Prime Analysis in Binary

Brandynne Cho
Saint Mary's College of California

September 17th, 2012

"The best number is 73. [...] 73 is the 21st prime number. Its mirror, 37, is the 12th, and its mirror, 21, is the product of multiplying - hang on to your hats - 7 and 3. [...] In binary, 73 is a palindrome: 1001001, which backwards is 1001001."
-Sheldon Cooper, The Big Bang Theory

1 Introduction

Working with prime numbers in binary is of great ease as binary is already the system of counting numbers that computers use. The binary system includes two numbers, 0 and 1. To transition from counting in binary, one simply can add the correct sum of a decimal number, a number in base 10, in the correct place holders of the base 2 system. For example, to create the number 5 in binary, one must place a 1 in the $4 = 2^2$ place holder, a 0 in the $2 = 2^1$ place holder, and a 1 in the $1 = 2^0$ place holder. So $5 = 101_2$.

A relationship is formed when one of the binary digits of a prime can be changed to create another prime [2]. For instance, changing the second digit of $17 = 10001_2$ forms the prime number $19 = 10011_2$. These primes now have a relationship or are "neighbors". $19 = 10011_2$ also has the neighbor $23 = 10111_2$. Again, $23 = 10111_2$ connects to $31 = 11111_2$, which then connects to $29 = 11101_2$. These primes form a neighborhood or "cluster" of at least five primes. It seems that most primes have neighbors, and lie in clusters. For some primes, such as $11 = 1011_2$ and $127 = 1111111_2$, a one digit change cannot be made to form another prime. We call these "isolated" primes.

For the primes that are not isolated, their relationships can be tracked on a graph. This graph, Figure 1, does not include all primes, but many of them. The graph is created through connections of a prime's neighbors and the neighbors that the prime's neighbors have. For instance, 5, which is the minimum (i.e. smallest) prime for this cluster [1], has neighbors 7 and 13. 7 has no other neighbors besides 5, so we must move on to 13. 13 has another neighbor, 29, and another connection is made. Next, 29's other neighbors are 31 and 61. We've already seen 31 earlier, so 61 can be inspected. This process continues and forms our very large main graph.

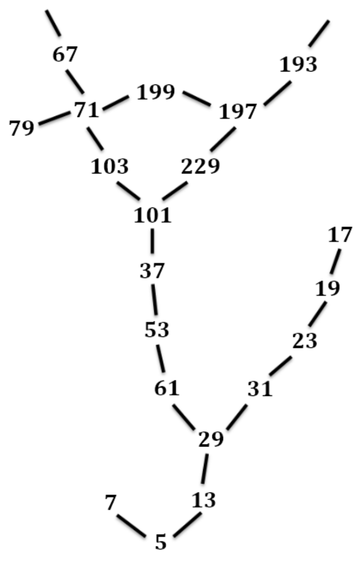

Figure 1: The beginning of the main graph.

2 Does the graph contain all primes?

The graph in Figure 1 does not contain all of the primes. This was described first by Paulsen [2]. For a prime p suppose that its binary representation is $\sum_{i=0}^{n} e_i 2^i$, where the e_i's are 0 or 1. Because $2 \equiv -1 \pmod 3$, we have

$$p \equiv \sum_{i=0}^{n} e_i (-1)^i \pmod 3.$$

Because $p > 3$ is prime it is not divisible by 3, and so this value is not 0. This suggest examining the value $\delta(p) = \sum_{i=0}^{n} e_i(-1)^i$, which is not a multiple of 3.

Now suppose q is a neighbor of p. Then $q = p \pm 2^j$ for some j, and hence $\delta(q) = \delta(p) \pm (-1)^j$. Because neither value is divisible by 3, we see there is some k so that $\delta(p), \delta(q) = 3k+1, 3k+2$ in some order.

Therefore, write $D(p)$ for the value k such that $\delta(p) = 3k+1$ or $3k+2$. We have shown that if p and q are neighbors, then $D(p) = D(q)$. Conversely, if $D(p) \neq D(q)$ it is impossible for p and q to be neighbors.

As an example, consider $p = 37$ and $q = 41$. Since $37 = 100101_2$, $\delta(37) = 1 = 0*3+1$, so $D(37) = 0$. Thus 37 can only be neighbors with primes of D-value 0. On the other hand, because $41 = 101001_2$, $\delta(41) = -1 = (-1)*3+2$, so $D(41) = -1$, it means that 41 cannot be neighbors with any primes of D-value 0. In particular, it is impossible to wander through the cluster of primes containing 37 and arrive at 41 – it must be in a separate cluster. For D-value 1, a few primes are 853, 1093, and 3541. D-value 2: 70981 and 120277. D-value -2: 2699 and 35363.

If we think of a large prime p in binary as consisting of leading and tailing 1's, with the other binary digits "randomly" distributed between 0's and 1's, then then there are $n-2$ choices of 0's and 1's, where n is the number of digits in p, i.e. its "digit length". Looking at $\delta(p)$, on average we would expect the 1's to fall into spots contributing $+1$'s about as often as into those contributing -1. That is, we'd expect $\delta(p)$, and so $D(p)$, to be near 0. On the other hand, to have a prime with large D, there will have to be many 1's and few -1's. This will be less common. Similarly it is less common to find primes with large negative D-values. We see this in the following chart. Notice D-values do appear to follow the normal distribution, however, there is a slight skew (Figure 2).

3 Is the graph infinite?

Since the graph is quite large and a majority of primes fall in the D-value 0 range, for this paper the focus of finding if the graph is infinite centers on the D-value 0 primes.

When thinking of numbers in their infinite capacity, it is important to

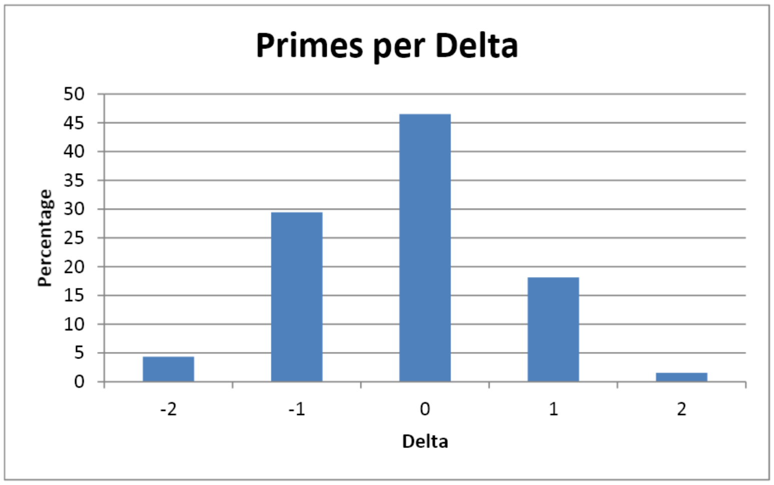

Figure 2: Percentages of primes per Delta (-2, 2) up to digit length 25.

realize infinity's capacity, or lack thereof. As a child one is taught to think of number on a number line. In reality, the number of numbers is infinite. Numbers are much more like stars which stretch forever, extending an eternalness of light years away. As large as this seems, there are still multiple ways to view the infinite graph of prime numbers.

One way to think of an infinite graph is by means of a straight line up, aiming to find the biggest prime. Measurement is taken by steps, where one step consists of going from one prime to its largest neighbor. This view of infinite prime numbers can move very quickly. Table 1 gives data showing the number of steps taken in the graph beginning at 5 until a prime larger than maximum $p = 2^b + 1$ is found. The number of steps travelled to reach the maximum is incredibly quick.

$b, p = 2^b + 1$	base 10 equivalence	number of steps
20	1 million	77
30	1 billion	233
50	quadrillion (15 digits)	607
100	nonillion (30 digits)	2843

Table 1: Total steps for the fastest path up to the maximum prime, $p = 2^b+1$.

This set of data is very significant. The extremely quick paths show a very direct path to huge numbers. The few steps taken to reach such large numbers suggests that the graph goes on forever and, hence, is infinite.

(This data and all other data sets were collected from computer programs written in the computer language Python. For code for the data in Table 1, see PathfinderFAST.py in Appendix.)

3.1 What is the shape of the graph?

An alternate way to think of an infinite graph is by keeping track of all the connecting primes in a big, tall and wide tree-like graph. This is similar to scooping up or tallying the primes desired.

The shape of the main graph appears to be more like the shape of a tree that expands the further it grows upward. The further the graph travels toward infinity, the more interconnected and complex it becomes. Figure 3 depicts such.

3.2 Are there loops?

There are many loops in the graph. The first loop in the main graph begins at 101 and splits to 103 and 229, where 71 and 199 follow 103 and 197 and 199 stem from 229. This loop, of six steps, is small, but the graph has many complex loops of greater sizes; as one loop is discovered, its discovery leads to more loops interconnecting with past loops. Figure 4 shows the loops in the graph for primes less than 1000. It is interesting to note that the steps taken to get around the loops will always be even. This is because for every step away from the base number a 0 is changed to a 1, or a 1 to a 0, in the binary form of a number. The step taken must eventually be reversed to get back. Since each step has an opposite, the total number of steps is even.

Outside of the main graph there are isolated primes, pairs, triples, quads, etc. Some of these excluded primes form loops themselves, but nonetheless do not contribute to loops in the main graph. Figure 5 samples a few isolated primes, a pair, and a quad.

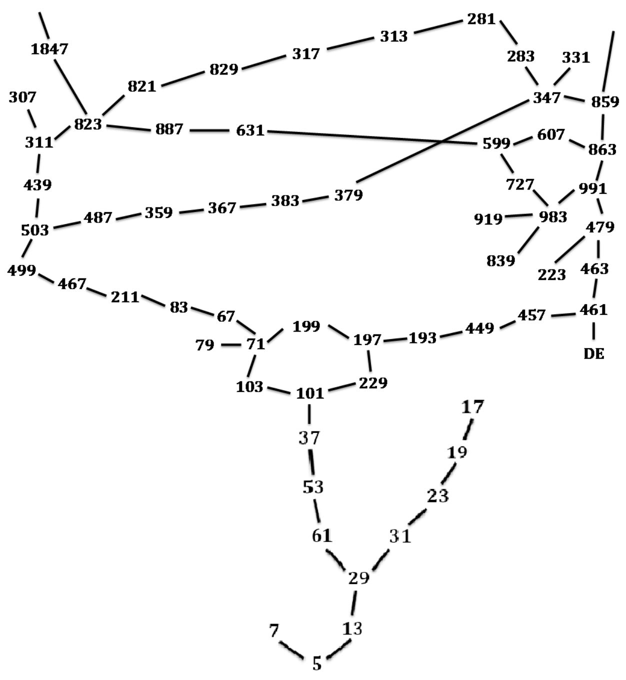

Figure 3: Tree-like shape of the graph which grows in complexity. (For most primes less than 1000.)

3.3 How many neighbors does each prime have?

When trying to determine whether the graph is finite or infinite, it is critical to consider the number of neighbors each prime has. In an infinite segment of the graph, there must be a series of primes all of whom have at least two neighbors each. If most primes have fewer than two neighbors, one would be more likely to think that the graph is finite and consists mostly of isolated clusters. Thus, if the graph is to be infinite, one expects primes to have, on average, at least two neighbors.

Theoretically the number of neighbors suggests that the graph is infinite. The Prime Number Theorem [2] states that the number of primes smaller than N is approximately $\frac{N}{\ln N}$. So the fraction of numbers near N that are prime is approximately $\frac{1}{\ln N}$. Disregarding numbers that are divisible by 2 or 3 or both, there are then $N - \frac{N}{2} - \frac{N}{3} + \frac{N}{6} = \frac{N}{3}$ numbers left. (It is necessary to re-add the multiples of 6 that were thrown away twice.) This mean that if random integers $1 - N$ are chosen, $\frac{2}{3}$ of the time a multiple of 2 and/or 3

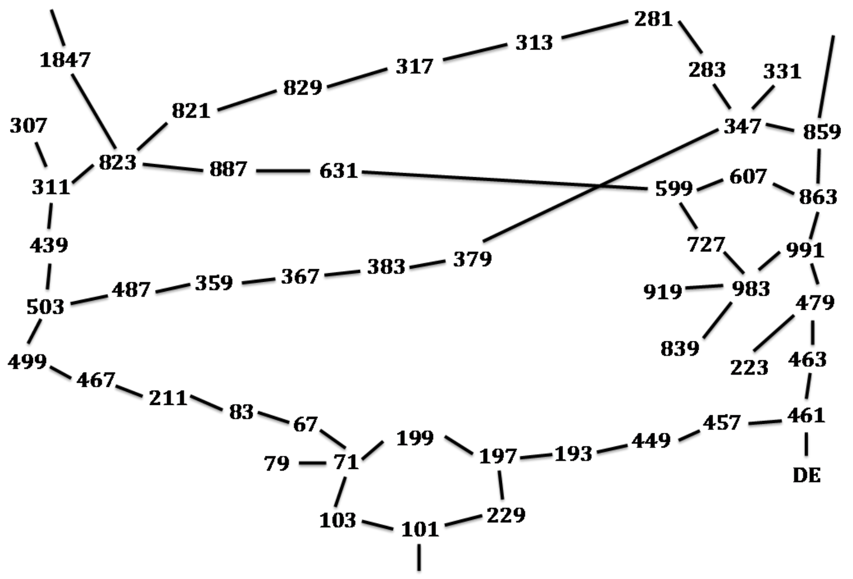

Figure 4: The main graph from 101 until 859 and 1847. DE denotes a Dead End where the graph continues but eventually ends without contribution to loops.

will emerge. So, only picking odd non-multiples of 3 will increase the odds of choosing a prime by a factor of 3.

In essence, it can be interpreted that a number randomly chosen near N that is an odd non-multiple of 3 has a $\frac{3}{\ln N}$ probability of being prime.

Now, fix a prime P with n binary digits. $P \sim 2^n$, where 2^n will be like the N above. P has n potential neighbors: $Q = P \pm 2^i for 1 \leq i \leq n+1$.

About half of these numbers are not prime. Look at $mod3$. P is either 1 or 2 $\mod 3$. For this argument, assume $P = 1 \mod 3$.

When i is even (for 2^i to be $1 \pmod 3$) then $P - 2^i$ is divisible by 3. The 1 in the $i-th$ spot of P cannot be changed to a 0. Thus, no even i spot may change from 1 to 0.

Likewise, a spot with odd i cannot be changed from 0 to 1.

Because we may assume the digits of P (except the leading and trailing 1) are randomly distributed among 1's and 0's, any particular spot has a 50-50 chance of having a 1 or a 0. The choice will be 'unchangeable' half the time. So, there is not really $n-1$ possible neighbors (with $1 \leq i \leq n+1$)

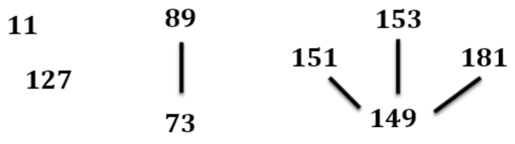

Figure 5: The graph for $D = 0$ does include "Loaners" not in the main graph, consisting of isolated primes, pairs, quads, etc.

but half that many. Adding up the number of primes gives:

$$(number\,of\,neighbors)(probability\,of\,primeness) =$$

$$(\frac{n-1}{2})(\frac{3}{\ln 2^n}) = 3(\frac{n-1}{2(n)(\ln 2)}).$$

The 2^n and 2^{n+1} terms are still left.

For 2^{n+1}: Only $0 \to 1$ is allowed. Like above, half the time a multiple of 3 appears, an automatic non-prime. So on average the number of neighboring primes is:

$$(\frac{1}{2})(\frac{3}{\ln 2^{n+1}}) = \frac{3}{2(\ln 2)(n+1)}.$$

For 2^n: Only $1 \to 0$ is allowed. Again, like above, half the time this leads to an automatic multiple of 3. Moreover, about half the time there is a 0 in the $i^n - 1$ place. Then, the average number of neighbors primes is:

$$(\frac{1}{2})(\frac{1}{2})(\frac{3}{\ln 2^{n-1}}) = \frac{3}{4(\ln 2)(n-1)}.$$

Thus, we end up with:

$$(\frac{3}{2(\ln 2)})(1 - \frac{1}{n} + \frac{1}{n+1} + \frac{1}{2n-2}).$$

This looks a lot like $\frac{3}{2(\ln 2)}$, which is approximately 2.16. This value is, indeed, larger than 2 and suggests that the graph is increasing as the binary digit length n grows.

Contrastingly, we calculated the average number of neighbors for primes with D-value 0 for digit length 5 up to digit length 200. The data in Figure 6 shows the average number of neighbors declining from 2 at digit length 5 to about 1.90 at digit length 200. This data suggests that the graph is finite because, while the average number of neighbors is less than 2, the chances for dead ends increases. (See Python code avgnebrsBIG.py in Appendix.)

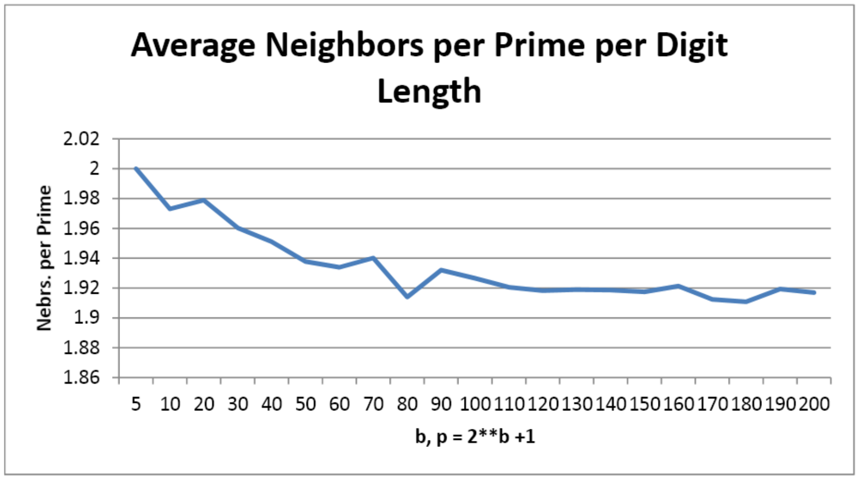

Figure 6: The average number of neighbors per prime per digit length for primes with digit length 5 to 200.

It is possible that the error in the number of neighbors comes from the different deltas. Most data focused on is according to $D = 0$, therefore this inconsistency could be related to lack of inclusion of other deltas. However, it is more difficult to reach other deltas as their graphs begin at greater numbers, corresponding to the more extreme deltas away from 0.

3.4 What is the graph made up of?

The graph is made up of clusters that vary in size. The cluster sizes range from isolated primes, pairs, other smaller groups, up to a large "mega-cluster". This mega-cluster begins at 5 and continues on from prime to prime. It appears to be very large; its expanse is undetermined.

The isolated and pairs seem to make up about 20 percent of all primes; 15 percent and 5 percent, respectively. The mega-cluster is a very large portion of the graph and appears to make up at least 75 percent of primes. This is a significant chunk of numbers. Table 2 depicts this data.

Cluster Size and Percentage of Graph			
$b, p = 2^b + 1$	30	50	100
1	13.07	14.93	15.91
2	3.81	4.29	4.09
3	1.96	1.81	1.64
$\vdots$	$\vdots$	$\vdots$	$\vdots$
2500	78.96	76.96	76.36

Table 2: Cluster sizes and percentages for primes with digit length 30, 50, and 100 for $D = 0$. See clusteravg.py in Appendix for Python Syntax.

The distribution of the mega cluster is interesting to note. (See Figure 7.) Within the 75 percent of primes in the mega-cluster, almost 90 percent of primes have either one, two, or three neighbors. Specifically, 25.77 percent have one neighbor, 38.03 percent have two neighbors, and 24.74 percent have three neighbors. The remaining 11 percent are made up of primes that contain more than three neighbors. (4, 5, 6, and 7 neighbors per prime for this set of data.) It is critical to have enough neighbors to avoid dead ends. In the mega-cluster, since 74.23 percent of primes have more than one neighbor, we expect the branching of the cluster to continue towards infinity.

The program for the cluster size and percentage of the graph was also run for other delta values -2, -1, 1, and 2 (respectively, Table 3,Table 4, Table 5, and Table 6). The data varied per digit length, but was similar to the D-value 0 case, with approximately 20 percent of the graph was made up of isolated and pairs and approximately 75 percent of the graph was made up of the delta's mega-cluster. (Again, clusteravg.py in Appendix.)

Cluster Size and Percentage of Graph			
$b, p = 2^b + 1$	30	50	100
1	20.24	17.52	15.58
2	3.42	3.92	4.02
3	2.9	2.04	1.86
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1500	69.4	73.98	76.3

Table 3: $D = -2$

Cluster Size and Percentage of Graph			
$b, p = 2^b + 1$	30	50	100
1	13.6	15.26	15.34
2	3.84	4.32	4.28
3	2.22	1.6	2.04
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1500	77.6	76.76	76.26

Table 4: $D = -1$

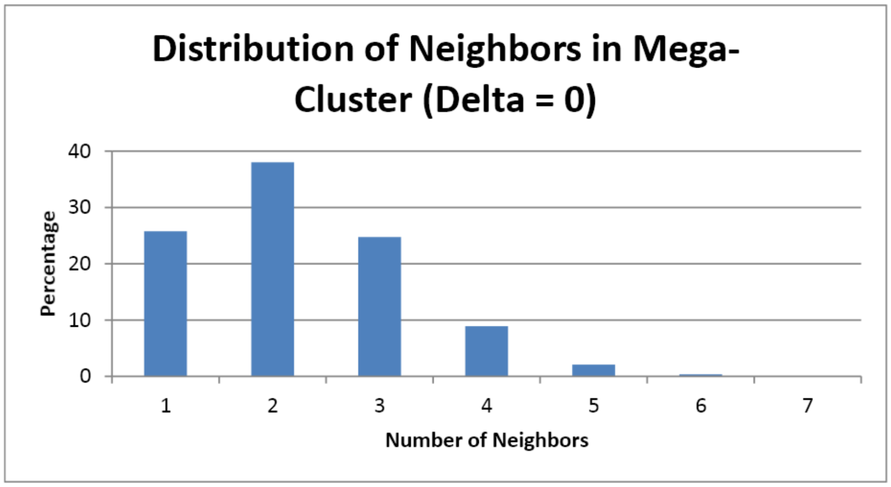

Figure 7: In the mega-cluster, the distribution of the number of neighbors per prime. For Python code, see nebrdistribution.py in Appendix.

Cluster Size and Percentage of Graph			
$b, p = 2^b + 1$	30	50	100
1	17.44	17.02	17.06
2	3.22	4.1	4.38
3	1.66	1.76	2.02
⋮	⋮	⋮	⋮
1500	74.58	74.34	74.36

Table 5: $D = 1$

Cluster Size and Percentage of Graph			
$b, p = 2^b + 1$	30	50	100
1	25.86	18.48	16.02
2	3.06	3.72	4.4
3	2.98	2.22	2.2
⋮	⋮	⋮	⋮
1500	62.94	72.92	75.2

Table 6: $D = 2$

4 Other Findings

There are a few other findings concerning the graph as a whole.

An approximate function to determine the number of isolated primes per digit length has been found. Equation 1 approximates the growth of the number of isolated primes by digit lenth n.

$$I(n) = \begin{cases} \frac{1.91^n}{122} & \text{while n is odd} \\ \frac{1.91^n}{103} & \text{while n is even} \end{cases} \tag{1}$$

Another function that determines the decay of the clusters of primes is noted. The decay begins with number of isolated primes and decreases until the largest group(s). (Generally there is only one large group.) If the number of isolated primes is known, Equation 2 will approximate the decay of the number of other groups, where k comes from the previous number of groups. However, for closer approximations this function changes based on the digit length. Figure 8 shows the values of k when the digit length is 15 to 23. (15 is where regularity for k values begin.)

$$D(k) = k^{0.9} \tag{2}$$

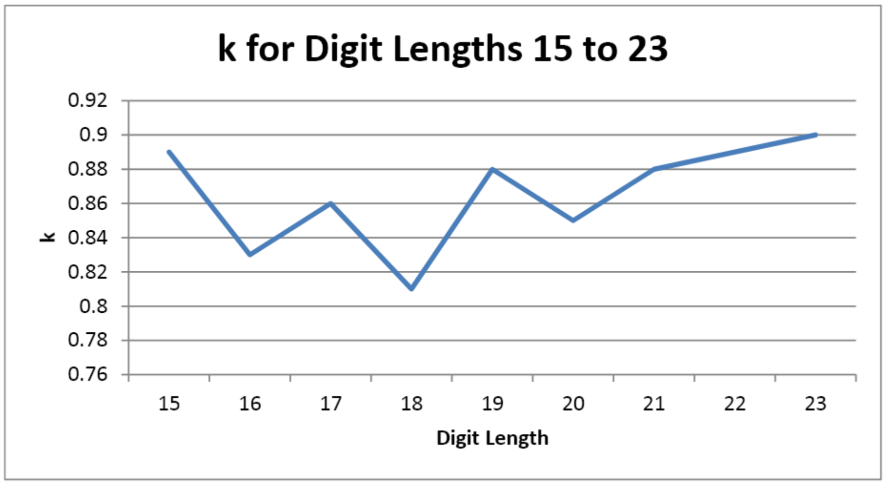

Figure 8: k values to approximate decay.

Tracking the relationships of the graph by putting the values into a matrix allows for finding the rank of the graph. Up to digit length 23 (about 8.4 million in base 10), the rank of the matrix is approximately 82 percent.

5 Concluding Thoughts

Exploring prime numbers in binary has proved to be fascinating. Primes not only are separated into distinct Delta areas, but within each Delta contain intriguing clusters. These clusters are a means of determining if the graph is infinite, through both the directness and shape. We can also test the number of neighbors and quality of loops in the main mega-cluster.

Despite what has been elaborated on and discovered concerning prime numbers, there are still more questions that have arisen regarding the clusters.

1. Exactly how big is the mega cluster?
2. Is the mega cluster really multiple large clusters?

It is hoped that in the future, that these questions will be answered.

Acknowledgements

Thank you to the School of Science at Saint Mary's College of California for the support and funding of the summer research program.

I would also like to thank Dr. Jim Sauerberg for his dedication and guidance through this research.

References

[1] Hartley, M. I. (2002). Partitions in the Prime Number Maze. *Acta Arithmetica, 105*(3), 227-38.

[2] Paulsen, W. (2000). The Prime Number Maze. *The Fibonacci Quarterly, 40*(3), 272-79.

Appendix: Python Code

PathfinderFAST.py (Table 1)

```
from functionfile import *

def main():
        p=5
        path=[ ]
        directions = [ ]
        pseen = 0
        variable = [0]*30

        while p < 200000000000:
                nbrs, numnebrs = nebrs(p)
                variable[numnebrs]+=1
                nbrs = remove(nbrs, path, pseen)
                if len(nbrs)>0:
                        path.append(p)
                        directions.append(nbrs[:-1])
                        p = nbrs[-1]
                else:
                        pseen += 1
                        while len(directions[-1])==0:
                                pseen += 1
                                path=path[:-1]
                                directions = directions[:-1]
                        if len(path)==1:
                                print("DIED")
                        p = directions[-1][-1]
                        directions[-1]=directions[-1][:-1]
        print(p, log(p, 2))
        print(len(path))
        totall = 0
        for a in directions:
                totall += len(a)
        print(totall/len(directions))
        print(variable)
```

```
main()
```

avgnebrsBIG.py (Figure 6)

```
from functionfile import *

def main():
        printfile = open(”avenbers.txt”, ”w”)
        for d in range(0, 1):
                howmany = 100000
                print(”DELTA =”, d, ”Average number of nbrs for”, howmany,”,
primes”, file = printfile)
                for b in range(20, 202,10):
                        totalnebrs = 0
                        totalP = 0
                        totaliso = 0
                        maxnebrs = 0

                        P=2**b+1
                        while(totalP<howmany):
                                if delta(P, d) == d and is-prime(P):
                                        listt, numnebrs = nebrs(P)
                                        totalnebrs += numnebrs
                                        totalP += 1

                                P+=2

                        print(”Nebrs per first” ,howmany,” primes, start at 2**b+1,
b=”, b,” ave=”,totalnebrs / totalP, totalP, file = printfile)
                printfile.close()

main()
```

clusteravg.py (Tables 2, 3, 4, 5, and 6.)

```
from functionfile import *
```

```
def main():
        b = 30 #or 50 or 100
        d = 0 #fix delta
        cap = 2500
        maxx = 7500
        ccount = 0 #fix total clusters found
        avgnebrsclusters = [0]*(cap + 1)
        nclusters = [0]*(cap + 1)

        printfile = open("clusterdata"+str(b)+".txt", "w")
        print("b =", b, "cap =", cap, "maxx =", maxx, file = printfile)

        while ccount < maxx:
                #find p
                p = findnextprime(b)
                while delta(p, d) != d:
                        p = findnextprime(b)

                #find p's clusters
                n, a = clusters(p, cap)

                #average number neighbors
                avgnebrsclusters[n] = (avgnebrsclusters[n]*nclusters[n] + (a/n))
/ (nclusters[n] +1)
                #cluster size
                nclusters[n] += 1

                ccount += 1

        for i in range (1, len(nclusters)):
                if nclusters[i] > 0:
                        print(i, round(avgnebrsclusters[i], 2), round(nclusters[i]/
maxx*100, 2), round(nclusters[i]/i, 2), file = printfile)
        printfile.close()
        print("Done.")

main()
```

nebrdistribution.py (Figure 7)

```
from functionfile import *

def main():
        """This program shows the distribution of the number of nebrs
                for primes in the mega cluster"""

        d = 0
        p = 5
        cap = 1500
        maxx = 100000
        nwnnbrs = [0]*100

        while p < maxx:
                if delta(p, d) == d and is-prime(p):
                        n, a = clusters(p, cap)
                        if n == cap:
                                listt, numnebrs = nebrs(p)
                                nwnnbrs[numnebrs] += 1
                p += 2
                while not is-prime(p):
                        p += 2

        print(nwnnbrs)

main()
```

functionfile.py: This file contains all functions that main codes call on.

```
from math import log
from random import *

SMALL = [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 ]

LIST = [2, 3, 5, 7, 11]
```

```
def is-prime(n):
        SMALL = [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 ]

        LIST = [2, 3, 5, 7, 11]

        """Miller-Rabin primality test. Strong PProbable Prime.
If n < 2,152,302,898,747 is a 2, 3, 5, 7 and 11-SPRP, then n is prime [Jaeschke93]."""

        #assert that n is a positive, and not too large.
        if n == 1:
                return False
        assert n >= 2
        if n > 2152302898747 :
                LIST = SMALL[:20]

        # special case: small primes
        if n in SMALL:
                return True

        #check to see if divisible by small prime
        for p in SMALL:
                if n % p == 0:
                        return False

        #Begin MR TEST

        # write n-1 as 2**s * d
        # repeatedly try to divide n-1 by 2
        s = 0
        d = n-1
        while d % 2 == 0:
                d >>= 1
                s += 1
        assert(2**s * d == n-1)
```

```
    # test the base a to see whether it is a witness for the compositeness of n
    def try-composite(a):
        a-to-power=pow(a,d,n)
        if a-to-power == 1: #pow uses binary exponentiation
            return False
        for i in range(s-1):
            if a-to-power == n-1:
                return False
            a-to-power = (a-to-power * a-to-power) % n
        return not (a-to-power == n - 1)

        # return True # n is definitely composite

    for a in LIST:
        if try-composite(a):
            return False

    return True # no base tested showed n as composite

def delta(n, d):
    summ = 0
    sign = 1
    while(n > 0):
        summ += (n % 2) * sign
        n = n >> 1 #n = n // 2
        sign = sign * -1
    if d == summ // 3:
        return(summ // 3)

def nebrs(P):
    listt = [ ]
    L = int(log(P, 2))
    for k in range (1, L):
        Q = P  2**k
        if is-prime(Q):
            listt.append(Q)
    if P >> L-1 == 3:
        Q = P2**L
```

```
                if is-prime(Q):
                        listt.append(Q)
        Q = P̂(2**(L+1))
        if is-prime(Q):
                listt.append(Q)

        listt.sort()

        numnebrs = 0
        numnebrs = len(listt)

        return listt, numnebrs

def finder(p, d, maxx, gamma):
        nebrslist, numnebrs = nebrs(p)
        nebrslist.append(p)
        nebrslist.sort()

        pset = set(nebrslist)

        gamma2 = [ ]
        for s in gamma:
                if pset.intersection(s):
                        pset = pset.union(s)
                else:
                        gamma2.append(s)

        gamma2.append(pset)
        gamma = gamma2
        return gamma

def setcounter(d, gamma, printfile):
        countt = [ ]
        maxxc = 0
        smallone = 0
        for s in gamma:
                lenn = len(s)
```

```
            countt.append(lenn)
            if lenn > maxxc:
                maxxc = lenn
                smallone = min(list(s))
        maxxc = max(countt)
        for i in range(1, maxxc+1):
            if i in countt:
                print(countt.count(i), ”group(s) of”, i)
                print(countt.count(i), ”group(s) of”, i, file = printfile)
        return maxxc, smallone

def rank-counter(p, lead):
        plist, numnebrs = nebrs(p)
        if plist:
            first = plist[0]
            i = 0
            while i < len(lead):
                if lead[i][0] == first:
                    for j in lead[i]:
                        if j in plist:
                            plist.remove(j)
                        else:
                            plist.append(j)
                            plist.sort()
                            i = 0
                            first = plist[0]
                else:
                    i += 1
                if plist:
                    first = plist[0]
                else:
                    i = len(lead)
            if plist:
                lead.append(plist)
                return 1
            else:
                return 0
        else:
```

```
        return 0

def ranker(plist, n):
    for i in plist:
        if i == 3:
            nprimes = 1
            rank = 1
            lead = [ [7] ]
            plist.remove(i)
        else:
            nprimes = 0
            rank = 0
            lead = [ ]
    for p in plist:
        rank += rank-counter(p, lead)
    for j in lead:
        if j[0] < 2**(n-1):
            lead.remove(j)
    nprimes += len(plist)
    return rank, nprimes

def smallestp(p, d, dsofar, printfile):
    if d not in dsofar:
        print("The smallest prime for delta =", d, "is", p,"n", file = printfile)
        dsofar.append(d)

def counter(P, d, totalnebrs, totalP, totaliso, maxnebrs, maxwho, NpP):
    listt, numnebrs = nebrs(P)
    totalnebrs += numnebrs
    totalP += 1
    if numnebrs >= maxnebrs:
        if maxnebrs == numnebrs:
            maxwho.append(P)
        else:
            maxnebrs = numnebrs
            maxwho = [ ]
            maxwho.insert(0, P)
    if listt == [ ]:
```

```
                totaliso += 1
        NpP = totalnebrs / totalP

        return totalnebrs, totalP, totaliso, maxnebrs, maxwho, NpP

def rankofsets(gamma, minn, maxx, ranksum, ranknum, rankwho, printfile):
        for s in gamma:
                s = list(s)
                rank, nprimes = ranker(s, maxx)
                if rank <= minn:
                        minn = rank
                if rank >= maxx:
                        maxx = rank
                        rankwho = len(s)

                ranksum += rank
                ranknum += 1

        return minn, maxx, ranksum, ranknum, rankwho

def findnextprime(n):
        p=randbin(n)
        while not is-prime(p):
                p +=2
        return p

def randbin(n):
        a=1
        for i in range(n-2):
                a=a*2+randint(0,1)
        a=2*a+1
        return a

def remove(nbrs, path):
        newnbrs=[ ]
        for n in nbrs:
                if not(n in path):
                        newnbrs.append(n)
        return newnbrs

def clusters(p, cap):
        """ clusters takes as imput a prime number. It outputs the size of p's cluster,
and the total number of neighbors of all the primes in p's cluster. """


        nbrs, nn = nebrs(p)

        path = [ ]
        directions =[ ]

        numprimes = 1
        numnbrs = len(nbrs)

        while not (path == [ ] and nbrs ==[ ]):

                if nbrs == [ ]:
                        p = path[-1]
                        path=path[:-1]
                        nbrs = directions [-1]
                        directions = directions[:-1]

                else:
                        path.append(p)
                        directions.append(nbrs[:-1])


                        p = nbrs[-1]

                        numprimes +=1
                        nbrs, nn = nebrs(p)
                        numnbrs += len(nbrs)
                        nbrs = remove(nbrs, path)
                        if numprimes >= cap:
                                return numprimes, numnbrs
        return numprimes, numnbrs
```