in Figure 2). Differentiating with respect to u and w and evaluating at $u = 1$ and $w = 1$ yields the exponential generating function for the component size as seen from a node in all ternary functional graphs of size n.

(iv) The double marking process is similar to the process to construct (iii). For all of the unmarked components, $e^{c(x)}$, there is a marked component of connected ternary trees:

$$\ln\frac{1}{1 - uw\frac{x}{2}t(wx)^2}.$$

Within the component, u acts as a counter for the size of the cycle by marking the number of trees rooted on the cycle. Then, w acts as a counter for the number of nodes total in the component that "see" the cycle length. The marked generating function results in a Taylor series:

$$\frac{1}{2}uw^3x^3 + \left(\frac{1}{6}uw^6 + \frac{1}{8}u^2w^6 + \frac{1}{4}uw^3\right)x^6 + O\left(x^9\right).$$

The coefficient of the term corresponding to $n = 3$ tells us that there are $\frac{1}{2} \cdot 3!$ possible ternary functional graphs with 3 nodes, and in all of them the three nodes "see" a cycle of length 1. In the case of $n = 6$, the coefficients mean that $\frac{1}{6} \cdot 6!$ possible ternary functional graph with 6 nodes have 6 nodes that "see" a cycle of length 1 ((c) in Figure 2), $\frac{1}{8} \cdot 6!$ have 6 nodes that see a cycle of length 2 ((a) in Figure 2), and $\frac{1}{4} \cdot 6!$ have 3 nodes that see a cycle of length of 1 ((b) in Figure 2) . Differentiating with respect to u and w and evaluating at $u = 1$ and $w = 1$ yields the exponential generating function for the cycle length as seen from a node in all ternary functional graphs of size n. □

As in the previous section, in order to calculate average mapping statistics from the total mapping statistics, there needs to be a "normalizer". Unlike the previous case, in "as seen from a node" mapping statistics, the number of graphs and the number of nodes need to be taken into account. The average component size and cycle length are:

$$avg.\ component\ size\quad =\quad \frac{component\ size}{total\ number\ of\ graphs \cdot number\ of\ nodes}$$

$$avg.\ cycle\ length\quad =\quad \frac{cycle\ length}{total\ number\ of\ graphs \cdot number\ of\ nodes}$$

Now, the theoretical values for average cycle length can be computed and compared with actual values generated from discrete exponentiation functional graphs (see section 6).

5 Observed Results

Using a C++ program written by Dan Cloutier in [3], we collected data for the number of ternary functional graphs, average number of components, average number of cyclic

nodes, and average cycle length as seen from a node for nine primes of the form $p = 36k + 1$. Originally when we chose the primes we thought this would be beneficial. We now realize that more ternary DEFGs would have been generated had we chosen primes of the form $p = 3q + 1$ where q is prime (and sufficiently large).

Theorem 5 ([3], Theorem 1). *Let m be any positive integer that divides p-1. Then there are $\phi\left(\frac{p-1}{m}\right)$ m-ary functional graphs produced by the map $x \to \gamma^x \mod p$ for a given g and p. Furthermore, if r is any primitive root modulo p, and $g \equiv r^a \mod p$, then the values of g that produce an m-ary graph are precisely those for which $\gcd(a, p-1) = m$.*

By Theorem 5 each prime of the form $3q+1$, for q prime would generate $q-1$ Ternary Discrete Exponentiation Functional Graphs, and we will use primes of this form when we obtain more data. However, as our sample size was nine primes, the implications that we may draw from our data are limited already by the scope of the project. Ideally, we would like to compute the statistics for a distribution of primes, giving us more general and interesting data. However, these nine primes, though flawed in design, are a good place to start.

6   Theoretical Results

The functional graphs associated with each prime p contain $p-1$ nodes (because we do not include 0 mod p in the functional mapping). The theoretical values for the mapping statistics associated with each prime p are the $p-1^{th}$ term in the exponential generating functions for the average number of components (i), number of cyclic nodes (ii), and cycle length (iv). Computational limits of the program did not allow for the inclusion of (iii).

Note that since the primes are all of the form $36k+1$, we are considering graphs with $36k$ nodes, guaranteeing a nonzero coefficient in the exponential generating functions. This is because ternary functional graphs must contain n nodes, where n is a multiple of 3, in order to maintain $3 - aryness$ and the property that the graph is $functional$. Therefore, any generating function that describes ternary functional graphs can only have nonzero coefficents for x^n, where n is a multiple of 3.

Because of the complexity of the generating functions, we were not able to obtain a closed form for the coefficients of the exponential generating functions for the mapping statistics. In order to obtain the theoretical values for the average mapping statistics, we used Maple to convert the exponential generating functions (i), (ii), and (iv) into differential equations. Then we converted the differential equations to recursive equations that were used to calculate the coefficients for the ternary functional graphs associated with our primes.

Avg. Number of Components				
Prime Number	Number of DEFGs	Theoretical Values	Observed Values	Relative Error
100297	9504	6.09055	6.03188	0.009632
100333	11136	6.09086	6.01419	0.012588
100549	9072	6.09273	6.03616	0.009286
100621	8064	6.09336	6.10379	0.001713
100693	11184	6.09398	6.01967	0.012194
100801	7680	6.09492	6.04922	0.007499
100981	7680	6.09650	6.04089	0.009122
101089	10368	6.09745	6.01987	0.012723
101161	8960	6.09808	6.06641	0.005194

Avg. Number of Cyclic Nodes				
Prime Number	**Number of DEFGs**	**Theoretical Values**	**Observed Values**	**Relative Error**
100297	9504	279.998	281.555	.005559488
100333	11136	280.049	282.662	.009332805
100549	9072	280.351	280.018	.001184713
100621	8064	280.451	279.935	.001839435
100693	11184	280.552	280.444	.000385063
100801	7680	280.702	280.444	.000922161
100981	7680	280.954	280.373	.002066765
101089	10368	281.104	279.932	.004168046
101161	8960	281.204	281.447	.000862035

Avg. Cycle Length (as seen from a node)				
Prime Number	**Number of DEFGs**	**Theoretical Values**	**Observed Values**	**Relative Error**
100297	9504	140.498	139.191	.009298580
100333	11136	140.523	139.432	.007763986
100549	9072	140.674	141.117	.003146833
100621	8064	140.724	140.438	.002033890
100693	11184	140.774	140.215	.003970767
100801	7680	140.850	140.269	.004122784
100981	7680	140.975	141.540	.004007301
101089	10368	141.051	140.958	.000657469
101161	8960	141.101	140.716	.002726134

The theoretical values appear to be very close to the observed values for the DEFGs generated by the primes. The number of ternary DEFGs generated for each prime ranged from 7680 to 11184 graphs. The results for average cycle length had all of the relative errors less than 1%. For average number of components and average number of cyclic nodes, all of the relative errors are less than 2%.

There does not appear to be a correlation between number of DEFGs and relative error in our results. In the data set for the cycle length, the highest error of .9% occured for a prime that generated 9504 graphs, however the second highest error of .1% occured for a prime that generated 11136 graphs. In the same data set, the least two errors of .06% and .2% occured for primes that generated 10368 and 8064 graphs respectively.

The results from the average number of components and average number of cyclic nodes also show that number of DEFGs does not appear to be correlated with relative error.

7   Conclusions

Our observed results showed that for ternary DEFGs, the three mapping statistics (average number of cyclic nodes, average cycle length as seen from a node, and average number of components) behaved very similarly to the average ternary functional graph. This means that, for these three mapping statistics, the discrete logarithm function does not appear to show any characteristic behavior or pattern that could later be exploited.

To strengthen this argument, future research should include prime numbers that generate more ternary DEFGs. This may take more computation time; however it would

give a more general sampling of prime numbers to analyze. Also, future research could include more theoretical exponential generating functions for other mapping statistics such as average tail length, tree size, maximum cycle length, and maximum tail length. In addition, a thorough statistical analysis could be performed on the data to evaluate the significance of the results.

References

[1] *Generatingfunctionology*. Academic Press, University of Pennsylvania, 1994.

[2] Miklos Bona. *Introduction to Enumerative Combinatorics*. McGraw-Hill Science Engineering, 2005.

[3] Dan Cloutier. Mapping the discrete logarithm. Undergraduate thesis, Rose-Hulman Institute of Technology, 2005.

[4] Philip Flajolet and Andrew Odlyzko. Random mapping statistics. *Advances in Cryptology, Proc. Eurocrypt'89*, 1990.