MACQUARIE UNIVERSITY - COMPUTING DEPARTMENT
MASTER OF RESEARCH

Functional Graphs of Polynomials and their Twist over Finite Fields

Jeffrey Smith

Supervisor: Professor Bernard Mans

June 16, 2019

Recently, functional graphs, i.e., graphs generated from a polynomial function over Finite Fields F_p, have received renewed interest, due to their applications in Computer Science. In this thesis, we study new functional graphs of degree two generated by a polynomial and its twist.

We perform a computational analyses of these functional graphs by developing new algorithms, and optimizing their implementation performances (including multi-threading). Our results show that (i) most graphs are strongly connected or have only two components (including one giant component and one small component of two or three vertices); (ii) every connected component of the graph have many Hamiltonian cycles (growing exponentially with the finite field); (iii) these Hamiltonian cycles can be used to construct balancing binary sequences. Also the Hamiltonian property makes these graphs distinct to well-known random mappings where the expected cycle length is about $\sqrt{p}$. These experimental results were used to guide several theoretical analysis and then compared with the relevant mathematical proofs.

This work has not been submitted for a higher degree to any other university or other institution.

JEFFREY SMITH

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

Randomness is an important concept in computer science as it is required as source of random numbers or random decisions in many fields such as statistical analysis using a Monte-Carlo simulation or in cryptography where there is a requirement for a random seed used to generate cryptographic hashes. There is an inherent problem with computer generated random numbers in that number generating algorithms are deterministic in nature where at best a complicated analogue to randomness can be used, referred to as pseudo-randomness. Generation of pseudo-random sequences is only practical if they are unpredictable, so algorithms that produce an unpredictable sequence of pseudo-random numbers that are similar to a truly random process have been an area of study for some time.

One such method of pseudo-random number generation is the concept of a random walk, where some transformation function $f(x)$ is used to dictate the next step along an iterative path from x. In Equation (1.1) this process is illustrated where for an element x in a finite field F_q of q elements, the next value is determined by some function $f(x)$, which is in turn used as the input for the next step. This mapping can be visualized as a functional graph where each edge is the transformation of $x \rightarrow f(x)$ between elements of F_q as shown in Figure 1.1. In this thesis, we will only focus on the case where q = p prime.

$$\text{Given } x_0: \quad x_1 = f(x_0),\ x_2 = f(x_1),\ ...,\ x_q = f(x_{q-1}) \tag{1.1}$$

The choice of generator function $f(x)$ for these random walks has been experimented with for several decades, and there have been some notable number theoretic functions examined in this way, such as the Collatz conjecture defined by the generator function ($f(n) = n/2$ if $n \equiv 0 \mod 2$, $f(n) = 3n+1$ if $n \equiv 1 \mod 2$) , discussed in [Lagarias, 1985] as one of many research papers, where computing resources have been directed at processing this function for large input values of n, research is yet to find a positive integer that does not end its random walk in a cycle of $4 \rightarrow 2 \rightarrow 1 \rightarrow 4$. Similarly, theoretical analysis has attempted to prove this conjecture but there remains no theoretical proof that there is or is not a positive integer for this mapping that does not eventually reach a value of 1.

As more powerful computing has become available the ability to test theoretical conjectures with experimental analysis has become a tool for number theoretic analysis that was

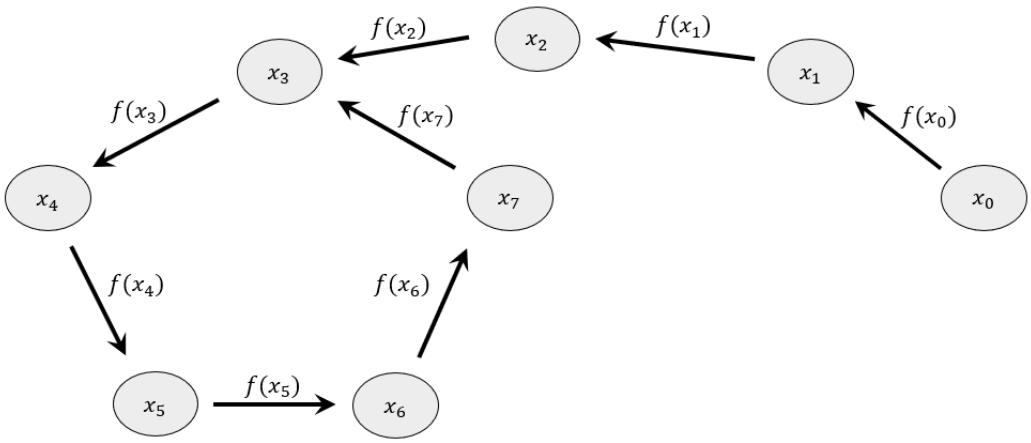

Figure 1.1: Random walk visualization as a functional graph.

previously out of reach. Even without an exact proof it may be possible to empirically investigate these problems to gain some insight. This modern approach to have computational and theoretical analysis to work together has become a valuable tool for the analysis of number theoretic functions. In this way, both approaches are enhanced as theoretical expectations guide computational analysis for investigation. Conversely, computational findings demonstrate expected, uncovering unexpected results that encourage further theoretical analysis.

Flajolet and Odlyzko [Flajolet and Odlyzko, 1990] discuss analysis of random mappings and how computationally driven pseudo-random mappings compare with truly random mappings. Focusing on using random walks as pseudo-random number generators that map a field of positive integers onto itself, a framework for analyzing random mappings is introduced and the analysis conducted in [MacFie and Panario, 2012] shows the application of this framework. A combinatorial construction of the mapping is developed and any asymptotic behavior investigated. Second, they performed a singularity analysis over the construction to identify interesting properties and exceptions to the established general case. The mappings are analyzed as directed graphs as shown in Figure 1.1 where visualization aids investigation of specific combinations of variables, for small finite fields.

More recently, similar analysis method has been used to investigate the use of small degree polynomials over finite fields as the generator functions of pseudo-random mappings. These studies have discovered the idiosyncrasies of each polynomial studied such as length of tails and cycles as well as interesting features, and a measure of similarity to truly random mappings. Useful result such as the prevalence of squared-size loops and many partitions

which limit the usefulness of the mappings due to splitting the field into smaller sets of elements with which to generate pseudo-random sequences.

Investigating untested functions to assess the properties of their functional graphs creates the possibility of finding a more suitable mapping for applications such as random generators. Building on the results of previous studies, the generator function was altered to improve the connectivity of the resulting functional graphs. The new generator is expected to produce new characteristics in the pseudo-random mappings that are a more suitable source of pseudo-random sequences. If successful, an improved generator function that provides consistent, hard to predict pseudo-random sequences could be developed and applied to such fields as cryptographic protocols, random seed generation and contribute to broadening the pool of knowledge around number theoretic functions more generally.

1.2 Functional Graphs of Polynomials

Using random walks of polynomial functions as generators of random sequences has been applied as unpredictable pseudo-random number generators [Blum et al., 1986] and algorithms for factorization [Pollard, 1975, Brent, 1980], both of which have application for cryptography.

In the case of pseudo-random number generators, the $f(x) = x^2 \mod N$ generator function (where N is a product of distinct primes both congruent to 3 mod 4) was compared with the $f(x) = 1/P$ generator function (where P is some prime). While both generators produced well mixed sequences of pseudo-random numbers, the $1/P$ generator was demonstrated to be predictable with only limited knowledge of the generated sequences, where as the $x^2 \mod N$ generator had no simple way to predict other elements in generated binary sequences. This illustrated the need for cryptographically strong pseudo-random number generators which are polynomial time unpredictable, in that there are no polynomial time algorithms that allow prediction of generated sequences [Blum et al., 1986].

Factorization algorithms for prime factors [Pollard, 1975] using a Monte Carlo method apply the generator function $f(x) = x^2 - 1 \mod p$ ($p \in P$) and used the random walk generated to provide a sequence of pseudo-random numbers to iteratively test for prime factors. The average number of iterative steps that this method takes to find a factor is of order of $\sqrt{p}$ steps and that the sequence is periodic due to the modulus. Variations of the generator function were discussed showing that all permutations of $f(x) = x^2 + b$ should be equally useful for this method excepting where $b = -2$.

Later, [Flajolet and Odlyzko, 1990] investigated these random mappings, that were attracting interest, and developed a framework to characterize random mappings through the use of generator functions to profile the statistics over all permutations of variables accepted by generators followed by singularity analysis for any exceptions or significant features. Importantly the use of directed functional graphs is introduced to represent the generated mappings, and graph features are used to better understand the nature of these maps. The term asymptotic analysis is coined where for all permutations of generator parameters that are investigated, how the limiting case for the various features behaves as the functional graphs become large. This gives us a measure of what can be expected from these features from a typical graph. This paper also begins study of automated analysis, or computer experimentation to implement this framework for analysis of specific generator functions.

The features discussed include tree or branch length, maximum cycle length, connected components among other features, and some important results are found. Similar to the result from [Pollard, 1975], the maximum cycle length is of the order $\sqrt{p}$ for a finite field of p elements, F_p. Another result is that random functional graphs have generally a few large trees that are connected to one giant component, which is demonstrated with computer analysis of the DES cryptographic system as a generator function for pseudo-random maps.

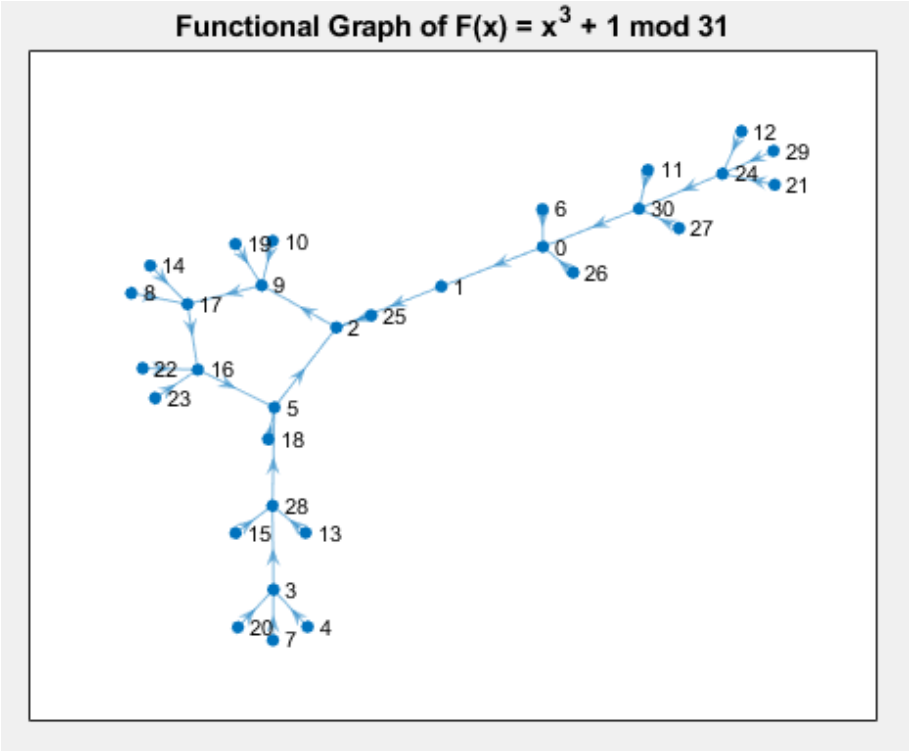

Figure 1.2: Functional Graph generated from $f(x) = x^3 + 1 \ (mod\ 31)$.

As an example of using directed functional graphs to visualize a random mapping, a simple directed graph generated by the equation $f(x) = x^3 + 1 \mod 31$ is shown in Figure 1.2. This process of generating a number sequence using a random walk is the basis of what is studied in this thesis. Note that the addition of the modulus for the generator equation creates a limit on the field of integers to the size of the modulus and forces the end of the sequence into a cycle. Starting at x = 5, it is clear that the modulus creates a finite field, and so the iterative path becomes a cycle of size 5. Iterating from vertex 5 for example, gives the sequence: $5 \rightarrow 2 \rightarrow 9 \rightarrow 17 \rightarrow 16 \rightarrow 5$. Also note that the degree of the generator function is 3 though is not a regular degree three graph, and that this is defining the number of inbound edges, or the 'in degree' of each node, with a few exceptions.

Following on from these papers, [Vasiga and Shallit, 2004] directly looks at two quadratic cases from [Pollard, 1975] for the general case discussed of $f(x) = x^2 \mod p$ and the exception case of $f(x) = x^2 - 2 \mod p$ due to its involvement with Mersenne numbers. Applying the asymptotic analysis framework discussed in [Flajolet and Odlyzko, 1990] to analyze these two functions to investigate the cautionary note and explain with a quantitative analysis as to why the features of the mapping $f(x) = x^2 - 2 \mod p$ are not good for Pollards factorization algorithm and further number theoretical analysis of Mersenne and Fermat prime numbers.

In [Konyagin et al., 2016] the complimentary set of quadratic mappings for the $f(x) = x^2 + a \mod p$ is studied where all permutations for $a \mod p$ are considered except for $a = 0, -2$, and proves that this generator will test all permutations of quadratic functions up to isomorphism, which is a reduction from the complete quadratic form $f(x) = ax^2 + bx + c \mod p$. The paper investigates the nature of isomorphic graphs for this generator and continues with the framework from [Flajolet and Odlyzko, 1990] to profile these permutations of a for a large number of primes. The results concerning the number of components are particularly interesting as the average number of components per graph is around 7, with the minimum being 1 and the maximum over one hundred for these large primes on the order of 500,000.

Another significant result is that the average number of elements in a cycle is around 866 for primes of the order again of 500,000, with a minimum cycle size of 2 elements and a maximum between 3000 and 4000 elements. This is just over the expected result of $\sqrt{p}$ and justifies the expectation from [Pollard, 1975, Flajolet and Odlyzko, 1990]. This paper is followed by [Mans et al., 2017] where the more precise relationship for average cycle size in all graphs is found to be $\sqrt{\pi p/2}$, and unexpectedly that the average size of the cycles from connected graphs is lower than that of unconnected graphs. Other graph features are

quantified such as tree size and number of trees with size one.

1.3 Functional Graphs of Polynomials and their Twist

From papers mentioned in the previous section, much of the form and features of the graphs generated from quadratic polynomials is known. The cycles will be roughly the square root of the size of the finite field of elements, the graphs are often unconnected and may have many components and that the functional graphs have both tree and cycle features as well as isomorphic properties. For the purposes of generating long sequences of unpredictable pseudo-random numbers using a graph feature that incorporates as much of mapped field as possible is desirable. Previously mentioned generator functions are limited by having multiple trees branching off each cycle, splitting the vertices into smaller features, and often unconnected graphs creating multiple cycles with additional trees for functional graph. Using the largest cycle in these functional graphs as a sequence generator is shown to use only the square root of the field size on average.

It is desirable to avoid the creation of unconnected graphs, increase the size of the cycle and reduce the number of trees, or remove them entirely so that longer sequences of pseudo-random numbers may be generated from the largest cycle in each graph. By introducing multiple solutions for the outbound degree by introducing a squared term, two possible edges out of each vertex are created such that $y^2 = f(x)$. Following this, to find the next vertex on the random walk the square root of $f(x)$ must be found where $\pm y \mod p$ are solutions. This is more likely to produce strongly connected functional graphs than that of the example graph in Figure 1.2 due to the increase in out-degree for all vertices.

Not all edges will have solutions however, as only those with rational solutions along the curve $f(x)$ have outbound edges. This leaves about half of the vertices without solutions which will end the sequence at a stopping point with no cycle. To generate these missing edges a twist term is added to the equation when there is no solution. This twist term is referred to here as λ which can be any rational number in F_P that is not quadratic residue modulus prime p. Adding this alternate twisted function defines a set of equations as described in Equation (1.2) where C_0 is the untwisted function, and C_1 is the twisted function that is used if there is no solution found with C_0. The notation $G(\lambda, f(x))$ describes a graph with these parameters for a given prime p where the function and the twist are neatly shown.

$$\begin{aligned} C_0 &: \quad y^2 = f(x) \bmod p, p \in P. \\ C_1 &: \quad \lambda y^2 = f(x) \bmod p, p \in P. \end{aligned} \tag{1.2}$$

Intuitively, this generator function should produce more connection in the graphs and are unlikely to have trees or branches. In fact we will show that they are strongly connected and have many Hamiltonian cycles per connected graph or component as opposed to the functional graph shown in Figure 1.2. Given the strongly connected expectations there was also the prospect of finding Hamiltonian cycles which could be of the maximum size cycle.

Generator functions of the form $y^2 = f(x)$ have not been tested before, so the techniques and algorithms used from previous studies such as cycle detection redesigned for this new type of generator. There are twice as many edges to traverse when searching the functional graphs and the presence of cycles smaller than Hamiltonian mean a cycle may encounter the same vertices from a different origin as there must be as many inbound edges as outbound edges. The complexity of generators of the form $y^2 = f(x)$ could be as high as 2^p given each graph has p vertices and each vertex has two out edges. As such, this thesis will design efficient algorithms to investigate these new functional graphs.

1.4 Hamiltonian Cycles

Following on from the expectation that Hamiltonian functional graphs are generated with the twisted polynomial function, there is a practical application in cryptography if it can reliably produce graphs that have a Hamiltonian cycle. The Hamiltonian cycle is this special case where there is a cycle that passes through each element of the functional graph.

The Hamiltonian cycle is a well-known computational problem. Testing if a cycle in the graph is Hamiltonian can be done in polynomial time but typically finding an Hamiltonian cycle quickly becomes difficult as the size of the graph increased. Algorithmic techniques exist to avoid calculating all possible paths, in particular for directed Hamiltonian cycles (e.g., [Kühn and Osthus, 2012]) but they remain computationally expensive in the general case. The problem of finding a Hamiltonian Cycle for a given graph is the basis of public key cryptography in digital communications which is related to another difficult computational problem (i.e., the factorization of primes factors from large numbers [Diffie and Hellman, 1976, Rivest et al., 1978]) as well as other proposed cryptographic methods such as using the NP-Complete knapsack problem [Merkle and Hellman, 1978].

Analyzing the number of Hamiltonian cycles in a given graph becomes clearly more difficult as this involves the same process of finding all Hamiltonian cycles. In this thesis, this is particularly challenging as there are many functional graphs for each prime, and ideally the analysis needs to be done for large primes. If these graphs are tested for the

existence of a Hamiltonian cycle and it turns out there is not one then the time complexity is also out of reach using an exhaustive search of all possible paths. We will show that these problems quickly become intractable as the size of the finite field grows. Finding practical algorithms for functional graphs must be done by exploiting features of the graph, as they deviate from the general case, and by designing short cuts for processing specific cases to reduce the complexity. The generator functions for the graph described in Section 1.3 allow a simpler case for Hamiltonian analysis as the graphs are of small degree.

Finding efficient solutions to specific cases of intractable problems has been well studied. While the general case of many problems remain intractable, under certain conditions the complexity of the problem is reduced significantly and the use of probabilistic and heuristic techniques are often used to solve questions around the existence of a solution, such as determining the existence of a Hamiltonian cycle is unlikely. Frieze and Haber [Frieze and Haber, 2015] devised a near linear time heuristic algorithm for finding Hamiltonian cycles for a limited set of directed functional graphs where they are of degree at least three. By targeting a specific case of functional graphs it is possible to exploit the features and in this case, as the ratio of edges to vertices increases the algorithm becomes more efficient to the limiting case of complexity $O(n^{1+o(1)})$. It is also worth noting that this method determines if the functional graph tested is Hamiltonian by finding one Hamiltonian cycle and so ignores any additional Hamiltonian cycles after the first. For the purposes of finding all Hamiltonian cycles this heuristic is not suitable. A similar heuristic method is used for a broader set of functional graphs in the un-directed case in [Baniasadi et al., 2014], and a probabilistic heuristic for directed graphs in [Frieze, 1988]. In [Hefetz et al., 2016] a heuristic for testing if a graph is Hamiltonian is developed from a proof in 1960 that asserts that a functional graph of degree more than half the number of vertices, and discuss the probability of a functional graph being Hamiltonian as the degree is reduced below this.

1.5 Summary of Contribution

Functional graphs of a polynomial and its twist as described in Section 1.3 have not yet been studied, and as outlined the generator function allows us to specify a great variety of different polynomials to use the enhancements of twist and forcing out-degree of two. This thesis will focusing on the linear case, where $f(x) = x + a$. This is a contrast to the previous literature that has focused on $x^2 + a$ but as the y^2 term and the twist make this generator much more complicated already the linear function will not produce simple functional

graphs. So our general twisted Equations (1.2) become the following in Equations (1.3).

$$\begin{aligned} C_0: &\quad y^2 = x + a \bmod p, p \in P. \\ C_1: &\quad \lambda y^2 = x + a \bmod p, p \in P. \end{aligned} \tag{1.3}$$

Figure 1.3 is a visualization of a small directed graph $G(6, x+6)$ for prime $p = 7$, generated by the function described by Equations (1.3). Graphs where $p = 7$ can be calculated by hand without too much trouble from Equations (1.3) and some of our intended outcomes are evident in the small graphs. Tracing around the outer edge of the connected graph it is easy to find a Hamiltonian cycle, and to count the edges and see that each vertex aside from vertex one has out-degree two.

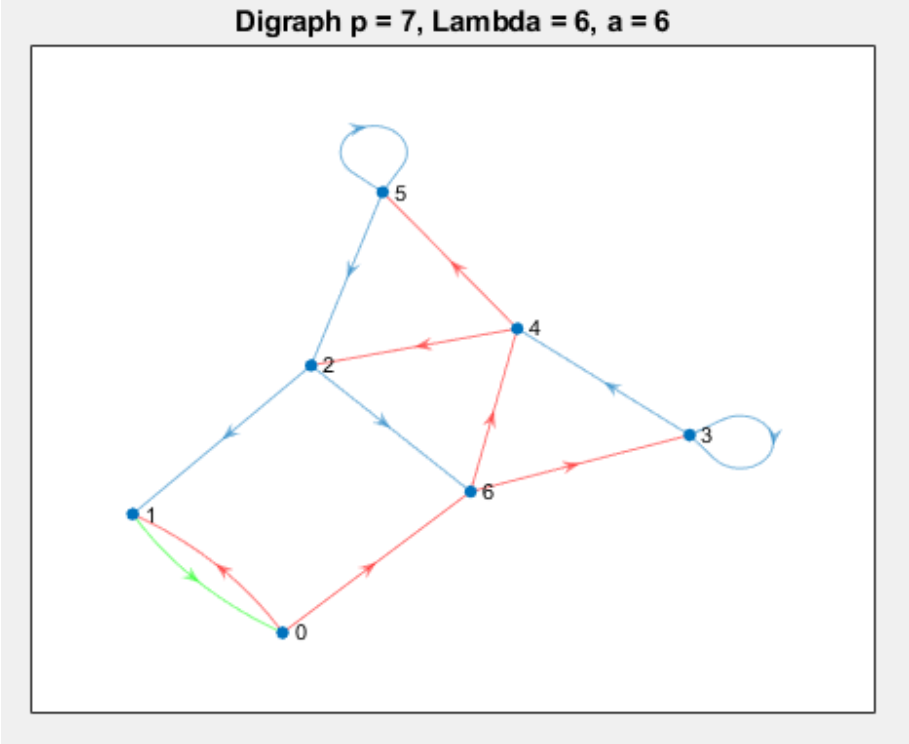

Figure 1.3: Example of directed graph $F(x) = x + a \bmod p$

The analysis framework described in [Flajolet and Odlyzko, 1990] will be applied where all combinations of variables λ and a for primes p up to infeasibility. This involves developing and writing software tools to perform this combinatorial analysis and visualize the functional graphs to inspect any features that are discovered in the process. Asymptotic results are determined for the functional graphs, the connectivity of the graphs is investigated for

both connected and unconnected graphs and cycle properties including an analysis of Hamiltonian graphs.

Computational analysis conducted in collaboration with theoretical work is outlined in [Mans et al., 2019]. In this way, computational analysis is used to confirm theoretical expectations and conversely, use the results of computational analysis to uncover unexpected features of these graphs that are useful to theoretical development. The contribution this research makes is to examine the pseudo-random generator function outlined here and provide computational analysis that supports theoretical research into a new method of generating strongly connected Hamiltonian graphs that produce extended pseudo-random sequences to that of previous studies and to highlight any features found that may be of use to pseudo-random number generation, cryptographic techniques and number theoretical knowledge more generally.

The investigation conducted in this thesis has discovered that the twisted polynomial generator produces almost entirely connected graphs, and that these graphs are strongly connected graphs despite all being small degree graphs that have a maximum in-degree and out-degree of two. Another key finding is that all connected graphs and unconnected components are Hamiltonian, and all have a great many Hamiltonian cycles, the number of which grows exponentially proportionately to the field size. And since all graphs are Hamiltonian, the cycle size is a one to one mapping of the finite field, which greatly improves on previous polynomial generator cycle sizes averaging the square root of the finite field size. Another discovery of this thesis is a feature of the generator allows these Hamiltonian cycles to be used to generate balancing binary sequences.

1.6 Structure of Thesis

Section one already outlined research intentions and reviewed existing literature that supports the approach taken in this thesis to enhance existing functional graphs of polynomials for the purpose of sequence generation. Section two discusses the development of software tools, methods and algorithms that produces the mapping from the generator function, as well as the methods used to visualize the functional graphs produced by the random mappings for small finite fields.

Section three discusses the algorithm used to analyze the connectivity of the functional graphs, the results found from the computational analysis and compares them with theoretical expectations. Specifically an asymptotic analysis of the connected and unconnected graphs is conducted including unconnected component analysis. Assessment of exceptions to the asymptotic results including some cases of isomorphic conditions is also demon-

strated.

Section four conducts a similar analysis on functional graph cycles for connected graphs establishing that the graphs are Hamiltonian and counting the number of Hamiltonian cycles produced. Discussion include a comparison of theoretical expectations and experimental results, with some comments on the limitations of the analysis due to complexity. A method of producing balancing binary sequences is discovered and analyzed as a potential source of pseudo-random binary sequences, and some special cases are analyzed and compared with theoretical assessments.

Finally in Section 5 the results of this thesis are discussed and measured against the expected benefits while also considering some of the drawbacks encountered in this approach. Possible uses for the generator are discussed from the properties discovered in these functional graphs and future research directions are considered.

2 Building Experimentation Tools

2.1 Development Environment

The analysis tools developed and used to study the functional graphs generated by Equations (1.3) are written in C++, which was chosen for processing efficiency over higher level programming languages and also compatibility with numerical tools for large integer handling [Shoup, 2019] written specifically for C++. Microsoft Visual Studio was used as the IDE for this project due to prior experience with the tools, and that it integrated well with the development environment that was running Windows 10 and using 64-bit Microsoft compilers.

All code was first developed as single-threaded code with centralized variables for ease of debugging, and then multi-threaded if required. Once the code was tested, it was recompiled for Linux for use with The National Computing Infrastructure (NCI) [NCI, 2018], a dedicated high performance Linux cluster of 84,656 cores over 4416 nodes, housed at the Australian National University. Access to the NCI was granted through partnership with Macquarie University. Intel compilers recommended by NCI user documentation were used to compile for Linux executable code and some code changes were made in text to be compatible from the Windows executable code due to some differences in compilers.

The code was largely written using native C++ vectors for the ease of not needing to define the size which was particularly useful for vectors of temporary vertices which could be pushed back and indexed easily for iteration over graph vertices. C++ structs were used to create data structures to hold all data for edge tables and results, which allowed for thread safety in multi-threaded code when calling functions and passing data. In all environments, code was optimized to run on multiple threads where it made sense to do so using the boost C++ libraries [Anthony Williams and Vicente J. Botet Escriba, 2017] as C++ does not have native threading modules. The Boost libraries were chosen for compatibility with available boost modules installed on the NCI. Collected data was exported from the application by printing to command line which was saved to text file for import to Microsoft Excel for display and analysis. Design and development of code was an iterative process of experimentation where results were examined and design was reassessed for improvements to existing code and also for new research objectives as they were discovered.

Much of this project was iterative, and if the experiment was consuming significant processing time, iterations were run in on multiple threads to utilize many processor cores. The benefits of using the NCI for processing were that the NCI has significantly more cores to process iterative jobs and many experiments can be run concurrently on different compute

nodes. The multi-threaded jobs were somewhat inefficient, running at a large fraction of the maximum processing power that was attributed to on-processor resource management where the memory management for multiple cores accessing memory for iterative tasks will be impacted by threads competing for memory space. Multi-threading efficiency was observed to vary due to processor architecture and also the size of the primes. For example, the efficiency of multi-threading using the NCI increased for larger primes, which seems counter intuitive but may be explained by the larger tasks taking longer to process and so transfer between memory levels may be less frequent on the processor, reducing a bottleneck.

Multi-threaded code was developed and tested first on the Windows desktop computer running a quad core processor with a maximum of 3 threads to avoid the use of Hyper-threading and to avoid starving the operating system of resources, both of which would hinder performance. Once tested, the code was reformatted to run under the NCI compilers for Linux and recompiled to run as batch tasks on single nodes, which consisted of two processors of either eight or twelve cores each, providing a single processing unit of 16 or 24 cores depending on which queue was used (or available). Later, the windows desktop was upgraded to an AMD Ryzen Threadripper2 32 core processor which had the benefit of additional on CPU resources as well as a higher overall processing speed compared with previous environments including the NCI and allowed for quick testing multi-threaded tasks and high performance processing of a single multi-threaded task. Processing was again limited to $n-1$ threads for an n-core processing node to avoid over threading leading to degraded performance due to on processor resource limitations. Hyper-threading is formally switched off on the NCI due to it having a negative impact on scientific processing similarly due to processor resource limitations and thread handling bottlenecks. Given the abundance of processors and that most tasks ran at close enough to full load this was still a great advantage. Little time was spent investigating processing speeds and compiler optimization as it was outside the scope of this project, just the minimum time necessary was spent to achieve good performance.

Load balancing multi-threaded code was manually tuned so that tasks were not allowed to be queued over a certain limit of tasks in memory so that there was never a wait for new tasks, but also the tasks in memory never grew to a size that would use too much memory while waiting for a processing thread from the thread pool. To achieve this, the thread pool was monitored while queuing tasks and a small threshold of tasks was set that would halt queuing until the tasks dropped below this threshold. Manual tweaking of the task pool and the job queuing process sleep period would allow the queue to be full and ready to process

new jobs, but keep memory usage of queued threads to a minimum.

Likewise, the output data structs of the threads were cleared early by calculating results such as averages on the fly as the size of the data for all permutations becomes large if all stored together and processed later. This thread-balancing was done to prevent an out of memory error as if all threads were queued or all processed task results were left in memory the combined data structs of mapping tables and result data would create a prohibitive memory requirement. Another consideration was that recursive functions might not be inherently thread safe using structs to pass data on threads for later analysis of Hamiltonian cycles.

Message Passing Interface (MPI) parallel computing libraries were investigated to run analysis over multi-node batch jobs with the NCI however, the usage constraints and re-coding time limited the advantages over submitting multiple single node batch jobs using multi-threading to use all of the available cores of a single compute node.

2.2 Implementing the Maps from Function Parameters

The developed tools to generate the mapping table require the three function parameters p the prime modulus, λ the twist coefficient and a the constant, all of which are in F_p. λ has two extra conditions where any λ must be excluded that are quadratic residue for Equation (1.3), and roughly half of the remaining λ values are also removed due to isomorphism as explained in [Mans et al., 2019] due to the condition $G(\lambda, x+a)$ is isomorphic to $G(\lambda^{-1}, X+\lambda a)$, where isomorphic functional graphs will have identical results for graph feature analysis.

Removing any λ that is quadratic residue for a given prime modulus p is done by creating a list of all squares modulus $p \in F_p$. Any integer $x \in F_p$ found in the list is a quadratic residue modulus p and so is excluded. An example table of squares modulus $p = 31$ is shown in Figure 2.1, showing there is symmetry about the modulus. Taking $x = \pm 11 \mod 31$ gives the pair $x = 11, 20$ with a quadratic residue of 28. Allowing for $x = 0$, this means there are $p - 1/2$ non-zero quadratic residue values in F_{31}

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
x^2 mod 31	0	1	4	9	16	25	5	18	2	19	7	28	20	14	10	8	8	10	14	20	28	7	19	2	18	5	25	16	9	4	1

Figure 2.1: Example table of squares modulus 31

Reduction due to isomorphism is more complicated as for each non-quadratic residue λ there will be a modular inverse non-quadratic residue such that $\lambda \times \lambda^{-1} = 1 \mod p$, where λ can be its own modular inverse. By looping through the available λ values the modular inverse condition is tested and the larger of the two values is removed. Where λ is found

that is its own modular inverse, it still needs to be processed. This reduces the field of useful λ by half, plus one if there is an odd number of λ values.

Algorithm 1 Mapping and Edge Table Generation

Require: Prime p, Twist λ, Constant a, Array of Squares y^2,
1: **for** all $i \in p$ **do** ▷ Build the in-bound edge function table
2: MappingTable[i] ← Calculate preceding vertices x_{C0} and x_{C1} from i
3: **for** all $i \in F_p$ **do** ▷ Transform the mapping table to the edge table by indexing from x instead of to x.
4: **for** x_{C0} & x_{C1} **do**
5: **if** y_0 = undefined **then**
6: EdgeTable← Set Edges from Mapping Table($[x_{C0}, x_{C1}] \rightarrow i$) as ($x \rightarrow y_0$) and Twist $T[0,1]$
7: **else**
8: EdgeTable← Set Edges from Mapping Table($[x_{C0}, x_{C1}] \rightarrow i$) as ($x \rightarrow y_1$) and Twist $T[0,1]$
9: **if** y_0 and $y_1 = 0$ **then**
10: EdgeTable ← Set Twist T = 2
11: **return** $EdgeTable$

Algorithm 1 shows the process used to generate the mapping table as a two-dimensional vector from the above three parameters and a table of squares for F_p. Solutions are calculated for (1.3) in reverse for both C_0 and C_1 by rearranging the equation for x as shown in Equations (2.1). This is done so there is no direct computation of a square root term and a table of squares can be can pre-calculated for F_p as this is more efficient than calculating the squares during iterations and calculating squares is easier than calculating roots. The effect of reversing the equation is that rather than finding the outgoing edges from vertex x, ie. $x_n \rightarrow x_{n+1}$, we are instead calculating the incoming edges to x, i.e. $x_{n-1} \rightarrow x_n$. To represent the previous edge of each vertex, these in-bound edges are labeled from vertex x_{C0} and x_{C1} for the two rearranged equations.

$$\begin{aligned} C_0 : &\quad x_{C0} = x^2 - a \bmod p \\ C_1 : &\quad x_{C1} = \lambda \times (x^2 \bmod p) - a \bmod p \end{aligned} \tag{2.1}$$

The table shown in Figure 2.2 is the result of line 1 and 2 in Algorithm 1 where the preceding vertices are found for each vertex in the field P. This in itself is sufficient for building an edge table to visualize graphs but since these directed graphs are to be analyzed iteratively, looking up edges from a table like Figure 2.2 is poorly indexed. It is better to have an orderly table

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
C0	30	0	3	8	15	24	4	17	1	18	6	27	19	13	9	7	7	9	13	19	27	6	18	1	17	4	24	15	8	3	0
C1	30	2	11	26	16	12	14	22	5	25	20	21	28	10	29	23	23	29	10	28	21	20	25	5	22	14	12	16	26	11	2

Figure 2.2: Mapping table of $[C_0, C_1] \rightarrow x$ for inputs $p = 31$,$\lambda = 3$, $a = 1$

to look up the outbound edges for each vertex as done in a random walk. The remainder

of Algorithm 1 takes each edge from the mapping table and re-index from x to y_0 and y_1, also taking note of whether the edge came from C_0 or C_1 as this defines if the path used the twist function or not. The final edge table is shown for the same functional graph in a table in Figure 2.3. For the edge table, the corresponding twist value is shown and marked also as a colour code (blue and $T = 0$ for no twist via C_0, red and $T = 1$ for twisted function C_1). As you can see by comparing tables the relationship for the function path is lost when looking at outbound edges where it is clearly defined from the table of inbound edges. The exception is edges leading to the zero vertex will be one from each the twisted and the non-twisted function so this is tested in the algorithm and assign this special case a twist T of 2 to differentiate it.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
y0	1	8	1	2	6	8	10	15	3	14	13	2	5	13	6	4	4	7	9	12	10	11	7	15	5	9	3	11	12	14	0
y1	30	23	30	29	25	23	21	16	28	17	18	29	26	18	25	27	27	24	22	19	21	20	24	16	26	22	28	20	19	17	0
T	0	0	1	0	0	1	0	0	0	0	1	1	1	0	1	0	1	0	0	0	1	1	1	1	0	1	1	0	1	1	2

Figure 2.3: Edge Table for inputs $p = 31$,$\lambda = 3$, $a = 1$

Algorithm 1 was decided upon after investigation of the form of the functional graphs by hand for small p and theoretical expectations outlined in [Mans et al., 2019], proves this generator function yields a digraph where every vertex has in-degree and out-degree of two, with the exception of the zero vertex having an in degree of one and the preceding vertex with and out-degree of one. As the degree of the directed graph is 2 and always has a rational solution (or solutions), i.e. all but one vertices have two inbound edges and two outbound edges which means processing time is saved that would be required for a graph with arbitrary vertex degrees. The exception being the zero vertex having only one in edge, however this can be viewed as having the same solution for both the twisted and untwisted functions as shown in Figures 2.2 and 2.3. To represent this the label ‘2‘ is used as the function indicator to differentiate from the other edges and designate these values in green on tables and graphs. This label of ‘2‘ will be used in other algorithms to quickly identify the zero vertex.

2.3 Graph Visualization Tools

From the edge table created in the previous section it is trivial to output a text script, formatted to execute in MATLAB to use the ’digraph’ graphing tools available in the application.

An example of the MATLAB diagraph output is shown in Figure 2.4 where the numbered vertices are joined by the edges defined in Table 2.3 and the edges are color coded to match the twisted (red) or untwisted (blue) function choice. The special case of the zero or double edge mentioned earlier is in green.

The use of a visualization of these functional graphs is important for analyzing the topology of the map and unit testing. By following through the mapping table, it is easy to verify (for small graphs!) that the edges are as expected and also to test specific input variables individually to consider theoretical proofs.

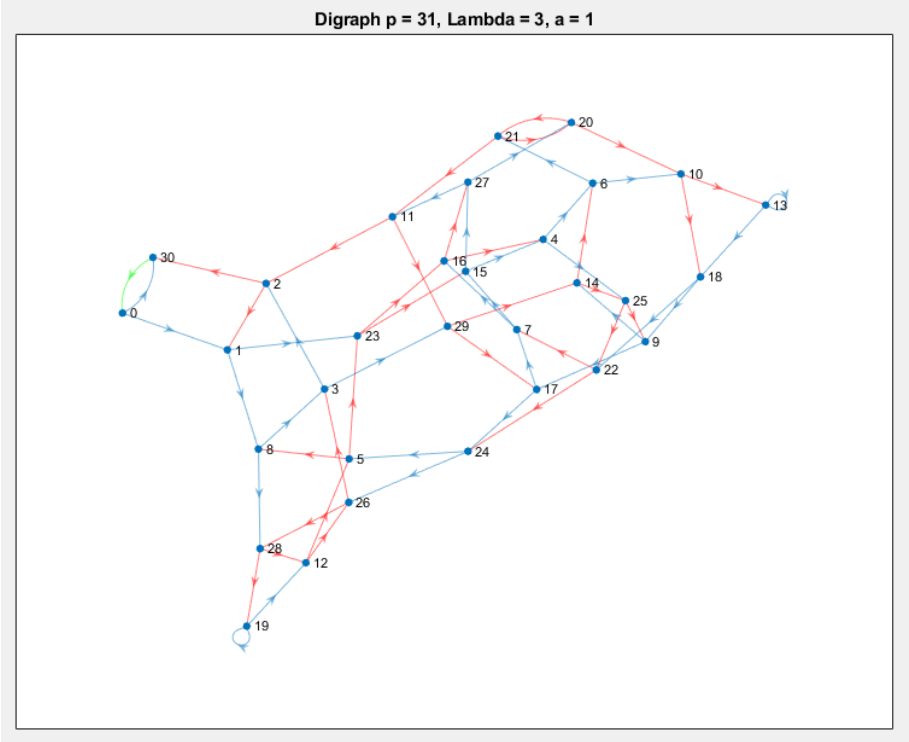

Figure 2.4: Example generated functional graph for inputs $p = 31$,$\lambda = 3$, $a = 1$

Figure 2.4 is the MATLAB generated functional graph from the edge table in Figure 2.3 where one can follow the edges on the table around the graph from vertex to vertex and check the function path by colour. The in-bound edges from Figure 1 allow one to work backwards along the directed path. This visualization is used in the following sections to analyze interesting features of these functional graphs of polynomials and their twist that are discovered from iterative analysis.

3 Connected Graph Analysis

3.1 Graph Component Detection Algorithm

The generated edge tables from Section 2.1 of this thesis are used by a component detection algorithm to profile the corresponding functional graph. Examining all graphs for a prime number is time consuming and had to be limited to smaller prime numbers. However, as outlined by [Flajolet and Odlyzko, 1990] these functional graphs are profiled statistically by looking at all permutations of variables over a range of prime numbers. From these computational results asymptotic trends are analyzed, and from these observations are then explored for any exceptions to the general case.

A connected component consists of vertices that are all connected by edges in a graph, where a graph will have one or more connected components. A strongly connected component is one where any vertex is reachable from any other vertex in the graph component. The connected components are tested for maximum and minimum numbers of components, maximum and minimum component sizes and what the averages of these values are over a range of primes. This gives an indication of how these features compare with previous studies. This component detection algorithm is designed to measure these properties and is iterated over all combinations of λ and a values for as many prime numbers as feasible.

Algorithm 2 Graph Component Detection

Require: As Input: Prime p, vector<x,y> *EdgeTable*

$S_n \Leftarrow 1$
for $i : 0 \rightarrow (p-1)$ **do**
  if $S[i]$ is Undefined **then** ▷ Only processes vertex if not already assigned a set in vector 'S'
    $j \Leftarrow i$
    while $S[EdgeTable[j, y_0]]$ and $S[EdgeTable[j, y_1]]$ Undefined **do** ▷ Iterate along first edge until flagged vertex found
      $Z[] \Leftarrow EdgeTable[j, y_0], EdgeTable[j, y_1]$ ▷ from the jth vertex, add both adjacent vertices y_0 and y_1 to visited set vector 'Z[]'
      $S[EdgeTable[j, y_0]], S[EdgeTable[j, y_1]] \Leftarrow -1$ ▷ Flag all checked edges with temporary set
      $j \Leftarrow EdgeTable[j, y_0]$ ▷ Iterate the path
    if $S[EdgeTable[j, y_0]]$ Xor $S[EdgeTable[j, y_1]] > 0$ **then** ▷ One existing set found, on y_0 OR y_1
      Assign all $Z[]$ to the existing set found.
      $Z[] \Leftarrow$ clear
    else if $S[EdgeTable[j, y_0]]$ and $S[EdgeTable[j, y_1]] > 0$ **then** ▷ Two existing sets found, merge all to one set.
      Merge sets and assign all $Z[]$ to merged set.
      $Z[] \Leftarrow$ clear
    else ▷ Loop found without any assigned vertices
      $S_n \Leftarrow S_n + 1$
      $S[Z[]] \Leftarrow S_n$ ▷ Assign all stored vertices to new set
      $Z[] \Leftarrow$ clear
for $i : 0 \rightarrow (p-1)$ **do** ▷ Loop through Set vector 'S' and merge all partial sets
  if $S[i] \neq S[EdgeTable[i, y_0]]$ and/or $i \neq S[EdgeTable[i, y_1]]$ **then** ▷ Partial sets found
    Merge sets $S[i], S[EdgeTable[i, y_0]], S[EdgeTable[i, y_1]]$
  return $S[]$

Algorithm 2 is a new algorithm designed to exploit the specific form of the degree two in and out edges of graphs generated to perform a component detection test for a given edge table, generated as described in Section 2 of this thesis. From an edge table an iterative process can be used to discover how many connected components there are in the functional graph by walking through the edges and sorting the vertices into sets that are connected to each other. This becomes an optimization problem to test all combinations of λ and a values for large primes p which will become limited by the size of p and will have roughly $p^2/4$ graphs to test per p given all $a \in F_p$ are looped through. Due to quadratic residue eliminating half of λ values and isomorphism removing roughly half of the remaining λ are left with near one quarter of the $\lambda \in F_p$ as discussed in Section 2.2 of this thesis.

Given that this is a special case of graph to profile, i.e. a two-regular digraph, every vertex will have exactly two outbound edges so the problem is reduced in complexity compared with higher degree graphs and graphs with unknown or variable degrees. From Algorithm 2, the process of sorting the vertices of a graph into component sets is to test for adjacency to other vertices. This algorithm uses a native C++ vector '$S[]$' (not the actual C++ set class) to store the set label assigned to each vertex, where the index is the x value from the edge table. Looping through the vertices in P a check for an previously assigned set for the vertex in the set vector '$S[]$'. If not, then iterate through the edge table along the y_0 path, testing both y_0 and y_1 for each vertex and for every vertex found not to be set, assigning it a temporary set of -1 to indicate that the vertex has been visited on this iteration of the while loop. Figure 3.1 visualizes the vertices captured by the adjacency testing iteration for a subsection of a functional graph $G(\lambda, x+a)$.

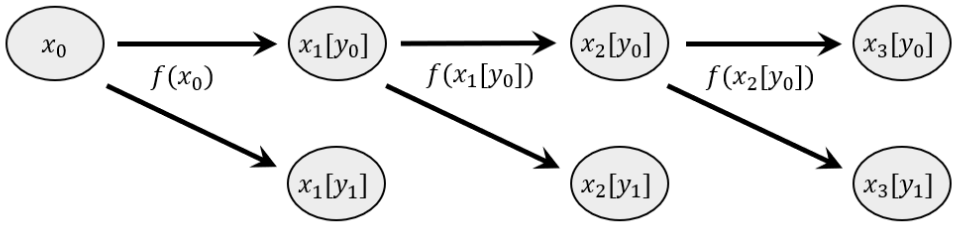

Figure 3.1: Adjacency testing iteration for Algorithm 2

This loop continues along the $x \rightarrow y_0$ iterations as defined by the edge table until an existing set is found, or a -1 flag is found, indicating that the algorithm has walked around a loop. In the case of finding an existing set, the walk stops and the algorithm loops through the '$S[]$' vector and assign all vertices flagged with -1 to the adjacent set found. It is possible that two different adjacent sets are found on y_0 and y_1 for a given vertex and in this case we

pick one and merge the two sets as well as the -1 flagged vertices. In the case where a loop is found, there is no set assigned so a new set is created and all flagged vertices are assigned to this new set.

This creates a number of sets that will almost certainly be subsets of component vertices as there has not been an exhaustive search of the graph on each iteration of the while loop and the algorithm has only been walking along one edge of the two out-bound edges. The advantage of this method is it keeps the logic simple and requires little resources to keep track of an growing set of vertices. This is important for multi-threaded processing as any stored path information will impact on processor resources. Recursive testing is also not safe for this test due to potential for stack overflow as the finite field is increased.

This method still takes advantage of the graphs regular form to test two vertices per step and reduces the number of loops required to merge sets when compared with just testing two edges per adjacency test and moving to the next iteration as longer path sections means a reduction in loops to merge sub paths. This method is hitting a compromise between additional logic that slows the processing and adds additional memory requirements, and many small path sections that are expensive to merge.

The final step of the algorithm is to merge the sets that are subsets of larger sets. To do this, one last loop through all the of the vertices in the graph is required to test that the edge and the two adjacent vertexes are in the same set. If there are two or even three different sets, these are merged once again by picking one set label and looping through $S[]$ setting all other values to the retained set. The result is a vector of set values indexed by the vertex number which can then be used to count distinct sets and the number of vertices contained in them. If there is only one set then the graph is a connected graph. Any number of sets greater than one is a graph with some unconnected components which will need further investigation.

3.2 Graph Connectivity

The proportion of connected functional graphs generated from a polynomial function is a key feature this thesis is aiming to improve from previously studied functional graphs with the choice of generator functions as described by Equations (1.3). Ideally a high proportion of graphs per prime number that are connected graphs would be produced by this generator and it was expected that this would be the case. Experimentally, the same approach as initially discussed in [Flajolet and Odlyzko, 1990] is used, where asymptotic results are

assessed before any exception analysis. These graphs are processed for all combinations of values for λ and $a > 0$ where $p \leq 5003$, and sample values at $p = 10007$ and $p = 20011$ excluding λ values for quadratic residue or isomorphic duplicate as discussed previously.

In Table 3.1 the results of this iterative analysis are shown where the number of connected graphs per prime p has been collected as a percentage (G_C%), the number of connected graphs (G_C), and the number of disconnected digraphs (G_U). Along side these values are the asymptotic approximations $(p^2 - p)/4$ for connected graphs, and $p/4$ for disconnected digraphs. These values are guided by fitting a curve and the $1/4$ term is fairly clearly in line with the $\lambda/4$ reduction due to the number of iterations. Another observation from early results was that any $a = 0$ graph would always have a self-loop on zero, which makes the zero element an connected component of size one having both edges on the same self-loop. As such all $a = 0$ tests are excluded from results. Figure 3.2 shows examples of graphs where $a = 0$.

p	G_C %	G_C	$(p^2-p)/4$	G_U	$p/4$	p	G_C %	G_C	$(p^2-p)/4$	G_U	$p/4$
5	75%	3	5	1	1.25	101	98.96%	2474	2525	26	25.25
7	91.67%	11	10.5	1	1.75	149	99.31%	5438	5513	38	37.25
11	90%	27	27.5	3	2.75	199	99.51%	9851	9850.5	49	49.75
13	88.89%	32	39	4	3.25	293	99.65%	21242	21389	74	73.25
17	90.63%	58	68	6	4.25	397	99.74%	39104	39303	100	99.25
19	94.44%	85	85.5	5	4.75	499	99.8%	62125	62125.5	125	124.75
23	96.21%	127	126.5	5	5.75	599	99.83%	89551	89550.5	149	149.75
29	95.92%	188	203	8	7.25	691	99.86%	119197	119197.5	173	172.75
31	96.67%	232	232.5	8	7.75	797	99.87%	158204	158603	200	199.25
37	96.91%	314	333	10	9.25	887	99.89%	196471	196470.5	221	221.75
41	97.5%	390	410	10	10.25	997	99.9%	247754	248253	250	249.25
43	97.62%	451	451.5	11	10.75	1249	99.92%	389064	389688	312	312.25
47	98.01%	541	540.5	11	11.75	1499	99.93%	561375	561375.5	375	374.75
53	97.93%	662	689	14	13.25	1747	99.94%	762565	762565.5	437	436.75
59	98.28%	855	855.5	15	14.75	1999	99.95%	998501	998500.5	499	499.75
61	98.22%	884	915	16	15.25	2477	99.96%	1532024	1533263	620	619.25
67	98.48%	1105	1105.5	17	16.75	2999	99.97%	2247751	2247750.5	749	749.75
71	98.65%	1243	1242.5	17	17.75	3499	99.97%	3059875	3059875.5	875	874.75
73	98.61%	1278	1314	18	18.25	3989	99.97%	3975038	3977033	998	997.25
79	98.78%	1541	1540.5	19	19.75	4493	99.98%	5043392	5045639	1124	1123.25
83	98.78%	1701	1701.5	21	20.75	5003	99.98%	6256251	6256251.5	1251	1250.75
89	98.86%	1914	1958	22	22.25	10007	99.99%	25032511	25032510.5	2501	2501.75
97	98.96%	2280	2328	24	24.25	20011	100%	100105027	100105027.5	5003	5002.75

Table 3.1: Connected / unconnected graphs and related functions per prime number.

The clear observation from Table 3.1 is that nearly all graphs per prime p are connected, and as p becomes large G_C% approaches 100% due to the number of connected graphs growing quadratically while the unconnected graphs grow linearly. This is of great interest

and impactful as the increase in the number of connected graphs help to create a generator that produces almost entirely connected graphs. Theoretical analysis discussed in [Mans et al., 2019] proves all graphs are strongly connected despite the small degree of two which is an important result itself, and helps explain the prevalence of connected graphs.

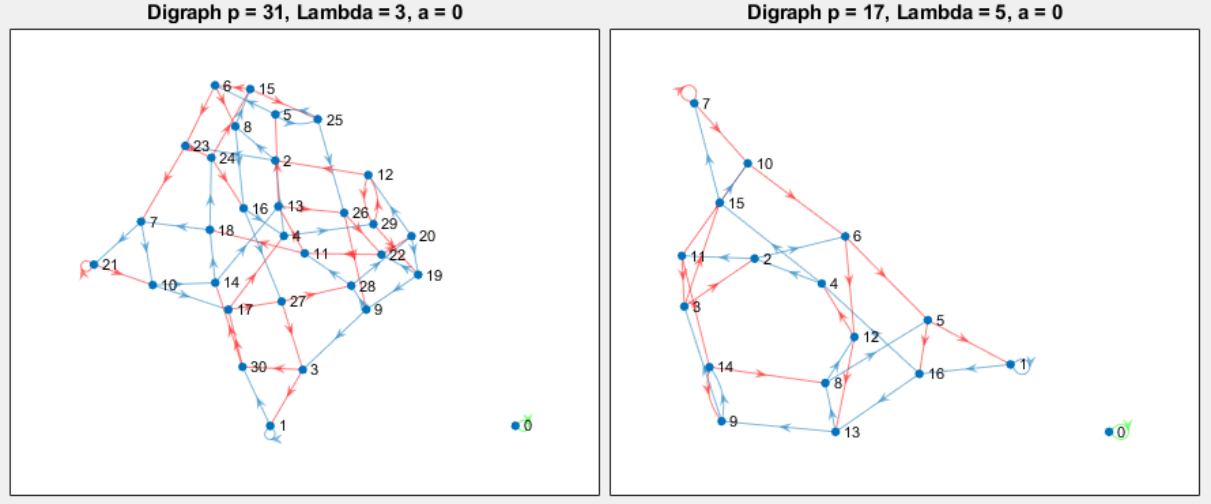

Figure 3.2: Examples of graphs where $a = 0$

3.3 Unconnected Graph Characteristics

From the previous section it is shown that while nearly all graphs are connected, there are always a small number of graphs that are unconnected. Table 3.1 shows the number of unconnected graphs is linear and grows with p at the rate of roughly $p/4$. From Table 3.2 the unconnected graph characteristics are shown for the maximum number of components per prime p (C_{Max}), the minimum size of the smallest component (SC_{Max}), the maximum size of the smallest component (SC_{Max}) and the average size of the smallest component (SC_{Avg}). The format of Table 3.2 was decided after some preliminary testing of individual graphs with the visualization tools which showed unconnected graphs having generally two components and that one component is nearly the size of the field F_p.

The first observation for the test running through all primes to $p = 5003$ for the same variables as previously, there were never more than two components found. Second, most unconnected graphs had a small component with only 2 vertices, and by inspecting the graph ratios and the average small component size it is shown that with the exception of two small primes the smallest component is of at most size 3. There are two exceptional primes, $p = 17$ and $p = 31$ where both have a small component size of 6.

These results are fairly uniform for all primes tested and from Sections 3.1 and 3.2 it is

shown that the twisted polynomial generator function is consistently producing a majority of connected graphs, as well as some small fraction of graphs that are unconnected yet still contain all but two or three vertices of the entire field F_p.

p	C_{Max}	SC_{Min}	SC_{Max}	SC_{Avg}	p	C_{Max}	SC_{Min}	SC_{Max}	SC_{Avg}
5	2	2	2	2	101	2	2	3	2.04
7	2	2	2	2	149	2	2	3	2.03
11	2	2	3	2.33	199	2	2	2	2
13	2	2	3	2.25	293	2	2	3	2.01
17	2	2	6	3.17	397	2	2	3	2.01
19	2	2	3	2.20	499	2	2	3	2.01
23	2	2	2	2	599	2	2	2	2
29	2	2	3	2.13	691	2	2	3	2.01
31	2	2	6	2.50	797	2	2	3	2.01
37	2	2	3	2.10	887	2	2	2	2
41	2	2	2	2	997	2	2	3	2
43	2	2	3	2.09	1249	2	2	2	2
47	2	2	2	2	1499	2	2	3	2
53	2	2	3	2.07	1753	2	2	2	2
59	2	2	3	2.07	1999	2	2	2	2
61	2	2	3	2.06	2477	2	2	3	2
67	2	2	3	2.06	2999	2	2	2	2
71	2	2	2	2	3499	2	2	3	2
73	2	2	2	2	3989	2	2	3	2
79	2	2	2	2	4493	2	2	3	2
83	2	2	3	2.05	5003	2	2	3	2
89	2	2	2	2	10007	2	2	2	2
97	2	2	2	2	20011	2	2	3	2

Table 3.2: Unconnected graph characteristics per prime number.

3.4 Giant Component Graphs

Following an asymptotic analysis from Sections 3.1 and 3.2, the exceptions to the general case are investigated and characterized where possible. Knowing when these graphs will not behave as the general case is important for applications. For instance, there will be p unconnected graphs associated with any prime p. For larger primes, the percentage of all graphs for the prime p that are unconnected will become insignificant due to the total number of graphs growing quadratically.

First the obvious exceptions to the general case of unconnected graphs are investigated, where there are three cases where there are graphs that have a small component greater than 3 vertices. Testing all values for λ, $a \in F_p$ for all $p \leq 5003$ it is found that only three graphs that have a small component greater than three and these are shown in Figures 3.3 and

3.4. While interesting, given that these graphs are found at small values of p the generator stabilizes quickly as p becomes large and these exceptions disappear.

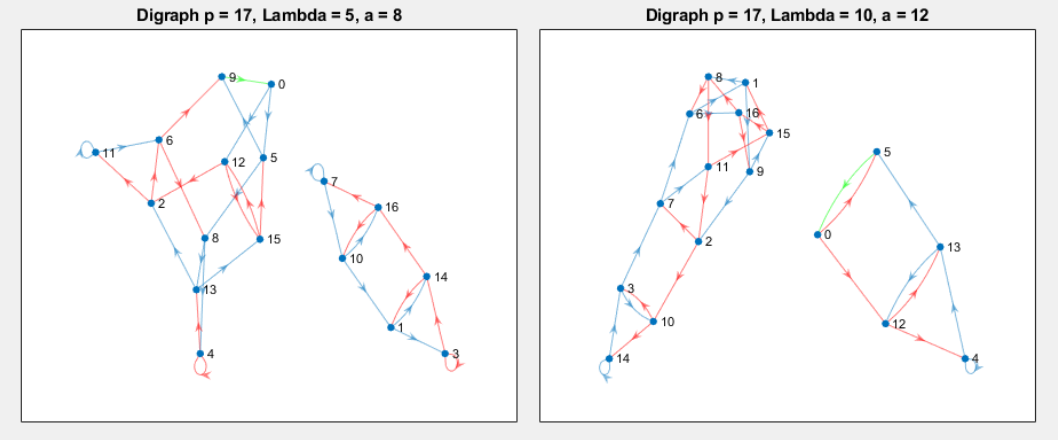

Figure 3.3: Exceptional digraphs for $p = 17$ where small component size over 3

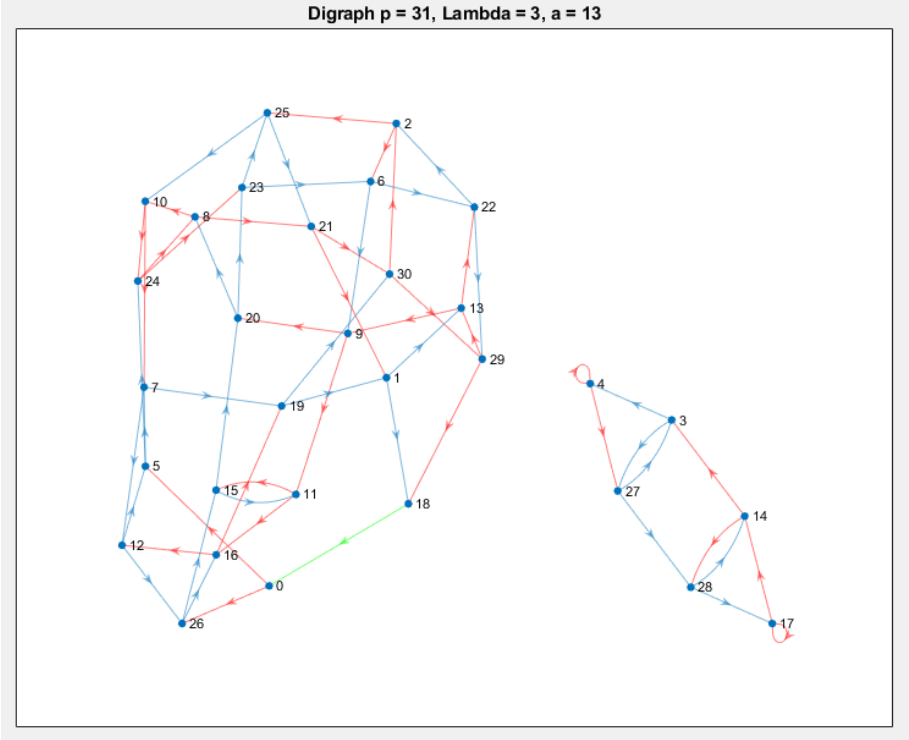

Figure 3.4: Exceptional digraph for $p = 31$ where small component size over 3

The small component for the remainder of unconnected graphs were found to always be three vertices or less, so the other component must be nearly the size of p. Once again looking at the asymptotic case, as prime p becomes large, from Table 3.2 the average smallest

component of unconnected graphs (SC_{AVG}) approaches 2. So the largest component of the two will be $p-2$ vertices on average for large p. This is important and useful as it means the maximum cycle size in the largest component is still close to p vertices. It may become insignificant for larger p as the proportion of unconnected graphs is linear with p where as the total number of graphs is proportional to p^2.

[Mans et al., 2019] characterized functional graphs for the case when the smallest component is of size three. A small component of size three is a special case where $\lambda = 2$ and $a = 1$ or when $\lambda = 1/2$ and $a = 2$. First note that these are isomorphic as discussed earlier where functional graphs have a conjugate pair of λ and λ^{-1}, so in our analysis there will only be one of these as the conjugate graph will have been eliminated in the isomorphic reduction of tested λ values per prime p. Secondly, these conditions will only arise when $\lambda = 2$ is not quadratic residue. Interestingly they will also produce the same small component of $[0, \pm 1]$, or $[0, \pm 2]$ for $\lambda = 2$ or $1/2$ respectively. Figure 3.5 shows this condition for both $\lambda = 2$ and the inverse which in this case for $p = 11$, $\lambda = 6$ as $6 \times 2 = 1 \mod 11$. Note that the large component edges are reversed, but otherwise the two graphs are clearly isomorphic. Also note that the inverse path has the opposite edge colouring, corresponding with an inverted function path.

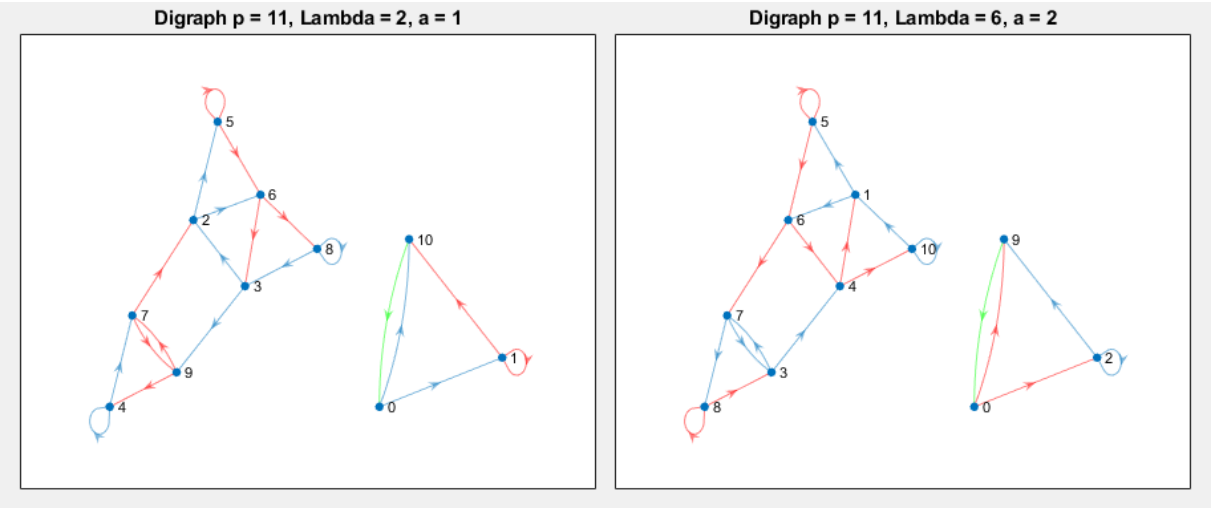

Figure 3.5: Graph of $p = 11, \lambda = 2, a = 1$ and $\lambda = 1/2 \mod 11, a = 2$

The smallest component of an unconnected graph of size 2 can also be predicted, once again from [Mans et al., 2019], there will be an unconnected graph with a component of size

two where the conditions $\lambda \neq -1$ and $a = 2(\lambda+1)/(\lambda-1)^2$ are met. Moreover, the vertices that make up the component will be $2/(1-\lambda)$ and $2/(\lambda-1)$. As an example of this, using $p = 11$ again the condition for an unconnected component of size two is met by the variables $\lambda = 2$ and $a = 6$. This creates a small component with the vertices 2 and 9, i.e. $\pm 2 \mod 11$. This holds true using the visualization tools to create Figure 3.6, and it is easy to follow the calculations through to get the resulting graph features from the proposition.

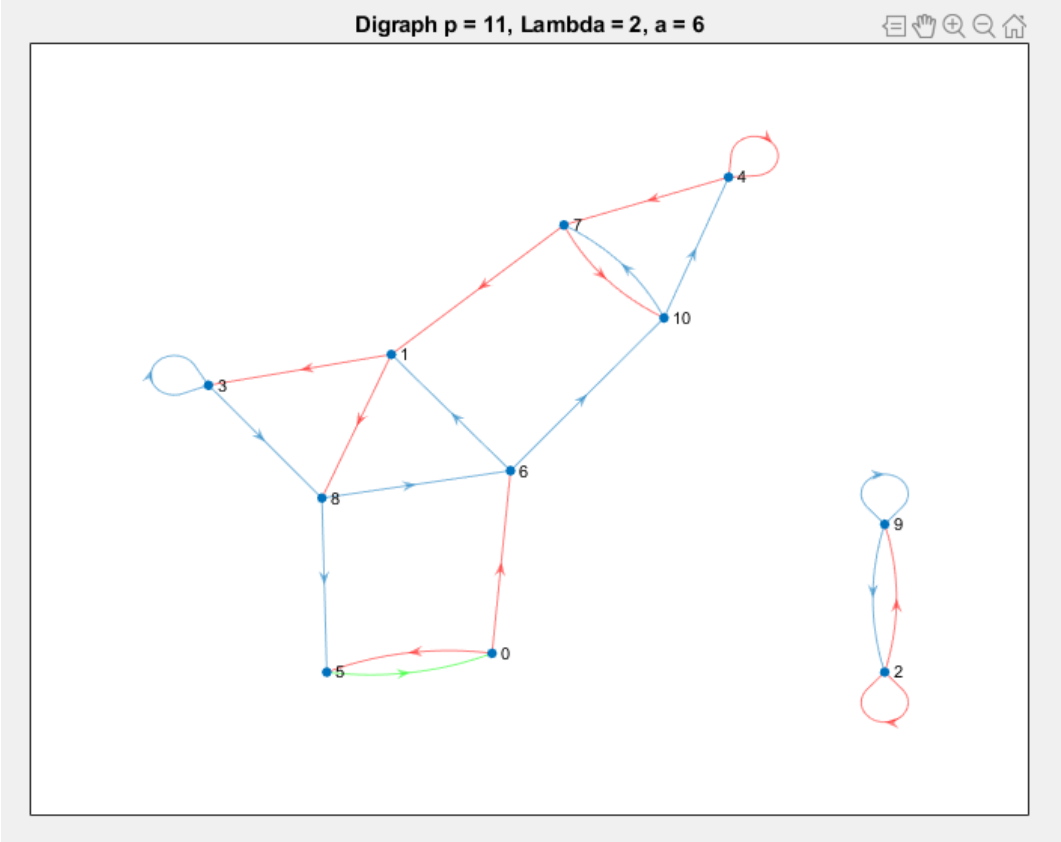

Figure 3.6: Graph of $p = 11, \lambda = 2, a = 6$ demonstrating small component of two

4 HAMILTONIAN CYCLE ANALYSIS

The form of these functional graphs is known from analysis in Section 3 and it is expected in almost all cases a strongly connected, single component graph and also what to expect from the few cases that are unconnected. Here an algorithm is designed to count the Hamiltonian cycles and analyze the sequences generated from these graphs. The details of why this is an important feature is discussed in Section 1.4 of this thesis.

4.1 HAMILTONIAN CYCLE COUNTING ALGORITHM

Running a short computational analysis for all connected graphs for some small primes found that there was at least one Hamiltonian cycle for each graph tested. Once this was established, a count of the Hamiltonian cycles for all graphs $G(\lambda, x+a)$ for $a \in F_p$ where $a > 0$, for small p was conducted as an asymptotic analysis. The initial plan to exhaustively find all Hamiltonian cycles for all graphs was proven to be out of reach even for small primes. The expectation of the number of Hamiltonian cycles was exceeded by far, and is much too large to allow complete characterization. The functional graphs generated an astonishing number of Hamiltonian cycles per graph and the number of Hamiltonian cycles was increasing exponentially with the size of the primes.

The algorithm was then optimized as shown in Algorithm 3 using a recursive backtracking depth-first search, built as a new algorithm that exploits the consistent two in-bound and two out-bound edges. This algorithm would normally cause a risk of creating a stack overflow for larger primes, however as large primes are not able to be processed this algorithm can be safely used. Another refinement was to eliminate the double counting of Hamiltonian cycles due to the final element always having two paths to the zero element which is achieved by testing the vertex path for a length of length of $p-1$, and for the second to last vertex which is easy to determine from the edge table (or more specifically the mapping table as shown in Figure 2.2. This change in algorithm produced a 60% improvement in processing speed when measured on single threaded performance with the main saving expected to be due to the recursive backtracking nature that can eliminate many vertex permutations without having to reference stored sub-paths in memory.

Given the edge table from Section two as an input, a vector of Boolean values is built as a flag for each vertex in F_p as a quick reference to prevent reprocessing assigned vertices. Another vector is used to hold the vertices in order of visit along the path generated by suc-

Algorithm 3 Hamiltonian Cycle Counting (Recursive)

Require: EdgeTable, Visited[], Path[], HC_c
if Path[].length = Hamiltonian Length and Path[Last Node] = Origin **then** ▷ Condition for Hamiltonian Cycle
    $HC_c \Leftarrow HC_c + 1$
    return ▷ After HC found, terminate recursion
for $i = y_0$[Path[Last Node]], y_1[Path[Last Node]] **do** ▷ For both next edges from edge table
    if !Visited[i] **then** ▷ Process if not already in path
        Path[] $\Leftarrow i$ ▷ Push vertex to path
        Visited[i] $\Leftarrow$ True ▷ Set the visited flag to true
        Call Recursive (EdgeTable, Visited[], Path[], HC_c) ▷ Recursive with new variables
        Path[] $\Leftarrow$ Clear last element ▷ Backtrack up the stack, pop last vertex in path
        Visited[i] $\Leftarrow$ false ▷ Backtrack up the stack, reset flag to not visited
return HC_c

cessive recursive calls and a pointer to a counter variable to be incremented by any instance of recursion that returns true for a Hamiltonian cycle. As a diagnostic, any Hamiltonian cycle found is printed to the command line.

For every vertex from the zero vertex, there are two possible paths as defined from the edge table. Each possible next vertex is tested to see if it has already been visited, and if not is submitted as a new recursive instance with the current path extended for the tested vertex, and an updated visited vector. This process ends for two conditions, where either (i) a Hamiltonian cycle is found when the recursive path is p elements in size and ends at the origin of the cycle, in this case it is always zero as all tests use 0 as the origin or (ii), the path is eliminated when encountering a recurrent vertex before the expected length of the cycle. The final vertex is determined from the $p-1$ element as due to the zero condition both edges from that vertex will go to zero and save one step from the processing with the rationale this reduces the possible paths to $2^{(p-1)}$.

4.2 Hamiltonian Cycles on Connected Graphs

The primary interest in Hamiltonian cycle analysis is to get an idea of how successful the choice of generator function is for generation of Hamiltonian cycles. An initial depth-first search algorithm was developed to test for Hamiltonian cycles in all functional graphs for small primes. Tests were limited to connected graphs, which as shown previously were the vast majority of the graphs per prime. Also, as for previous tests nearly half of the λ values were excluded and hence roughly half of the functional graphs due to isomorphism, as well as the exclusion of any graphs where $a = 0$.

This proved to be an efficient test, where every graph up to 101 had at least one Hamiltonian cycle. This was an excellent result as it proved the choice of functions did produce

Hamiltonian cycles and it was possible that a Hamiltonian cycle would be found for all connected graphs generated this way. This prompted theoretical speculation that lead to a proof as discussed in [Mans et al., 2019] where the strongly connected nature of the graphs generated caused all connected graphs to have a Hamiltonian cycle. Further to this, it is proven that any connected components of unconnected graphs will too have at least one Hamiltonian cycle.

The code that leveraged the cycle detection algorithm was quickly written to test all connected graphs for a prime number and a test function of a simple brute-force, depth-first search was implemented to count the number of Hamiltonian cycles. The results of this test are in Table 4.1. Testing of the multi-threaded performance against single thread performance for prime $p = 41$ showed a speed multiple of approximately 25 times when run over 28 threads which gives an efficiency of approximately 90 percent for processor utilization.

Asymptotic analysis was applied to the Hamiltonian cycles generated by connected graphs, with the results shown in 4.1 for the minimum, maximum and average number of Hamiltonian cycles counted per graph for each prime number. Clearly from Table 4.1 the average of Hamiltonian cycles per graph grew exponentially with p and quickly came to a point where calculations became unfeasible, even with the use of a much more powerful workstation. Using regression analysis from fitted trend lines the average number of Hamiltonian cycles was related to the prime p by $0.3e^{0.3p}$. This result is better than expected in terms of finding Hamiltonian cycles as each graph could supply a great deal of Hamiltonian cycles.

There are so many Hamiltonian cycles that profiling large primes is not feasible. With such a large number of Hamiltonian cycles, reducing the brute force algorithm to a efficient algorithm becomes difficult, and from discussion in Section 1.4 of this thesis, it is a computationally intensive problem and without some efficient refinement from the general case remains unfeasible for larger primes. An un-optimized, brute-force, depth first search will have 2^p possible paths and even with some small improvements to the algorithm, you can see from 4.1 that the time taken to count Hamiltonian cycles is also exponential. It is clear at this point that running multi-threaded analysis is pointless as once an unfeasible prime is reached for processing a factor of 28 threads is not going to take the analysis much further than just the next prime number.

Algorithm 3 was developed as discussed in the previous section, to optimize Hamiltonian cycle counting tools for small primes which uses recursive back tracking. Since large primes are already excluded by complexity the stack depth for recursion is not going to be large enough to cause a stack overflow.

While counting Hamiltonian cycles with such a large number per graph will be unfeasible for even slightly larger primes, it is clear that the generator function Equation (1.3) produces a large number of Hamiltonian cycles per graph. Looking at the maximum and minimum Hamiltonian cycle count there is some anomaly for prime $p = 31$, where the minimum Hamiltonian cycle count is less than the previous prime number. By inspecting the maximum and minimum graph for prime $p = 31$ as shown in Figure 4.1 there are clear differences in form. This is likely due to the relatively small prime numbers analyzed and this trend will likely stabilize for larger p.

p	Max HC	Min HC	AVG HC	# Graphs	Processing Time
5	2	1	1.34	3	-
7	3	1	1.55	11	-
11	10	1	3.67	27	-
13	22	1	6.32	32	-
17	72	1	18.31	58	-
19	148	4	34.04	85	-
23	423	5	93.70	127	-
29	2840	34	666.14	188	-
31	5410	30	1206.09	232	-
37	45546	448	7906.60	314	3 seconds
41	175428	1223	28473.25	390	13 seconds
43	255558	2222	53999.00	451	33 seconds
47	1273729	6576	195723.00	541	3 minutes
53	6795031	63363	1297780.00	662	36 minutes
59	52305140	353112	-	855	9.3 hours

Table 4.1: Hamiltonian Cycle count results using modified DFS, over 28 threads.

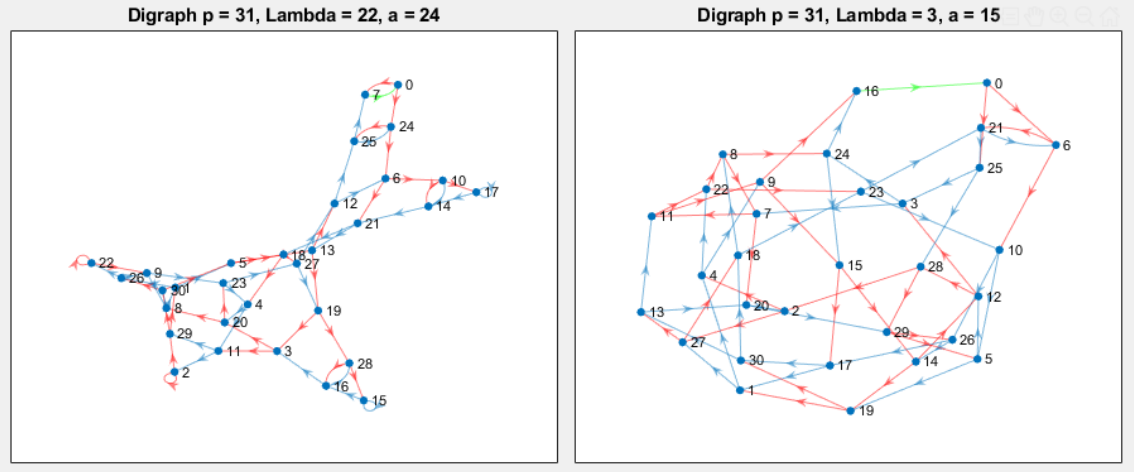

Figure 4.1: Comparison of graphs with minimum and maximum Hamiltonian cycle count for $p = 31$.

By inspecting the maximum and minimum graph for prime $p = 31$ as shown in Figure 4.1 there is a clear difference in form. The graph $G(22, x + 24)$ on the left has only 30 distinct

Hamiltonian cycles when compared with $G(3, x+15)$ which has 5410 distinct Hamiltonian cycles. The maximum case shows a homogeneous graph with little in the way of exceptional features. The minimum case has several graph features that restrict possible variations of Hamiltonian cycles by forcing the cycle through the same sequence of vertices. For there to be a Hamiltonian cycle in the minimum graph, there must be a sequence $12 \rightarrow 25 \rightarrow 7 \rightarrow 0 \rightarrow 24 \rightarrow 6 \rightarrow 10 \rightarrow 17 \rightarrow 14 \rightarrow 21$ and another sequence of $19 \rightarrow 28 \rightarrow 15 \rightarrow 16 \rightarrow 3$. This is limiting half of the vertices on the graph and so the possible permutations that can be Hamiltonian cycles are reduced.

4.3 Hamiltonian cycle binary sequences

Analysis has shown there are many Hamiltonian cycles per connected graph and that most graphs for 1.3 are connected. As a source of random binary numbers it might be possible to use the choice of twisted or untwisted sub-equation from Equation (1.3), i.e. using C_0 indicates a 0 and C_1 indicates a 1. This transforms the Hamiltonian cycles from these functional graphs into binary sequences from the choice of generator function used for each edge transition from vertex to vertex on the graph. These binary values are referenced from the edge table generated by Algorithm 1.

The Hamming weight of a binary sequence is the decimal value for the sum of the binary elements. Calculating the Hamming weight for each sequence, starting at vertex 0, the sum of the zeros and ones was always $p/2$ or $(p/2)+1$. On inspection of the graphs it is easy to see that this is due to the root of $0 \mod p$ is always zero, and both equations have a solution and as with previous sections, the special case for zero is highlighted with a 2. Excluding the final edge on the Hamiltonian cycle, the hamming weight is always $(p-1)/2$ for all $p \in P$, so this clearly shows the binary sequences are always balancing. This is also proven in [Mans et al., 2019].

	Hamiltonian Cycle	Function Path	Hamming Weight
p:7 ,λ:3, a:1	0 1 3 5 4 2 6 0	0001112	3
	0 1 4 5 3 2 6 0	0011012	3
p:7, λ:3, a:3	0 1 2 5 6 3 4 0	1010012	3
p:7, λ:3, a:4	0 2 4 6 1 5 3 0	0101102	3
	0 5 4 6 1 2 3 0	0001112	3
p:7, λ:3, a:5	0 5 1 4 3 6 2 0	1110002	3
p:7, λ:3, a:6	0 3 4 6 5 2 1 0	1011002	3
p:7, λ:6, a:1	0 1 4 3 2 5 6 0	0010112	3
p:7, λ:6, a:2	0 3 4 6 1 2 5 0	0110102	3
p:7, λ:6, a:3	0 2 3 1 5 6 4 0	1110002	3
	0 5 1 2 3 6 4 0	1001102	3

	0 5 6 3 1 2 4 0	1001012	3
p:7, λ:6, a:4	0 2 1 4 6 5 3 0	0110102	3
	0 2 6 5 4 1 3 0	0110012	3
	0 5 4 6 2 1 3 0	0001112	3
p:7, λ:6, a:5	0 4 3 1 6 5 2 0	1001012	3
p:7, λ:6, a:6	0 6 3 4 5 2 1 0	1101002	3

Table 4.2: Hamiltonian Cycle paths, Function Sequences and Hamming weights for p = 7.

An example of function paths is shown in Table 4.2. For all connected graphs where $p = 7$ displaying all Hamiltonian cycles per graphs, the function path created by these Hamiltonian cycles and the Hamming weight or sum of the function path. The edges returning to zero are labeled as before with a 2 to mark the special case and excluded this edge from analysis as it is both a zero and a one. Notice that the Hamming weight for all paths is the same for all graphs and cycles, and is always $(p-1)/2$. As described in [Mans et al., 2019], this finding matches the theoretical expectations where there are an equal number of edges from both the twisted and untwisted functions so the sequences generated from this process from the Hamiltonian cycles must be balancing.

4.4 Type 1 Hamiltonian Cycles

Function sequences will be a balanced set of zeros and ones and by limiting the function sequences to certain patterns the number of valid Hamiltonian cycles will be reduced. The most restrictive test is for any Hamiltonian cycle that creates a functional sequence that has no repeated value for example 010101....0101, or a sequence of 1. There can only be one of this special case of Hamiltonian cycle per graph where the inverse of the sequence is an isomorphism when starting at the zero vertex on the graph. This is due to the graph feature that two outbound edges from a vertex are always either both from the twisted or both from the untwisted function so will both be zeros or both be ones.

The terminology from [Mans et al., 2019], definition 2.6 is used here. A Hamiltonian cycle H of a connected component of a graph $G(\lambda, f)$ is said to be of Type n (n is a positive integer) if in the cycle H there exist n consecutive edges (not including the edge going to the vertex 0) arised from the same equation (either $Y^2 = f(X)$ or $\lambda Y^2 = f(X)$), and there are no such $n+1$ consecutive edges.

This analysis restricts the possible path through the graph that is permitted, and so Algorithm 3 quickly eliminates possible paths and so become much more efficient to complete.

However some additional logic is needed to track the length of each sequence as the sequence changes from a 0 to a 1 or a 1 to a 0. This is achieved by adding code to the recursive algorithm to check the last n elements on the cycle path that correspond to the type of the Hamiltonian cycle tested. For example a Type 1 Hamiltonian cycle would be testing for the last vertex on the path and then looking at the next to see if the function path condition is breached for the type of the Hamiltonian cycle and if so does not submit a new recursive call to continue the depth first search.

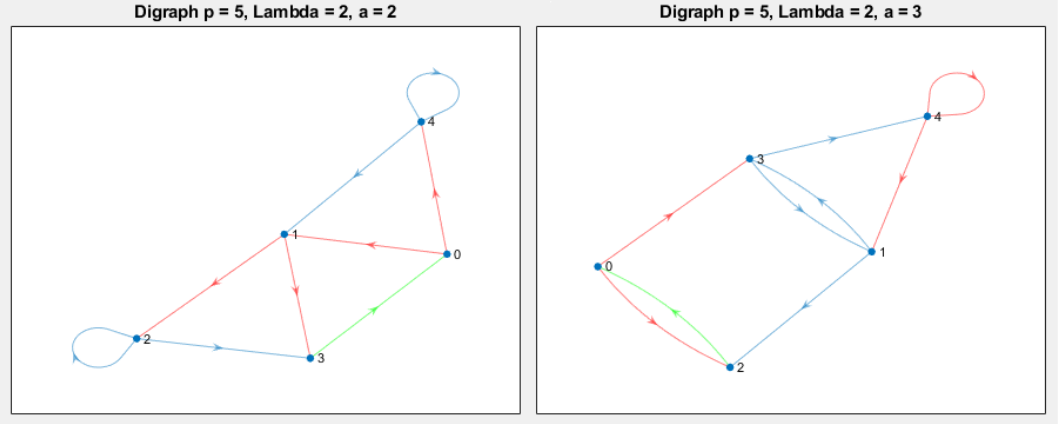

Figure 4.2: Digraphs for $p = 5$ with Hamiltonian Cycles of Sequence 1.

Testing all connected graphs for $p < 1000$, with the exception of $p = 5$ there were no connected graphs with Type 1 Hamiltonian cycles. Note that it is feasible to test for much larger primes than was possible for a full Hamiltonian cycle search, due to the quick elimination of viable paths for a Type 1 Hamiltonian cycle. The two exceptions are shown in the following Figures 4.2. These exceptions are due to the small size of p (i.e. the size of the field F_p) and hence the Hamiltonian cycle is not large enough to encounter a feature that forces the function sequence to repeat an element for the Type 1 cycle. This system quickly stabilizes as the size of the field increases and the opportunity for a Type 1 Hamiltonian cycle becomes improbable. From this result a theoretical proof was discovered as described in [Mans et al., 2019] for $p > 17$ there can be no graphs with Type 1 Hamiltonian cycles.

4.5 Type 2 and Type 3 Hamiltonian Cycles

Extending the type of Hamiltonian path to sequences of length 2 and 3 using the same process as for Type 1 Hamiltonian cycles, but increasing the sequence limitation to match

the type. Results are shown in Table 4.3 for both Type 2 and Type 3 Hamiltonian cycles, and for each case the percentage of graphs that have a Type 2 or 3 Hamiltonian cycle is tabulated, and also the average number of these Hamiltonian cycles per graph where the graph has at least one valid cycle of the type tested.

Type 2 Hamiltonian cycles are tested for $p < 100$ however the test for Type 3 Hamiltonian cycles becomes unfeasible quickly and only the results for $p \leq 61$ are possible. This is clearly due to the number of Type 3 Hamiltonian cycles found and hence the paths becoming less restricted and approaching the amount of work required for a search of all Hamiltonian cycles.

p	% Graphs Type 2	Avg HCs Type 2	% Graphs Type 3	Avg HCs Type 3
5	100	1.33	100	1.33
7	90.90	1.20	100	1.55
11	77.78	1.76	100	2.81
13	81.25	2.00	96.88	4.23
17	75.86	2.41	98.28	9.16
19	74.12	3.30	98.82	13.86
23	57.48	3.79	97.65	27.96
29	61.17	8.30	98.94	130.59
31	56.47	8.26	98.71	195.10
37	59.24	17.73	99.04	765.26
41	44.62	39.13	98.97	2054.62
43	42.79	32.31	99.11	3310.64
47	38.44	60.58	95.01	7960.04
53	43.96	107.67	99.09	35552.80
59	32.28	255.79	99.30	151503.00
61	38.91	322.48	99.21	247297.00
67	26.24	587.74	-	-
71	28.24	1042.34	-	-
73	39.98	1353.39	-	-
79	21.09	3027.37	-	-
83	20.05	4480.94	-	-
89	24.66	10407.50	-	-
97	24.52	31778.40	-	-

Table 4.3: Hamiltonian Cycle count for Type 2 and 3 Hamiltonian cycles.

The occurrence of graphs with both Type 2 and Type 3 Hamiltonian cycles decreases as p increases, though at different rates. This looks to be due to the likelihood of longer function sequences being generated increases with the length of the cycles in a similar way to that of a coin toss. The average number of Type 2 and Type 3 Hamiltonian cycles found (averaging only graphs that do have these cycles) also show a similarly rapid growth to that of all Hamiltonian cycles however with far less than that of all Hamiltonian cycles.

5 Discussion

5.1 Computational Analysis

Efficient algorithms for edge table generation, component detection and Hamiltonian cycle detection are developed as analysis tools for functional graphs of polynomials and their twist. Analysis has been demonstrated in this thesis for the linear case of $f(x) = x + a$, however the algorithms can be applied to any polynomial $f(x)$ and substituted into Equation (1.2). As the functional graphs were found to produce two-regular digraphs, developed algorithms could be made to consider this type of graph to minimize the computational effort required.

Computational analysis applying these developed algorithms was used to guide theoretical analysis by uncovering the various features of the functional graphs studied and also to further test the relevant theoretical conjectures. Throughout Sections 2, 3 and 4 there are comparisons of analytical results to theoretical proofs cited in a collaborative paper to be submitted [Mans et al., 2019], implementing a methodology outlined in [Flajolet and Odlyzko, 1990] as discussed in Section 1.2.

The functional graph analysis conducted in this thesis is iterative due to the combinatorial approach to asymptotic analysis: first testing on small instances before increasing the size and optimizing the algorithms and their implementation. Multi-threaded techniques are useful for this type of analysis, particularly when the computational complexity is a result of the number of the combinatorial iterations. It is not however, as demonstrated by the Hamiltonian cycle counting in Section 4 of this thesis, necessarily useful when the complexity of individual combinatorial tests become complex. With the availability of many-cored processors and cluster computing facilities, multi-threading and parallel processing are yet becoming increasingly useful tools for number theoretic analysis.

5.2 Functional Graphs of Polynomials and their Twist

The computational analysis of functional graphs conducted in this thesis showed a number of useful features with the introduction of the new generator function shown in Equation (1.3). Mostly connected graphs are produced, the graphs are strongly connected and with the exception of the in-edges to the zero vertex the connected graphs are regular degree two digraphs resulting in all connected graphs and all unconnected components being Hamiltonian. Previous literature [Mans et al., 2017, Konyagin et al., 2016] where the generator $f(x) = x^2 + a$ produced out-degree one digraphs as discussed in Section 1.2 often

produced unconnected graphs, small cycles and many trees, often with tree size of one vertex. By contrast, these functional graphs of polynomial functions and their twist have produced connected graphs with no trees, allowing our average cycle size to approach the size of the finite field p as primes become large, an improvement from the unmodified quadratic case from literature of $\sqrt{p}$ which is useful for sequence generating applications.

Expectations of generating large Hamiltonian graphs have been exceeded and their number is growing exponentially with the size of the prime p. The initial expectation was that the graphs were likely to contain one Hamiltonian cycle however it was discover that many Hamiltonian cycles were present in each graph. The objective of the Hamiltonian cycle analysis changed from the detection of Hamiltonian cycles to profiling the number of Hamiltonian cycles present and analyzing their characteristics.

These graphs of degree-two are of higher degree than the degree-one functional graphs of previously studied polynomial generators. This increases likelihood of connected graphs and Hamiltonian cycles over that of previously studied polynomial generators. Also, the graph components are all strongly connected which is surprising as the degree of two is sparse, typically sparse graphs are not strongly connected. The combination of these features allows for many permutations of vertices for Hamiltonian cycles, while still being much smaller than the total number of possible permutations 2^p.

5.3 Binary Sequences from Polynomials and their Twist

The unanticipated large number of Hamiltonian graphs became a problem for our asymptotic analysis and became unfeasible for primes over $p = 101$. There is an exponential increase in the number of Hamiltonian cycles generated for each graph and is too many count rapidly. By limiting the number of Hamiltonian cycles to Type 2 or 3, there is a reduction in the number of Hamiltonian cycles considered and there is a processing saving for detection and creating a limited set of Hamiltonian cycles may be useful for some applications. However, the number of Hamiltonian cycles of the type defined and tested still produced an increasingly large number that was exponentially proportionate to the size of the finite field. While the instances of prime numbers that were feasible was small, the relationship between the field size and the exponentially large number of Hamiltonian cycles produced is clear.

There was some downward trend in the percentage of graphs that contained Type 2 and 3 Hamiltonian cycles and for larger primes this may well impact the number of graphs per type as the finite field grows in size. This will be in part due to the probability of a larger type (i.e. a larger functional sequence) of Hamiltonian cycle being present in a graph increasing

with the length of the cycle. The processing resources required for this experiment however are prohibitive to investigate at this stage.

5.4 Other notable Features and applications

Where the application of the Hamiltonian cycle as a cryptographic hashing function was suggested in theory as discussed in Section 1.6, the presence of multiple Hamiltonian cycles increases their interest. Having many Hamiltonian cycles per graph means that there are many sequences that are valid for a given functional graph and so decreases the difficulty of finding a Hamiltonian cycle in an attempt to break the cipher.

The number of Hamiltonian cycles grows exponentially, but not as exponentially as the complexity of testing all paths in a functional graph. As the finite field grows large, the number of Hamiltonian cycles will still be relatively small and so will be still difficult to discover via a brute-force search. Applying the polynomial and twist generator as a hashing function will still require the detection of Hamiltonian cycles for large primes as hashes and so will need a way of finding a Hamiltonian cycles to use as keys for a given graph defined by the parameters of the generator function which would be a large prime p, and suitable λ and a. This may well be done via some Heuristic method as discussed in Section 1.4.

A much simpler application is the use of the function paths as balancing binary sequences of Hamiltonian cycles generated as discussed in Section 4.3 of this thesis. The Hamiltonian cycles still need to be found and this again might be done with some heuristic method. While this thesis has discovered the balancing nature of these sequences and the sequences do appear random, further study is required on larger sequences from larger primes is required to establish how well these sequences compare with truly random sequences.

5.5 Future Work

These polynomial generator functions have only been characterized for the linear case in this thesis. A natural extension of this work would be to substitute the linear function $f(x) = x + a$ for higher order polynomials or other functions for $f(x)$ to determine if the effect of the twist and higher degree are consistent and to characterize these generators. Other possibilities are to look at different types of modifications that might lead to Hamiltonian graphs with less Hamiltonian cycles, or other features that enhance applications of functional graphs that leverage the strongly connected functional graphs of small degree produced in this thesis.

The Hamiltonian cycle analysis conducted in this thesis was interested in asymptotic

analysis focused on counting Hamiltonian cycles on all permutations of variables per prime. This limited the study to small primes due to the complexity of the problem and the number of Hamiltonian cycles produced per graph by this generator. To get a real understanding of the randomness of the binary sequences generated from the function path of a Hamiltonian cycle, the sequences need to be much larger. While this is difficult to achieve using the algorithms implemented in this thesis, a heuristic method as discussed in Section 1.4 of this thesis could be developed and run more quickly allowing the capture of a single or small number of Hamiltonian cycles per graph that could then be analyzed and compared to random maps.

Further investigation of multi-threading and parallel processing for number theoretic analysis may also be of interest. As computing resources are increasingly being built with more and more processors and cores, code will have to be written with these techniques to take advantage of the new technology. Scientific processing becomes problematic however, due to the same type of task being run over all cores which can be difficult to optimize for the different processors and resources that the processors have available. As observed in this thesis, processors running multi-threaded code across all cores tended to perform bellow their full potential with varying effect due to compilers and memory resources of the different processors. Further study into this may well improve computational techniques for number theoretic analysis and scientific processing more generally.

6 Conclusion

In this thesis we have developed new, efficient algorithms to conduct a computational analysis of functional graphs of polynomials and their twist over finite fields for the linear case of $f(x) = x + a$, however the algorithms developed are applicable to all polynomial functions $f(x)$. The algorithms are further optimized using multi-threading to take advantage of the iterative nature of the analysis and utilize processors with many cores that are available with modern computers. While these optimization are useful, the complexity of the analysis limits this study to smaller primes, particularly for the Hamiltonian cycle analysis.

The computational analysis of the functional graphs produced by this generator revealed that nearly all functional graphs are strongly connected graphs, and the few unconnected graphs have a giant, strongly connected component consisting of all but two or three vertices in the finite field F_p. A surprising result is that though the functional graphs produced are all 2-regular graphs, these sparse graphs and components are all strongly connected.

Cycle analysis of these functional graphs also discovered that all connected graphs and unconnected components are Hamiltonian. In fact, all connected graphs have many Hamiltonian cycles, the number of which is exponentially proportionate to the size of the finite field. A significant result of this analysis is that the size of the cycle is a one to one mapping of the finite field, which much larger than that of other well-known random mappings where the cycle length is close to $\sqrt{p}$. These properties may be useful for cryptographic applications such as using Hamiltonian cycles for cryptographic hashing.

In addition to these findings, a feature of the twist property of the generator allows the creation of balancing binary sequences from the use of the twisted or untwisted polynomial. These sequences are then categorized into different types of Hamiltonian cycles by the sequences generated and further characterized as a way of limiting the number of Hamiltonian cycles per functional graph. These balancing binary sequences may be useful for pseudo-random number generation applications.

Experimental results were often limited to smaller primes due to complexity however these findings were enough to guide theoretical proofs which were referenced throughout this thesis. Between the experimental and theoretical results, the features of the functional graphs are well characterized and understood. Further study of polynomials and their twist focusing on higher order polynomials will help to better understand the effects of this technique beyond the linear case.

REFERENCES

[Anthony Williams and Vicente J. Botet Escriba, 2017] Anthony Williams and Vicente J. Botet Escriba (2017). boost C++ Libraries.

[Baniasadi et al., 2014] Baniasadi, P., Ejov, V., Filar, J. A., Haythorpe, M., and Rossomakhine, S. (2014). Deterministic "Snakes and Ladders" Heuristic for the Hamiltonian cycle problem. *Mathematical Programming Computation*, 6(1):55–75.

[Blum et al., 1986] Blum, L., Blum, M., and Shub, M. (1986). A Simple Unpredictable Pseudo-Random Number Generator. *SIAM Journal on Computing*, 15(2):364–383.

[Brent, 1980] Brent, R. (1980). An improved Monte Carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184.

[Diffie and Hellman, 1976] Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.

[Flajolet and Odlyzko, 1990] Flajolet, P. and Odlyzko, A. (1990). Random Mapping Statistics. *Lect. Notes Comput. Sci.*, 434:329–354.

[Frieze, 1988] Frieze, A. (1988). An algorithm for finding hamilton cycles in random directed graphs. *Journal of Algorithms*, 9(2):181–204.

[Frieze and Haber, 2015] Frieze, A. and Haber, S. (2015). An almost linear time algorithm for finding Hamilton cycles in sparse random graphs with minimum degree at least three. *Random Structures & Algorithms*, 47(1):73–98.

[Hefetz et al., 2016] Hefetz, D., Steger, A., and Sudakov, B. (2016). Random directed graphs are robustly Hamiltonian. *Random Structures & Algorithms*, 49(2):345–362.

[Konyagin et al., 2016] Konyagin, S. V., Luca, F., Mans, B., Mathieson, L., Sha, M., and Shparlinski, I. E. (2016). Functional graphs of polynomials over finite fields. *Journal of Combinatorial Theory, Series B*, 116:87–122.

[Kühn and Osthus, 2012] Kühn, D. and Osthus, D. (2012). A survey on Hamilton cycles in directed graphs. *European Journal of Combinatorics*, 33(5):750–766.

[Lagarias, 1985] Lagarias, J. C. (1985). The 3x + 1 Problem and Its Generalizations. *The American Mathematical Monthly*, 92(1):3–23.

[MacFie and Panario, 2012] MacFie, A. and Panario, D. (2012). *Random Mappings with Restricted Preimages*.

[Mans et al., 2017] Mans, B., Sha, M., Shparlinski, I. E., and Sutantyo, D. (2017). On Functional Graphs of Quadratic Polynomials. *Experimental Mathematics*, pages 1–9.

[Mans et al., 2019] Mans, B., Sha, M., Smith, J., and Sutantyo, D. (2019). On The Functional Graph of a Plane Curve and its Twist. *Manuscript*.

[Merkle and Hellman, 1978] Merkle, R. and Hellman, M. (1978). Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24(5):525–530.

[NCI, 2018] NCI (2018). National Computational Infrastructure, Australian National University, Canberra ACT 2601, Australia. *http://nci.org.au/*.

[Pollard, 1975] Pollard, J. (1975). A monte carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334.

[Rivest et al., 1978] Rivest, R., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Shoup, 2019] Shoup, V. (2019). NTL - A Library for doing Number Theory. *http://www.shoup.net/ntl/*.

[Vasiga and Shallit, 2004] Vasiga, T. and Shallit, J. (2004). On the iteration of certain quadratic maps over GF(p). *Discrete Mathematics*, 277(1-3):219–240.