ABSTRACT

Title of Dissertation: NONLINEAR COMPLEXITY OF BOOLEAN PERMUTATIONS

Thomas Gordon Draper, Doctor of Philosophy, 2009

Dissertation directed by: Dr. Lawrence C. Washington
Department of Mathematics

We introduce the concept of nonlinear complexity, where the complexity of a function is determined by the number of nonlinear building blocks required for construction. We group functions by linear equivalence, and induce a complexity hierarchy for the affine equivalent double cosets. We prove multiple invariants of double cosets over the affine general linear group, and develop a specialized double coset equivalence test. This is used to classify the 16! permutations over 4 bits into 302 equivalence classes, which have a maximal nonlinear depth of 6. In addition, we present a new complexity class defined in terms of nonlinearity.

NONLINEAR COMPLEXITY OF BOOLEAN PERMUTATIONS

by

Thomas Gordon Draper

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2009

DEDICATION

To Tyra, Thomas, Lillian, Elizabeth and most of all Susan, who made it all possible.

ACKNOWLEDGEMENTS

Thoughtful consideration revealed many who contributed to my dissertation, in ways large and small.

My parents head the list. From my first memories, they saw my affinity for mathematics and nurtured it. They challenged, inspired, balanced and loved me. I realize more and more how important this was.

My professors and teachers at BYU challenged me further and helped me set a higher standard for myself, both in mathematics and in life. Their greatest contribution to me was helping me to center on God and then to build my life around Him. Interestingly, I realize that much of this communication was made possible because of our common love of mathematics and science.

My job has been one of the greatest experiences of my life. I would never have imagined the satisfaction received from puzzle solving with impact. Again and again I have found myself in the right place at the right time for the right opportunity. I consider myself very blessed and wish everyone could do what they love for a living. In addition, my bosses and coworkers have been extremely supportive of my dissertation work, even when it increased their own burden. I am very grateful for this and I know this is not the norm for the workplace.

I wish to personally thank Dr. Washington and Dr. Gasarch. They were both extremely helpful in directing and focusing my research efforts. I admire them both as researchers and teachers.

Finally, I wish to thank my children and especially my wife. I know we sacrificed much of our time together and I hope it will be worth it. We have a long and exciting road ahead to share together. I hope Tyra, Thomas, Lillian and Elizabeth will each find and enjoy their talents. I am excited to see their lives unfold.

I conclude with an overwhelming appreciation for the love of my life, Susan. She encouraged me when I was tired. She strengthened me when I was weak. She burdened herself to make things easier for me. She loved me. I will always be in her debt. I adore her more than anyone else on this earth. I am looking forward to our many adventures yet to come.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 The Difficulty with Difficulty

In 1996, James L. Massey delivered the IACR Distinguished Lecture at EUROCRYPT '96, entitled "The Difficulty with Difficulty" [Mas96]. This talk, which served as an inspirational starting point for my own research, addressed why measuring the complexity of Boolean functions is so difficult. This thesis presents a new method for measuring complexity and develops tools that allow us to examine a previously unknown structure: the minimal complexity of all $16! = 20,922,789,888,000$ 4-bit Boolean permutations.

In particular, we introduce the notion of nonlinear complexity. We will prove that all permutations may be constructed using alternating rounds of linear and simple nonlinear permutations. In other words, every n-bit permutation p is a composition of the form

$$p = L_0 N_1 L_1 N_2 L_2 \cdots N_n L_n,$$

where each L_i is an n-bit linear function, and each N_i is a simple nonlinear permutation selected from a small set. The **nonlinear complexity** of a permutation will be the minimum number of rounds of nonlinearity needed to build the permutation.

To find the minimal nonlinearity of all 4-bit Boolean permutations, faster double coset identification methods were needed. Special methods for working with $\mathrm{AGL}_n(2)$ double cosets were needed, since general double coset methods would take months to compute the results given in this thesis.

Special Hamiltonian cycles accelerate the double coset equivalence test. Various invariants are used to accelerate double coset identification. The mathematical theory behind each of these advances is developed in this thesis.

Using these new tools, we examine the nonlinear complexity of many common 3-bit and 4-bit permutations.

1.2 Combinatorics and Complexity

Let $V = \mathrm{GF}(2)$, the finite field over two elements. Given a Boolean function $f : V^n \to V$, how difficult is f to compute? Typically, the difficulty of computing f is associated with how many "building blocks" are needed to construct f. One common set is the AND, OR and NOT gates. AND, OR and NOT are a universal set. This means any Boolean function can be constructed from a finite number of them. Such constructions are not unique, but there will be a minimal number of gates needed to construct a function (There may also be multiple minimal solutions).

Minimal solutions are found by exhaustively constructing all possible functions and marking the minimal instances of a function by where they first occur. This method is essentially a combinatorical search. Unfortunately, the number of functions $f : V^n \to V$ is 2^{2^n}. Due to the doubly exponential growth, classifying the $2^{2^5} = 4,294,967,296$ 5-bit to 1-bit functions is a challenge for current desktop computers. If we could classify a billion billion functions (10^{18}) per second, then classifying all $2^{2^7} = 2^{128} = 340282366920938463463374607431768211456$ 7-bit to 1-bit functions would take about 10^{13} years, which is a thousand times longer than the current age of the universe (1.3×10^{10} years).

In response to this massive explosion of complexity, researchers chose to focus on certain equivalence classes of functions.

1.3 History

In 1951, researchers at the "Computational Laboratory of Harvard" exhaustively classified the 402 classes under permutation and complementation of the input variables of the 65,536 four-variable Boolean functions [AtSotCL51]. Afterward, mathematicians realized Polya theory could easily enumerate the number of classes. This sparked a large amount of theoretical interest in counting the number of Boolean functions in various manners [Lor64].

Studying methods for finding optimal or minimal representations for these functions was largely ignored. This is likely due to the fact that look-up tables work very efficiently for small functions.

Quantum computing has forced mathematicians to reconsider the foundations of computer science with the added constraint of reversibility. In the reversible context, it is more natural to study Boolean permutations (n-bit to n-bit invertible functions) as opposed to the more commonly studied Boolean functions (n to 1 bit functions). Many foundational ideas that were thought to be settled have been resurrected. Even lowly addition has once again become a hot topic. In the search for efficient programs on a quantum computer, the exact calculation of small permutations is important. Look-up tables are problematic on a quantum computer since computation space is at a premium. Due to the difficulty of quantum computation, we are willing to spend a great deal of classical computational power to find an efficient quantum program.

This paper presents a new method for categorizing the complexity of a function in terms of the number of rounds of nonlinearity needed to achieve the function. Functions which differ from each other by a linear transformation will be considered to have the same nonlinear complexity. A small set of simple nonlinear permutations will be used to connect the linearly related equivalence classes. By counting the minimal number of nonlinear transitions needed to create a function, we can assign a partial ordering to the equivalence classes.

1.4 Summary of Primary Results

A new theory of complexity is introduced, based on nonlinearity. This new notion of complexity proves useful in comparing the exact complexity of very small functions, but also defines new complexity classes that fit nicely into the current complexity framework of computer science.

We show how Boolean permutations can be separated into double coset equivalence classes based affine transformations. We can then link the double cosets via a special class of nonlinear functions. The nonlinear connections induce a complexity hierarchy among Boolean Permutations.

Previous methods for finding the complexity of Boolean functions focused on counting the minimal number of gates needed to realize a function from a given universal set. The doubly exponential growth of Boolean functions has made research beyond 3-bit Boolean permutations and 5-to-1 bit Boolean functions impractical.

The central contribution of this paper is to count rounds of nonlinearity instead of counting gates. By relaxing the restriction from the fine granularity of gate count to a coarser measure, we can view the complexity structure present in permutations previously too difficult to analyze.

Using the results of this paper, we will revisit the complexity of 3-bit permutations in a new light and present a previously unknown complexity structure for 4-bit permutations.

1.5 Classification Strategy

1.5.1 What Functions are Equivalent?

Linear functions play an integral role in virtually every field of mathematics. Why? Because we can solve, manipulate, invert, combine and expand them in an understandable way that does not carry over very neatly into the realm of nonlinear functions. We do not use linear functions because they are more common than nonlinear functions. We use linear functions because they represent the majority of problems that we know how to solve.

A linear function of a bit vector is a matrix where all of the entries are 0 and 1 and the resulting product is reduced modulo 2. This is the same as the ring of matrices over the two-element finite field GF(2). Of special interest are the linear functions that are invertible. These linear functions form the group $\mathrm{GL}_n(2)$ and will be referred to as *linear permutations* since they permute the 2^n n-bit vectors.

Considering n-bit permutations in this light, linear permutations might be considered the permutations with minimal complexity. They form a subgroup in which it is easy to multiply and invert. Their matrix form is much more compact (n^2 bits) than their associated permutation description ($n2^n$ bits). In fact, we will want to slightly enlarge our definition of minimal complexity permutations to include all n-bit *affine permutations*, since affine permutations also share the same nice properties.

After deciding that affine permutations have minimal complexity, it is natural to associate two nonlinear functions if they are equivalent under affine transformations. This equivalence divides S_{2^n} into disjoint double coset where $\mathrm{AGL}_n(2)$ acts on the inputs and outputs of a permutation.

It is interesting to note that the current best known bound between the complexity of a function and its inverse is based on a sparse linear transformation with a dense inverse [Hil93].

1.5.2 How Do We Induce a Complexity Hierarchy?

We obviously need a nonlinear permutation to add to our affine permutations, or we will never be able to realize all permutations in S_{2^n}. There are two reasonable choices here, and both will be considered.

First, the Toffoli and Fredkin gates are two different universal gates for reversible computation. We will prove that these gates are affine equivalent, and thus they both induce the same complexity structure. This will be referred to as the *Toffoli basis* or B_T.

Second, controlled affine permutations are also universal for reversible computation. Although this class of functions is not as simple as a single Toffoli gate, controlled affine functions have a complexity roughly equivalent to the linear transformations between each nonlinear round. They are also easy to invert and cover a broader range of permutations faster. These gates will be referred to as the *controlled affine basis* or B_{CA}.

Once we have chosen which nonlinear basis we wish to use, we can induce the related complexity tree. We start with affine permutations as our level zero complexity (zero rounds of nonlinearity). Then all equivalence classes that can be reached using one gate from our nonlinear basis become level one complexity (one round of nonlinearity). Then the new equivalence classes that can be reached from level one, become level two complexity and so on. We can then create a graph illustrating how different permutations relate and the number of rounds of nonlinearity needed to achieve a particular function gives a rough complexity measure.

1.5.3 What Can We Compute?

Even though we have reduced the combinatorial complexity problem to a much more algebraic problem, we still face formidable computational problems. The equivalence classes themselves are double cosets, and there are currently no truly satisfactory algorithms for double coset enumeration, equivalence or canonical representation [Hol05].

Much of the paper develops the theory behind special methods for solving affine double coset problems specifically over S_{2^n}. The nonlinear complexity depth is computed for all 3 and 4 bit permutations.

Chapter 2

Complexity Measures

2.1 Summary

This chapter recalls three different, but known, complexity measures. These measures serve as a useful benchmark for assessing the advantages of the nonlinear complexity measure. The complexity measures introduced are:

- Transposition Complexity Measure
- Gate Complexity Measure
- Reversible Gate Complexity Measure

2.2 A Different View of Complexity

In computer science, studies of complexity usually focus on an infinite class of functions. For example, the complexity class P, contains decision problems that can be computed in polynomial time, and NP, contains decision problems whose answer can be verified in polynomial time. The open question of whether or not $\mathsf{P} = \mathsf{NP}$ is the most important question in computer science.

However, we may be interested in a particular function and not an infinite class. In designing a 64-bit adder, there will be optimal constructions given a set of building blocks (e.g. logic gates, transistors, etc.). Knowing which complexity class addition is in may provide a reasonable starting design, but finding an optimal design is a much harder problem.

Let us now consider a number of measures for assessing the optimal construction of a function.

2.3 Transposition Complexity Measure

One simple complexity measure in which a complete theory can be easily developed is the transposition complexity measure. Recall that a **permutation** p on $N = \{1, 2, \dots, n\}$ is a bijective map from N to itself. One common representation for permutations is:

$$\begin{pmatrix} 1 & 2 & \dots & n \\ p(1) & p(2) & \dots & p(n) \end{pmatrix}$$

NOTE: Due to the computational nature of the permutations studied in this paper, it will be more natural to consider "zero-based" permutations instead of the more traditional "one-based" permutations used in most algebra texts. Thus, permutations on n elements will be presented as:

$$\begin{pmatrix} 0 & 1 & \dots & n-1 \\ p(0) & p(1) & \dots & p(n-1) \end{pmatrix}$$

Example **2.3.1.** Consider the function $p(x) = 3x + 3 \mod 8$. Evaluating the function for each x in $\{0, 1, 2, 3, 4, 5, 6, 7\}$ yields:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 6 & 1 & 4 & 7 & 2 & 5 & 0 \end{pmatrix}$$

Lemma **2.3.2.** *Every permutation can be written as a composition of disjoint cycles.*

Proof. See [Jac74, 49] □

Example **2.3.3.** Decomposing the permutation $3x+3 \mod 8$ into cycles yields:

$$(0347)(1652)$$

A **transposition** is a permutation that swaps two elements and leaves all of the other elements fixed. Essentially, it is the simplest non-trivial permutation. Using the key fact that any cycle can be decomposed into a product of transpositions (e.g. $(0347) = (47)(37)(07)$), we arrive at another simple, yet powerful result.

Lemma **2.3.4.** *Every permutation can be written as a composition of transpositions. Furthermore, every m-cycle can be written as the composition of $m-1$ transpositions.*

Proof. See [Jac74, 49-50] □

We can now define the **transposition complexity**, C_T, of a permutation to be the minimum number of transpositions whose composition realizes that permutation.

Lemma **2.3.5.** *If p is a permutation on $\{0, 1, \dots, n-1\}$ then $C_T(p) = n - c$, where c is the number of disjoint cycles in p.*

Proof. See [Mas96] □

Let us now consider the pros and cons of the transposition complexity measure.

- **Pro:** C_T is easy to compute given a permutation.
- **Pro:** C_T defines a partial ordering on the complexity measure of permutations.
- **Con:** Linear functions have high complexity. Linear and affine functions, which are generally considered easy to compute, can obtain close to maximal complexity.
- **Con:** A permutation and its inverse always have the same complexity. This leads to the surprising result that there are no one-way permutations for the transposition complexity measure. If we believe that there are one-way permutations, then this complexity measure is not measuring the right thing.

2.4 Gate Complexity Measure

This is probably *the* complexity measure that first comes to mind for most researchers. First, choose a universal set of gates such as AND, OR and NOT, or all possible 2-bit gates, or even just NAND gates. Then count the number of gates necessary to compute a function. Functions are created by connecting the gates in an acyclic network and then computing the output given a specific input. The **gate complexity** of a given function with respect to a given set of universal gates is the minimum number of gates needed to realize the function.

Let us now consider how gate complexity stacks up against our criteria.

- **Pro:** Simple functions have simple constructions.
- **Pro:** Counting gates defines a partial ordering on the complexity measure of functions.
- **Con:** Computing the measure seems to be very difficult. It appears that exhaustively enumerating all possible functions from acyclic networks is the only way to find the minimum function in general.
- **Con:** Functions that are closely related can have very different looking gate structures. It is also difficult to know if functions are only one gate away from each other without exhaustively checking.

Current state of the art: Exhaustively searching the $2^{2^6} = 2^{64}$ functions that map 6 bits to 1 bit or the $(2^4)! \approx 2^{44}$ permutations over 4 bits is prohibitive. In a search of the literature, the author has found no indication that these cases have been exhausted. However, in 1997, the coordinated efforts of DESCHALL exhaustively searched the 2^{56} DES keys in an RSA Security challenge. Thus, either of these classifications could possibly be done by a large coordinated effort. To date, it seems that only the minimum number of gates needed for all 3-bit permutations and 5-bit to 1-bit functions have been computed.

2.5 Reversible Gate Complexity Measure

Since it appears that look-up tables can not be used very efficiently on a quantum computer, there has been increased interest in finding the optimal gate construction of a function under the restrictions of reversibility.

Unfortunately, the search for optimal gate implementations in the reversible context faces many of the same challenges that plague the gate complexity measure [MDM07].

- **Pro:** Counting reversible gates defines a partial ordering on the complexity measure of permutations.
- **Pro:** Has low complexity for simple functions. Since classical computation can be efficiently simulated by a reversible computation, simple functions will have simple reversible gate implementations.
- **Con:** Exhaustively enumerating all possible functions by applying new gates again appears to be the only way to find the minimum function in general.

2.6 Nonlinear Reversible Complexity Measure

Each of the previous complexity measures have been studied in depth, and will not be treated further in this paper. However, each of these complexity measures provides a useful measuring stick in evaluating new complexity measures. We now present the basics of nonlinear complexity, and analyze its advantages.

Since linear functions are so simple, yet powerful, it is reasonable to suggest that we consider two functions to be equivalent if a linear transformation on the input and output of one makes it equal to the other. Using this basic notion of equivalence, it is now natural to measure a function by the number of rounds of nonlinearity necessary to achieve it.

Obviously, we must restrict the class of nonlinear connections, otherwise every function is achievable in at most one round of nonlinearity – using itself!

The class of nonlinear connections should have low complexity and have inverses that are easy to compute. Inspired again by quantum computing, the most natural nonlinear function is the Toffoli gate. This gate is universal for reversible computation.

- **Pro:** Since the complexity measure graph can be constructed from the identity up, we get a partial ordering similar to the gate complexity measure. Except in this case, we are counting the number of nonlinear functions needed.

- **Pro:** Simple functions have low complexity. Linear functions have complexity zero, since they require no rounds of nonlinearity. Simple nonlinear functions have implementations requiring few NAND gates. These can be converted into reversible functions using few Toffoli gates. Since each Toffoli gate can be used as a round of nonlinearity, this sets an upper bound on the number of rounds of nonlinearity a function needs.

- **Pro:** We are able to compute a complexity graph for 4-bit permutations. This is currently not feasible for the gate complexity measure or reversible gate complexity measure.

- **Con:** Similar to combinatorical complexity measures, this approach also quickly runs out of steam due to the doubly exponential size of the space we are exploring.

Possibly the most exciting result of the nonlinear complexity measure is that we are able to use more powerful mathematics to analyze the problem of complexity.

Chapter 3

Reversible Circuits

3.1 Summary

This chapter introduces the basic gates of reversible computing and illustrates how these gates may be composed. Three different methods for visualizing reversible circuits are introduced: circuit diagram, truth table and wire program.

The majority of the content can be found in the open literature. It is presented here to provide the reader sufficient background to understand the material in the later chapters.

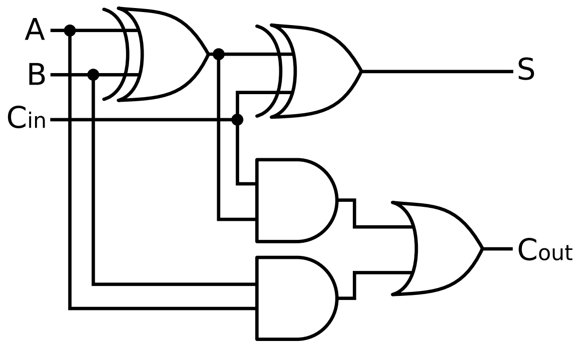
3.2 Classical Circuit Diagrams

Classical circuits may be represented as diagrams connecting the inputs and outputs by a combination of wires connected by universal gates. There are certain rules to building circuits, such as no closed loops, but these restrictions will not be discussed in this paper.

Example **3.2.1.** One very common building block of circuit design is the full adder, which computes the sum and carry of two bits and an incoming carry bit.

A common presentation is given in Figure 3.1.

Figure 3.1: Full Adder from XORs, ANDs and ORs



It is also useful to represent reversible circuits with diagrams. In [Ben73], Bennett discusses the necessary conditions for reversible computing. For reversible circuits, the conditions imply that

1. The number of wires is constant throughout the computation.
2. Every gate or action must be a permutation.
3. Wires may not be split or joined.

3.3 Basic Reversible Gates

The three most commonly used reversible gates are the NOT gate, the controlled NOT gate or CNOT, and the Toffoli gate, also referred to as a controlled-controlled-NOT gate, a doubly controlled NOT gate or simply CCNOT. As we discuss reversible functions it is useful to view the functions in multiple forms, the wire program, circuit diagram and truth table. We now present some basic gates in each of the three forms.

In the wire programs presented in this thesis, the binary operation $\oplus=$ will be used repeatedly. This notation is very useful in reversible computing since it ensures that the end function is a permutation. Similar to a C program where $a+=2$ means that the value of a is incremented by 2, $\langle k\rangle\oplus= f(a_0, a_1, \ldots a_n)$ means that the value of $\langle k\rangle$ is updated by XORing it with the value of f.

3.3.1 NOT Gate

The NOT gate is the simplest and acts on only one bit. The wire program for the NOT gate acting on the kth bit is simply $\langle k\rangle\oplus= 1$, In figure 3.2, the wire $\langle 0\rangle$ is negated.

Figure 3.2: Circuit Diagram: NOT Gate

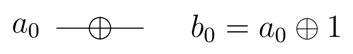


Table 3.1: Truth Table: NOT Gate

Input (a_0)	Output (b_0)
0	1
1	0

Given an n wire circuit, the NOT gate applied to various wires will generate any bit translation (e.g. $f(x) = x \oplus b$ where $b \in Z_2^n$).

Table 3.2: Wire Program: NOT Gate

Instruction	$\langle 0\rangle$
	a_0
$\langle 0\rangle \oplus= 1$	$a_0 \oplus 1$

3.3.2 Controlled NOT Gate (CNOT)

The controlled NOT gate or CNOT adds the value of one wire to another. If $\langle i\rangle$ is the control and $\langle j\rangle$ is the target, then the wire program for the CNOT gate is $\langle j\rangle \oplus= \langle i\rangle$. In figure 3.3, $\langle 0\rangle$ is the control and $\langle 1\rangle$ is the target.

Figure 3.3: Circuit Diagram: Controlled NOT Gate

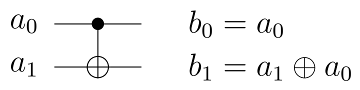


Table 3.3: Truth Table: Controlled NOT Gate

a_0a_1	b_0b_1
0 0	0 0
0 1	0 1
1 0	1 1
1 1	1 0

Note that the control and target can be any wires. The CNOT gates on n wires generate the linear group $\mathrm{GL}_n(2)$. Combining CNOT and NOT gates will generate the affine linear group $\mathrm{AGL}_n(2)$.

Table 3.4: Wire Program: CNOT Gate

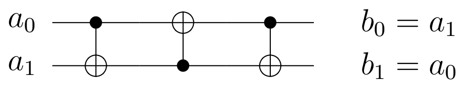
Instruction	$\langle 0\rangle$	$\langle 1\rangle$
	a_0	a_1
$\langle 1\rangle \oplus= \langle 0\rangle$	a_0	$a_1 \oplus a_0$

3.3.3 SWAP

Lemma **3.3.1.** *The value of two lines can be swapped using 3 CNOTs. (Referred to as SWAP.)*

Proof. If we wish to switch the values stored in $\langle i\rangle$ and $\langle j\rangle$, we can simply apply the circuit in Figure 3.4.

Figure 3.4: Circuit Diagram: SWAP



We can trace the values on line $\langle 0\rangle$ and line $\langle 1\rangle$ through the following wire program:

Table 3.5: Wire Program: SWAP

Instruction	$\langle 0\rangle$	$\langle 1\rangle$
	a_0	a_1
$\langle 1\rangle \oplus= \langle 0\rangle$	a_0	$a_1 \oplus a_0$
$\langle 0\rangle \oplus= \langle 1\rangle$	a_1	$a_1 \oplus a_0$
$\langle 1\rangle \oplus= \langle 0\rangle$	a_1	a_0

Table 3.6: Truth Table: SWAP

a_0a_1	b_0b_1
0 0	0 0
0 1	1 0
1 0	0 1
1 1	1 1

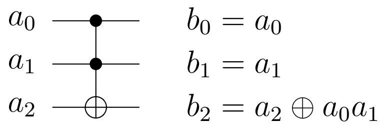
This is a well known trick from computer science when you wish to switch the value of two variables and not use a temporary variable.

□

3.3.4 Toffoli Gate

The doubly controlled NOT gate, CCNOT or Toffoli gate adds the product of two wires to another. If $\langle i\rangle$ and $\langle j\rangle$ are the controls and $\langle k\rangle$ is the target, then the wire program for the CCNOT gate is $\langle k\rangle \oplus = \langle i\rangle\langle j\rangle$. In figure 3.5, $\langle 0\rangle$ and $\langle 1\rangle$ are the controls and $\langle 2\rangle$ is the target.

Figure 3.5: Circuit Diagram: Toffoli Gate



The Toffoli is probably the most well known universal reversible gate. Note that both the NOT and CNOT gates can be derived from the Toffoli gate by forcing certain control lines to be 1.

Table 3.7: Wire Program: Toffoli Gate

Instruction	$\langle 0\rangle$	$\langle 1\rangle$	$\langle 2\rangle$
	a_0	a_1	a_2
$\langle 2\rangle \oplus = \langle 0\rangle \cdot \langle 1\rangle$	a_0	a_1	$a_2 \oplus a_0a_1$

Table 3.8: Truth Table: Toffoli Gate

$a_0a_1a_2$	$b_0b_1b_2$
0 0 0	0 0 0
0 0 1	0 0 1
0 1 0	0 1 0
0 1 1	0 1 1
1 0 0	1 0 0
1 0 1	1 0 1
1 1 0	1 1 1
1 1 1	1 1 0

3.3.5 Example Permutation

EXAMPLE **3.3.2.** Consider the function $f(a_0, a_1, a_2) = (a_2 \oplus a_0a_1, a_1 \oplus 1, a_2 \oplus a_0a_1 \oplus a_0)$. The circuit diagram, wire program and truth table for f are in figure **??**, and tables 3.9 and 3.10.

Figure 3.6: Circuit Diagram: $f = (a_2 \oplus a_0a_1, a_1 \oplus 1, a_2 \oplus a_0a_1 \oplus a_0)$

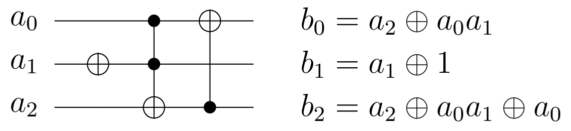

a_0
a_1
a_2
$b_0 = a_2 \oplus a_0a_1$
$b_1 = a_1 \oplus 1$
$b_2 = a_2 \oplus a_0a_1 \oplus a_0$

Table 3.9: Wire Program: $f = (a_2 \oplus a_0a_1, a_1 \oplus 1, a_2 \oplus a_0a_1 \oplus a_0)$

Instruction	$\langle 0\rangle$	$\langle 1\rangle$	$\langle 2\rangle$
	a_0	a_1	a_2
$\langle 1\rangle \oplus= 1$	a_0	$a_1 \oplus 1$	a_2
$\langle 2\rangle \oplus= \langle 0\rangle\langle 1\rangle$	a_0	$a_1 \oplus 1$	$a_2 \oplus a_0a_1 \oplus a_0$
$\langle 0\rangle \oplus= \langle 2\rangle$	$a_2 \oplus a_0a_1$	$a_1 \oplus 1$	$a_2 \oplus a_0a_1 \oplus a_0$

Table 3.10: Truth Table: $f = (a_2 \oplus a_0a_1, a_1 \oplus 1, a_2 \oplus a_0a_1 \oplus a_0)$

$a_0a_1a_2$	$b_0b_1b_2$
0 0 0	0 1 0
0 0 1	1 1 1
0 1 0	0 0 0
0 1 1	1 0 1
1 0 0	0 1 1
1 0 1	1 1 0
1 1 0	1 0 0
1 1 1	0 0 1

3.3.6 Fredkin Gate

Another common universal gate for reversible computing is the Fredkin gate, or controlled swap gate. While the Toffoli gate has two controls and one target, the Fredkin gate has one control and two targets. If the control is 1, then the values of the two targets are swapped. Again, if the control is 0, then no action takes place.

Writing wire programs with the Fredkin gate is awkward since it updates the value of two wires at once. For this reason, most wire programs will be written

Figure 3.7: Circuit Diagram: Fredkin Gate

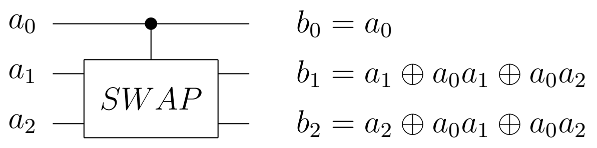

a_0
a_1
a_2
$SWAP$
$b_0 = a_0$
$b_1 = a_1 \oplus a_0a_1 \oplus a_0a_2$
$b_2 = a_2 \oplus a_0a_1 \oplus a_0a_2$

Table 3.11: Truth Table: Fredkin Gate

$a_0a_1a_2$	$b_0b_1b_2$
0 0 0	0 0 0
0 0 1	0 0 1
0 1 0	0 1 0
0 1 1	0 1 1
1 0 0	1 0 0
1 0 1	1 1 0
1 1 0	1 0 1
1 1 1	1 1 1

using Toffoli gates instead of Fredkin gates. Due to the linear equivalence between Toffoli and Fredkin gates (that will be proven later), the necessary conversion is fairly trivial.

3.4 Linear transformations on reversible circuits

Linear transformations are central to the equivalence relation that will be developed later. Not surprisingly, linear permutations can be constructed using only a small subset of the gates presented so far.

LEMMA **3.4.1.** *The CNOT gates on n wires generate $GL_n(2)$.*

Proof. Observe that the CNOT from $\langle i \rangle$ to $\langle j \rangle$ is the same as adding column i to column j and the SWAP gate on $\langle i \rangle$ and $\langle j \rangle$ simply swaps columns i and j. These elementary column operations can be used to reduce any invertible linear transformation to the identity matrix. Thus, by simply reversing the echelon reduction, any invertible linear transformation can be made from CNOTs. $\square$

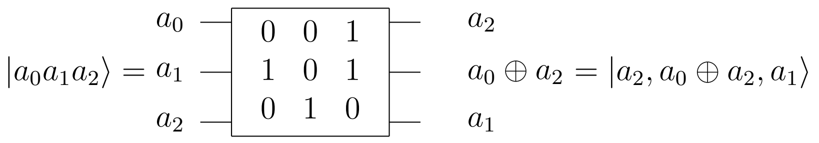
Instead of representing a linear transformation as a series of controlled NOT gates, we can represent the linear transformation as a single matrix action on a group of wires. This can be done without hiding too much complexity since every linear permutation may be constructed using $O(n)$ CNOTs [KMS07].

As the binary state progresses from left to right, the binary state will be a GF(2) column vector acted on by left multiplication of GF(2) matrices. Mirroring the physics convention for quantum vectors, we will represent the column vectors using the "ket" notation. (e.g. The column vector $[1, 1, 0]^T = |110\rangle$.)

Example **3.4.2.** Consider the vector $|110\rangle$ acted upon by the linear transformation $\left[\begin{smallmatrix} 0&0&1\\1&0&1\\0&1&0 \end{smallmatrix}\right]$. The result will be the vector $|110\rangle$ multiplied on the left by the matrix

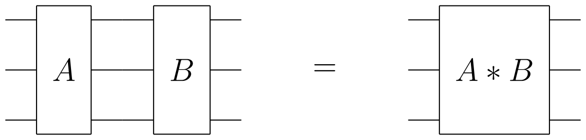
$$\left[\begin{smallmatrix} 0&0&1\\1&0&1\\0&1&0 \end{smallmatrix}\right] |110\rangle = \left[\begin{smallmatrix} 0&0&1\\1&0&1\\0&1&0 \end{smallmatrix}\right] [1,1,0]^T = \left[\begin{smallmatrix} 0&0&1\\1&0&1\\0&1&0 \end{smallmatrix}\right] \left[\begin{smallmatrix} 1\\1\\0 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 0\\1\\1 \end{smallmatrix}\right] = [0,1,1]^T = |011\rangle .$$

Figure 3.8: Circuit Diagram: Linear Matrix Example


a_0
$|a_0 a_1 a_2\rangle = a_1$
a_2
0 0 1
1 0 1
0 1 0
a_2
$a_0 \oplus a_2 = |a_2, a_0 \oplus a_2, a_1\rangle$
a_1

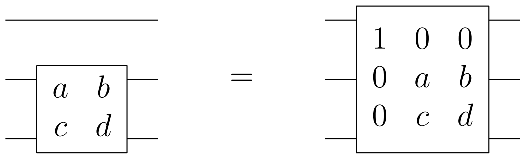
REMARK **3.4.3.** In order to maintain a consistent notation with established notation, actions are applied to the input ket from left to right. When considering linear actions applied to the ket, the matrix multiplication is applied on the left of the column vector. Thus, to retain readability, we will use a notation similar to the opposite ring or R^{op} notation when multiplying matrices in block form. Thus we define $A * B = BA$ for matrices A and B. Note that applying matrix A and then B to ket $|x\rangle$ yields $B(A|x\rangle) = (A * B)|x\rangle$.

Figure 3.9: Block matrix product acting on kets ($A * B = BA$)



It should also be noted that linear transformations can be enlarged to include bit lines they do not affect. Of course, the opposite is also true. Bit lines that are unaffected can be dropped out of the block matrix.

Figure 3.10: Two equivalent block matrix actions



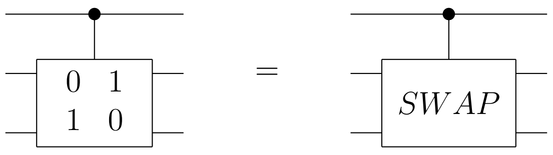
3.5 Controlled Linear Transformations

Controlled linear transformations will operate in a wire diagram as anticipated. If the controls are all 1 then the diagram will apply the linear transformation.

Otherwise, no action is taken.

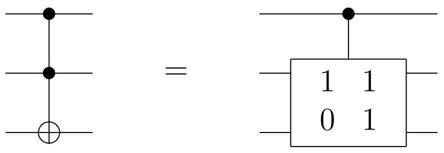
EXAMPLE **3.5.1.** We can express the Fredkin gate by controlling the matrix $\left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right]$.

Figure 3.11: Fredkin gate as a controlled linear transformation



EXAMPLE **3.5.2.** Since a single control affine shift is equivalent to a linear transformation, we can express the Toffoli gate as a controlled linear transformation.

Figure 3.12: Toffoli gate as a controlled linear transformation

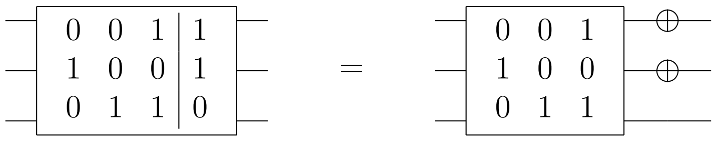


3.6 Affine Transformations

An affine transformation is simply a linear transformation plus the addition of a vector. Flipping the state of a wire in reversible circuits is typically represented by placing a $\oplus$ on the wire. While this notation is a little strange, it also makes sense. It is simply a controlled NOT without the control, and therefore just a NOT gate.

Example **3.6.1.** We can express the affine transformation $\begin{bmatrix} 0&0&1\\1&0&0\\0&1&1 \end{bmatrix} + |110\rangle$ in two different ways.

Figure 3.13: Equivalent Affine Transformations



Controlled affine transformations will also operate in exactly the same manner as controlled linear transformations. If the controls are all 1, then the affine transformation will be applied. Otherwise, no action is taken.

3.7 Scratch Space

Finally we consider scratch space, or additional computation space. For various calculations, additional wires may be needed to store intermediate values or the final calculation. In some cases, such additional wires are necessary, in other cases, they simply speed up the computation. It should be noted that although the Toffoli gate is universal, it is only universal given enough scratch space.

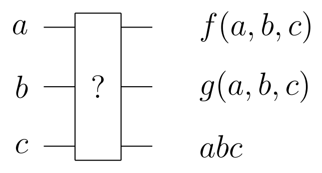
To illustrate how scratch space is used in reversible computation, we will consider a simple problem: computing the AND of three wires.

3.7.1 3-bit AND

Suppose we wanted to construct the 3-bit AND from Toffoli, CNOT and NOT gates. We will explore various ideas to show why two scratch bits are necessary and give motivation for the idea that works.

Since a permutation is a bijection, there do not exist functions f and g such that figure 3.14 can have the output abc on the third wire (Two values must map to one). Thus, we must utilize at least one additional wire to perform a reversible computation for the 3-bit AND gate.

Figure 3.14: 3-bit AND Using 3 Wires?



Now consider the circuit in figure 3.15.

Figure 3.15: 3-bit AND: Second Attempt

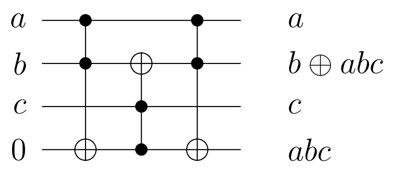


Figure 3.15 has computed the value of the 3-bit AND abc, but one of the wires has been modified. It is often preferable not to modify the input variables in the event they are needed for other computations. This is corrected in the implementation in figure 3.16.

Figure 3.16 now does seem to do the trick, but usually in reversible computing, one wants to compute a value and have it XORed onto some set of bits. In this case we are relying on the scratch bit to be initialized to zero for the 3-bit AND to work. Figure 3.17 shows the circuit evaluated with an arbitrary scratch bit d.

Figure 3.16: 3-bit AND: Third Attempt

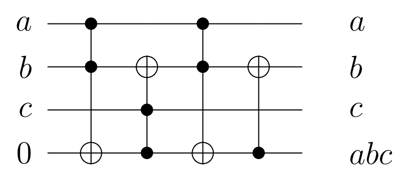
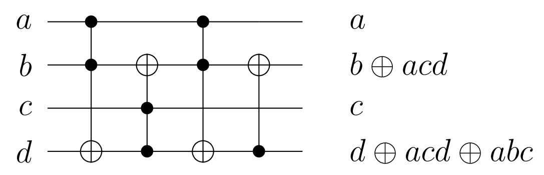


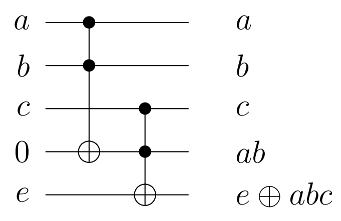
Figure 3.17: 3-bit AND: Third Attempt (Revisited)



Computing the 3-bit AND and XORing its value on the fourth bit is a triply controlled NOT. Upon further reflection, we realize that the CCCNOT, or triply controlled NOT is an odd permutation on 4-bit wires. Since the NOT, CNOT and CCNOT are all even permutations on 4-bit wires, we will need to add yet another scratch bit to create a permutation equivalent to CCCNOT.

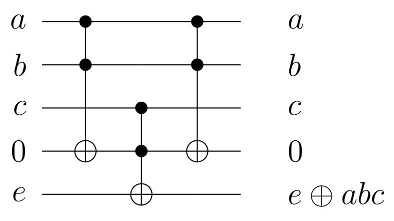
With 2 scratch bits, it doesn't take long to find the construction in figure 3.18.

Figure 3.18: 3-bit AND: Fourth Attempt



Thus, in figure 3.18 all of the input values are preserved and the 3-bit AND is XORed onto the target wire $\langle e \rangle$. The only problem is that one of our scratch bit still contains some left over information. For classical computation, this doesn't seem like much of an issue. But for reversible computation, erasure means heat dissipation. We would prefer to "uncompute" the scratch bit to return it to its original value. Furthermore, if the reversible computation is to be used on a quantum computer, it is necessary to return any scratch space used to its original state or it will destroy the superposition of the computation. Thus, figure 3.19 is still a better version of our 3-bit AND.

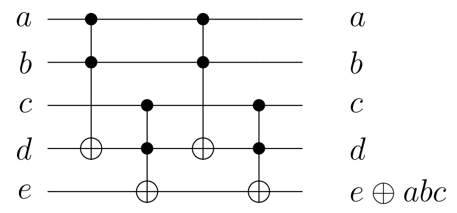
Figure 3.19: 3-bit AND: Fifth Attempt



Finally, there is one more surprising improvement that can be made. The fourth scratch bit does not need to be initialized to zero for the computation to work. For any value in d, figure 3.20 successfully computes the 3-bit AND of a, b, c, XORs the value onto e and returns d to it original value.

This amazing trick is even more useful in quantum computation. Qubit wires can be used as computation scratch space, even when we cannot observe their contents.

Figure 3.20: 3-bit AND: Final Attempt



3.8 Conclusion

Hopefully this small foray into reversible computation illustrates some of the difficulty in exploring permutation constructions. One important observation to make at this point is the relationship between scratch space and one-way permutations.

Any reversible computation that does not use any scratch space, or a computation that “uncomputes” the scratch space that it does use, is easily reversible. All of the gates can be run in the reverse order.

Thus for a permutation to be one-way, some circuit must exist which can compute one direction of the permutation efficiently. Furthermore, this circuit must leave “dirty” scratch space (not “uncomputed”), otherwise the inverse computation can simply run the circuit backwards.

Chapter 4

Which Permutations Should Be Equivalent?

4.1 Results and Application

The major results of this chapter are:

- Affine equivalence is a maximal equivalence relation in S_{2^n}.
- The two major universal reversible gates (Toffoli and Fredkin) are affine equivalent.
- Singly-controlled linear and affine permutations are affine equivalent.

4.2 Background

In the early 1950's, Aiken[AtSotCL51] and Moore[Moo52] exhaustively computed the 402 equivalence classes of the 65536 four-variable Boolean functions (4-bit to 1-bit) under permutation and complementation of the input variables. Shortly thereafter, mathematicians found combinatorical methods for counting that were far superior to the exhaustive methods used by Aiken and Moore. This work aroused a lot of interest in using Polya theory to count equivalent Boolean functions. References to much of this work can be found in [Lor64].

Permutations as well as functions may be classified by Polya theory. This chapter explores various notions of equivalence, and why affine equivalence is considered optimal.

4.3 The Equivalence Choice

Any group action on the input and output variables will define an equivalence. Aiken and Moore's original research stemmed from the question of how many 4-input, 1-output "cans" had to be designed to create all 4-bit nonlinear functions. Since the input plugs could be arranged in any order, equivalence under permutation of the inputs was one of the first equivalences studied. The following is a list of some of the more natural and common choices.

1. Complementation of variables.
2. Permutation of variables.
3. Complementation and permutation of variables.
4. Linear transformation of variables.
5. Affine transformation of variables.

Table 4.1 is extracted from [Lor64]. It is helpful to see how the different equivalence relations on inputs and outputs change the number of classes of permutations. For most of the equivalence relations, no closed form solution to count them is known. A lower bound is provided for the last four.

Table 4.1: Number of Permutation Equivalence Classes

Equivalence	n	1	2	3	4
None	$2^n!$	2	24	40320	20922789888000
Complementation	$\frac{2^n!+(2^n-1)^2 2^{n-1}!2^{2^{n-1}}}{2^{2n}}$	1	6	924	81738720000
Permutation	$> \frac{2^n!}{(n!)^2}$	2	7	1172	36325278240
Comp. & Perm.	$> \frac{2^n!}{2^{2n}(n!)^2}$	1	2	52	142090700
Linear	$> \frac{2^n!}{2^{2n^2}}$	2	2	10	52246
Affine	$> \frac{2^n!}{2^{2n^2+2n}}$	1	1	4	302

Note that the group of affine transformations contains all of the other listed group actions as subgroups. One obvious question is whether or not there is a larger subgroup than affine transformations suitable for larger equivalence classes.

THEOREM **4.3.1** (O'Nan-Scott)**.** *Let F_q be the finite field with q elements and F_q^n the n-dimensional vector space over F_q. Let $AGL(n,q)$, $S(F_q^n)$ and $A(F_q^n)$ be the affine general linear group, symmetric group and alternating group acting on F_q^n respectively. Then $AGL(n,q) \cap A(F_q^n)$ is a maximal subgroup of $A(F_q^n)$.*

Proof. See [LPS87]. □

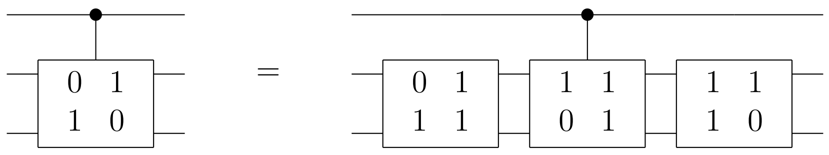
This proves that $\mathrm{AGL}_n(2)$ is a maximal subgroup in A_{2^n}. Thus, adding any nonlinear element of S_{2^n} to the affine group will generate all of A_{2^n} or S_{2^n}. Although S_{2^n} contains other maximal subgroups, affine equivalence is the natural choice if we wish linear functions to have low complexity.

4.4 Affine Equivalence of Toffoli and Fredkin Gates

As we start our journey of studying the complexity of functions using affine equivalence, it is refreshing to see that the two universal gates for reversible computing are in fact affine equivalent.

LEMMA **4.4.1.** *The Toffoli gate and the Fredkin gate are affine equivalent.*

Figure 4.1: Equivalence of Fredkin and Toffoli gates



Proof. We need to show that when the top wire is 0, no action as taken by both sides, and when the top wire is 1, the values in the bottom two lines are swapped.

When the top wire is 0, the controlled gate is not applied. Thus no action is taken on the left, and the action on the right is

$$\left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right] * \left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right] = \left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right] \cdot \left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right] = \left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right],$$

which is the identity, and therefore takes no action.

In the case where the top wire is 1, the action on the right is

$$\left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right] * \left[\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right] * \left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right] = \left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right] \cdot \left[\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right] \cdot \left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right] = \left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right],$$

which is equivalent to a swap. Thus the Toffoli and Fredkin gates are affine equivalent. □

4.5 Controlled Affine and Linear Permutations

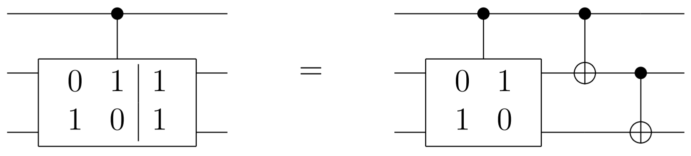
LEMMA **4.5.1.** *A singly controlled affine permutation is equivalent to a singly controlled linear permutation composed with an affine permutation.*

Proof. Separate the controlled affine function into the linear permutation and vector addition. The singly controlled vector addition is just the product of multiple CNOTs. Any composition of CNOTs is a linear permutation. Thus, the singly controlled affine permutation is equivalent to a controlled linear permutation and a linear permutation. □

Another way to consider the equivalence is: conditionally negating wire $\langle i \rangle$ by wire $\langle j \rangle$; which is the same as adding wire $\langle i \rangle$ to wire $\langle j \rangle$.

EXAMPLE **4.5.2.** Consider the following controlled affine permutation:

Figure 4.2: Affine Equivalent Controlled Transformations



Since the composition of CNOTs is simply a linear function, the controlled affine permutation on the left is linear (and therefore affine) equivalent to the controlled linear permutation on the right.

Chapter 5

Counting Equivalent Functions

5.1 Summary and Results

A method for counting affine equivalent double cosets in S_{2^n} is reviewed. This result is then extended to show how we can enumerate the affine equivalent double cosets in A_{2^n}.

5.2 Background

Polya theory is a very robust method for counting different types of objects, subject to an equivalence relation. Let us first review the basic counting theorems and how they are applied to counting the number of affine equivalent double cosets.

Definition **5.2.1.** Let f and g be n-bit permutations in S_{2^n}. Then f and g are **affine equivalent** or $f \equiv g$ if there exist a_0 and a_1, affine permutations in $\mathrm{AGL}_n(2) < S_{2^n}$, such that $f = a_0 g a_1$. This is equivalent to f and g belonging to the same double coset of S_{2^n} where the action on the right and left is by $\mathrm{AGL}_n(2)$

Anytime we have a group acting on a set, the group action will divide the set into disjoint orbits. The Cauchy-Frobenius Lemma, one of the most beautiful results of combinatorical group theory, states that the number of orbits is equal to the average number of fixed points by the group action.

We need to introduce some definitions before proceeding with the proof.

Definition **5.2.2.** Let G be a group acting on the set X. Define $X_g = \{x \in X | g \cdot x = x\}$. X_g is the set of **fixed points** under the action of G.

Definition **5.2.3.** Let G be a group acting on the set X. Define the **stabilizer** $G_x = \{g \in G | g \cdot x = x\}$. Then G_x is the subgroup of G that fixes the point x.

Definition **5.2.4.** Let G be a group acting on the set X. Define $G(x) = \{y \in X | \exists g \ni g \cdot x = y\}$. $G(x)$ is the **orbit** of the element x.

Lemma **5.2.5.** *(Orbit-Stabilizer relation) Let G be a group acting on the set X. Given $x \in X$, the length of the orbit $G(x)$ is equal to the index of the stabilizer:*

$$|G(x)| = |G/G_x|,$$

or equivalently,

$$|G(x)| \cdot |G_x| = |G|.$$

Proof. Define a mapping $\phi : G(x) \to G/G_x$ by $gx \mapsto gG_x$. Then using the following relations:

$$g_1 x = g_2 x \iff g_2^{-1} g_1 \in G_x \iff g_1 G_x = g_2 G_x,$$

we can see that ϕ is injective, onto, and well defined. Therefore, it is a bijection. It follows immediately that

$$|G(x)| = |G/G_x|.$$

□

We now have the machinery needed to prove the following.

THEOREM **5.2.6** (Cauchy-Frobenius). *Let G be a group acting on the set X. The number of disjoint orbits is equal to the average number of fixed points.*

$$|G//X| = \frac{1}{|G|}\sum_{g\in G}|X_g|,$$

where $|G//X|$ denotes the number of G-orbits in X.

Proof. Define the function $f : G \times X \to \{0,1\}$ by

$$f(g,x) = \begin{cases} 1 & \text{if } g\cdot x = x \\ 0 & \text{otherwise} \end{cases}$$

Then by summing over x or g we get:

$$\sum_{g\in G}|X_g| = \sum_{g\in G, x\in X} f(g,x) = \sum_{x\in X}|G_x|$$

Recall the Orbit-Stabilizer relation (we use it twice):

$$\text{For all } x\in X \qquad |G_x|\cdot|G(x)| = |G|.$$

Any group action divides $X = O_1 \cup O_2 \cup \cdots \cup O_N$ into a disjoint union of orbits, and it is exactly these orbits we wish to count. Consider the right hand sum over a single orbit generated by y.

$$\sum_{x\in O_y}|G_x| = \sum_{x\in O_y}|G_y| = |G_y|\cdot|G(y)| = |G|.$$

Thus, the sum of $|G_x|$ over any orbit is $|G|$. Therefore,

$$\sum_{g\in G}X_g = \sum_{x\in X}|G_x| = \sum_{x\in O_1\cup\cdots\cup O_N}|G_x| = \sum_{i=1}^{N}|G| = N|G|$$

where N is the number of orbits.

It follows immediately that

$$|G//X| = N = \frac{1}{|G|}\sum_{g\in G}X_g.$$

□

5.3 Counting Fixed Points Under a Double Coset Action

Define $\phi : (\mathrm{AGL}_n(2) \times \mathrm{AGL}_n(2)) \times S_{2^n} \to S_{2^n}$ by $\phi((a,b),p) = apb^{-1}$. In order to count the number of equivalence classes we have, we need to figure out how many permutations are fixed by the action of (a,b).

For any permutation we can define the cycle set α to be a tuple $(\alpha_1, \alpha_2, \alpha_3, \ldots)$ where each α_i indicates the number of i-cycles in the permutation. For any permutation on N elements,

$$\sum_i i\alpha_i = N.$$

Lemma **5.3.1.** *If the action of (a,b) on the permutation p fixes p, then a and b must have the same cycle set.*

Proof. Suppose the action (a,b) fixes p. Then $apb^{-1} = p$ or equivalently, $apb^{-1}p^{-1} = 1$. Let $C_k = (b_0 b_1 \ldots b_{k-1})$ be a k-cycle in b^{-1}. Then $pC_kp^{-1} = (p(b_0)p(b_1)\ldots p(b_{k-1}))$. This can be verified by checking that

$$pC_kp^{-1}(p(b_i)) = pC_k(b_i) = p(b_{i+1}).$$

Since conjugating by p maps k-cycles to k-cycles for all k, conjugating by p will take b to another permutation with the same cycle structure. Therefore a must also have the same cycle structure since a is the inverse of pbp^{-1}. □

Lemma **5.3.2.** *Given two permutations a and b with the same cycle set $(\alpha_1, \alpha_2, \ldots)$, the number of permutations fixed by the action (a,b) is*

$$\prod_k \alpha_k! k^{\alpha_k}.$$

Proof. Recall that when the action (a,b) fixes a permutation p, $apb^{-1}p^{-1} = 1$. Let $C_k^b = (b_0 b_1 \dots b_{k-1})$ be a k-cycle in b^{-1}. Then $pC_k^b p^{-1} = (p(b_0)p(b_1)\dots p(b_{k-1}))$. Given another k-cycle in a, C_k^a, there are k different permutation maps p that will conjugate C_k^b to become the inverse of C_k^a. This is simply due to the fact that the permutation can be shifted in k possible ways, all of which are actually the same permutation.

If we have α_k k-cycles in b^{-1} to match with α_k k-cycles in a, there are $\alpha_k!$ ways to choose the matching. With each matched pair we have shown that there are k permutations with which we can conjugate C_k^b to become C_k^a. Thus there are $\alpha_k! k^{\alpha_k}$ ways to match all k-cycles.

Since the permutation acts on each cycle independently, the total number of permutations fixed by the action (a,b) is the product over all k. □

For counting the number of double cosets in general, we would examine the cycle sets for the left and right action.

THEOREM **5.3.3.** *Let $H < S_{2^n}$ and let $H_\alpha = \{h \in H | h$ has cycle set $\alpha\}$. Then the number of HH double cosets of S_{2^n} is*

$$\frac{1}{|H|^2}\sum_{\alpha}|H_\alpha|^2\prod_k \alpha_k! k^{\alpha_k}.$$

Proof. Recall the double coset group action of $G = H \times H$ on S_{2^n}, $\phi : (H \times H) \times S_{2^n} \to S_{2^n}$ defined by $\phi((a,b),p) = apb^{-1}$. By Theorem 5.2.6, the number of orbits (double cosets) is

$$\begin{aligned} |G//S_{2^n}| &= \frac{1}{|G|}\sum_{g\in G} X_g \\ &= \frac{1}{|H|^2}\sum_{h_1,h_2\in H} X_{(h_1,h_2)} \end{aligned}$$

By Lemma 5.3.1 the number of fixed points is zero unless h_1 and h_2 are in the same cycle set. This reduces our equation to

$$\frac{1}{|H|^2}\sum_{\alpha}\sum_{h_1,h_2\in H_\alpha} X_{(h_1,h_2)}.$$

Applying Lemma 5.3.2, we get

$$\frac{1}{|H|^2}\sum_{\alpha}\sum_{h_1,h_2\in H_\alpha}\prod_{k}\alpha_k!k^{\alpha_k}.$$

Since the summand is independent of h_1 and h_2, we arrive at the desired result:

$$\frac{1}{|H|^2}\sum_{\alpha}|H_\alpha|^2\prod_{k}\alpha_k!k^{\alpha_k}.$$

□

5.4 Counting Affine Equivalent Double Cosets in S_{2^n}

By theorem 5.3.3, the number of affine equivalent double cosets in $\sum$ is

$$\frac{1}{|\mathrm{AGL}_n(2)|^2}\sum_{\alpha}|H_\alpha|^2\prod_{k}\alpha_k!k^{\alpha_k}$$

where H_α is the number of elements in H with the cycle type α. Recall that all the elements of any conjugacy class have the same cycle type. To compute $|H_\alpha|$, we will take the union of conjucacy classes in $\mathrm{AGL}_n(2)$ with cycle type α.

5.4.1 Example: Affine equivalent 3-bit permutations

We will now use MAGMA[BCP97] to find the cycle sets for the affine general linear group over GF(2).

```
> Classes(AGL(3,2));
Conjugacy Classes
-----------------
[1]     Order 1       Length 1
        Rep Id($)

[2]     Order 2       Length 7
        Rep (1, 2)(3, 4)(5, 6)(7, 8)

[3]     Order 2       Length 42
        Rep (1, 5)(2, 6)

[4]     Order 2       Length 42
        Rep (1, 2)(3, 4)(5, 8)(6, 7)

[5]     Order 3       Length 224
        Rep (1, 7, 4)(2, 5, 8)

[6]     Order 4       Length 84
        Rep (1, 4, 3, 2)(5, 8, 7, 6)

[7]     Order 4       Length 168
        Rep (1, 6, 2, 5)(7, 8)

[8]     Order 4       Length 168
        Rep (1, 8, 2, 5)(3, 6, 4, 7)

[9]     Order 6       Length 224
        Rep (1, 5, 7, 8, 4, 2)(3, 6)

[10]    Order 7       Length 192
        Rep (1, 5, 2, 3, 4, 6, 8)

[11]    Order 7       Length 192
        Rep (1, 6, 3, 5, 8, 4, 2)
```

Note that some classes have the same cycle set, even though they are not conjugate. These need to be grouped together in our counting. (i.e. conjugacy classes 2 and 4 are not conjugate, but both have a $(0, 4)$ cycle set.)

Table 5.1: Cycle Sets of AGL(3,2)

Cycle Set	Classes	Total	Count
(8)	1	1	$1^2 \cdot 8!1^8$
$(4,2)$	3	42	$42^2 \cdot 4!1^4 2!2^2$
$(0,4)$	$2,4$	49	$49^2 \cdot 4!2^4$
$(2,0,2)$	5	224	$224^2 \cdot 2!1^2 2!3^2$
$(0,0,0,2)$	$6,8$	252	$252^2 \cdot 2!4^2$
$(2,0,1,0,1)$	7	168	$168^2 \cdot 2!1^2 1!2^1 1!4^1$
$(0,1,0,0,0,1)$	9	224	$224^2 \cdot 1!2^1 1!6^1$
$(0,0,0,0,0,0,0,1)$	$10,11$	384	$384^2 \cdot 1!8^1$

Thus, we calculate the number of affine equivalent 3-bit permutations by summing the terms in table 5.1, and then dividing by $|\text{AGL}(3,2)|^2 = 1344^2$.

$$\begin{aligned}\text{SUM} &= 1^2 \cdot 8!1^8 + 42^2 \cdot 4!1^4 2!2^2 + 49^2 \cdot 4!2^4 + 224^2 \cdot 2!1^2 2!3^2 + 252^2 \cdot 2!4^2 \\ &\quad + 168^2 \cdot 2!1^2 1!2^1 1!4^1 + 224^2 \cdot 1!2^1 1!6^1 + 384^2 \cdot 1!8^1 \\ &= 7225344\end{aligned}$$

Thus the total number of double cosets is

$$\frac{\text{SUM}}{1344^2} = \frac{7225344}{1344^2} = 4.$$

5.4.2 Counting Affine Equivalent n-bit Permutations

Table 5.2 provides a general MAGMA program to compute the number of affine equivalent permutations for any n. The computation is mostly limited by the difficulty of finding the conjugacy classes of $\mathrm{AGL}_n(2)$.

Table 5.2: MAGMA function for counting double cosets

```
Count_AGL_double_cosets_in_Sym := function(n)
  H:=AGL(n,2); // Permutation group in Sym(2^n)
  ConjClass:=ConjugacyClasses(H);

  CycleTypes:={}; // Union together similar cycle sets
  for x in ConjClass do
    CycleTypes:=CycleTypes join {CycleStructure(x[3])};
  end for;

  sum:=0; // Compute and add summands for each cycle type
  for x in CycleTypes do
    size:=0;
    for y in ConjClass do
      if (x eq CycleStructure(y[3]))
      then size:=size+y[2]; end if;
    end for;
    summand:=size^2;
    for y in x do
      summand:=summand*Factorial(y[2])*(y[1]^y[2]);
    end for;
    sum:=sum+summand;
  end for;
  return sum/(#H)^2; // Divide by group order
end function;
```

Using the function in table 5.2, the author was able to compute the number of affine equivalent double cosets for $n \leq 7$. The total count grows doubly exponentially, so only the values for $n \leq 5$ are provided in table 5.4.

Table 5.3: Number of Affine Equivalent Permutations

n	Affine Equivalent	Total Permutations ($2^{n!}$)
2	1	24
3	4	40320
4	302	20922789888000
5	2569966041123963092	263130836933693530167218012160000000

5.5 Counting Affine Equivalent Double Cosets in A_{2^n}

Since the Toffoli gate is an even permutation for all $n > 3$, the Toffoli gate and $\mathrm{AGL}_n(2)$ can only generate A_{2^n}. For this reason, it will be more natural to examine the complexity structure of the double cosets in A_{2^n} and extend to S_{2^n} via an odd permutation only when necessary.

Let G be a group acting on a set X by conjugation. Let $C_G(x)$ denote the centralizer and x^G the conjugacy class of an element x. We will consider both S_{2^n} and A_{2^n} acting on A_{2^n} by conjugation. For any $\pi \in A_{2^n}$,

$$|C_{S_{2^n}}(\pi)| \cdot |\pi^{S_{2^n}}| = n!$$

$$|C_{A_{2^n}}(\pi)| \cdot |\pi^{A_{2^n}}| = \frac{n!}{2}.$$

Note that

$$C_{A_{2^n}}(\pi) = C_{S_{2^n}}(\pi) \cap A_{2^n} < C_{S_{2^n}}(\pi).$$

We will see that either the centralizer or conjugacy class will be half-sized in A_{2^n}. The following theorem details exactly what happens.

THEOREM **5.5.1.** *Let S_{2^n} and A_{2^n} both act on A_{2^n} by conjugation, and let $\pi \in A_{2^n}$. Then*

$$|C_{A_{2^n}}(\pi)| = \begin{cases} |C_{S_{2^n}}(\pi)| & \text{if } \pi \text{ has only odd distinct cycles.} \\ \frac{1}{2}|C_{S_{2^n}}(\pi)| & \text{otherwise.} \end{cases}$$

Proof. See [Sco87]. □

COROLLARY **5.5.2.** *Let S_{2^n} and A_{2^n} both act on A_{2^n} by conjugation, and let $\pi \in A_{2^n}$. Then*

$$|\pi^{A_{2^n}}| = \begin{cases} \frac{1}{2}|\pi^{S_{2^n}}| & \text{if } \pi \text{ has only odd distinct cycles.} \\ |\pi^{S_{2^n}}| & \text{otherwise.} \end{cases}$$

Proof. The order of the centralizer times the orbit must be $\frac{n!}{2}$. Thus if the centralizer doesn't change in A_{2^n}, the conjugacy class must split. Likewise, if the conjugacy class doesn't split in A_{2^n}, the centralizer must be half as large. □

Since most permutations do not have a cycle type with only distinct odd cycles, the conjugacy class for most permutations is the same between S_{2^n} and A_{2^n}. However, the centralizer is now half as large for these permutations.

Let β represent the distinct odd cycles and α the rest. Let β_1 and β_2 represent the two A_{2^n} conjugacy classes of cycle type β. Then the equation for theorem 5.3.3 reduces to

$$\frac{1}{|\mathrm{AGL}_n(2)|^2}\left(\sum_{\alpha}|H_\alpha|^2 \cdot \frac{1}{2}\prod_k \alpha_k! k^{\alpha_k} + \sum_{\beta,i=1,2}|H_{\beta_i}|^2 \prod_k \beta_k! k^{\beta_k}\right).$$

The number of affine equivalent double cosets in A_{2^n} is close to half the number in S_{2^n} due to the fact that most of the cycles types are not distinct odd cycles.

LEMMA **5.5.3.** *If the distinct odd cycles β of $AGL_n(2)$ split evenly into the two A_{2^n} conjugacy classes β_1 and β_2, then the number of affine equivalent double cosets in A_{2^n} is exactly half the number in S_{2^n}. I.e. if for all distinct odd cycles β*

$$|H_{\beta_1}^{A_{2^n}}| = |H_{\beta_2}^{A_{2^n}}| = \frac{1}{2}|H_{\beta}^{S_{2^n}}|$$

Then A_{2^n} contains half of the double cosets of S_{2^n}.

Proof. Since the summand result is already correct for cycle types α, that are not odd and distinct, we need only consider the summands associated with the odd distinct cycle types β.

$$\begin{aligned}
\sum_{\beta,i=1,2} |H_{\beta_i}|^2 \prod_k \beta_k! k^{\beta_k} &= \sum_{\beta,i=1,2} |\frac{H_\beta}{2}|^2 \prod_k \beta_k! k^{\beta_k} \\
&= \frac{1}{4} \sum_{\beta,i=1,2} |H_\beta|^2 \prod_k \beta_k! k^{\beta_k} \\
&= \frac{1}{4}\left(2 \cdot \sum_{\beta} |H_\beta|^2 \prod_k \beta_k! k^{\beta_k}\right) \\
&= \frac{1}{2} \sum_{\beta} |H_\beta|^2 \prod_k \beta_k! k^{\beta_k}.
\end{aligned}$$

Thus the contribution from both the odd distinct cycles β and the other cycles α is half in both cases. Therefore the number of affine double cosets A_{2^n} will be exactly half of S_{2^n}. □

Elements of the same AGL$_n(2)$ conjugacy class, will obviously be in the same A_{2^n} conjugacy class, thus if a A_{2^n} conjugacy class has a cycle type that is odd and distinct, there is no guarantee that there will be a matching conjugacy class. In order to test whether or not two permutations are in the same A_{2^n} conjugacy class, we will use the following result.

Lemma **5.5.4.** *Let p and q be permutations in S_{2^n} with the same odd distinct cycle structure. Since p and q have the same cycle structure, there exists $\sigma \in S_{2^n}$ such that $p = \sigma q \sigma^{-1}$. If σ is odd, then p and q are in different A_{2^n} conjugacy classes. If σ is even then p and q are in the same A_{2^n} conjugacy class.*

Proof. Find σ such that $p = \sigma q \sigma^{-1}$. Such a σ is guaranteed to exist since p and q have the same cycle structure.

Assume σ is an odd permutation. If p and q are in the same A_{2^n} conjugacy class, then there exists an even permutation τ such that $q = \tau p \tau^{-1}$. Thus

$$\begin{aligned} p &= \sigma q \sigma^{-1} \\ &= \sigma \tau p \tau^{-1} \sigma^{-1} \\ &= (\sigma\tau) p (\sigma\tau)^{-1}. \end{aligned}$$

Therefore $\sigma\tau$ is an odd permutation in the centralizer of p, which is a contradiction, since the centralizer of p is contained in A_{2^n}.

Assume that σ is an even permutation. Since $\sigma \in A_{2^n}$, this implies that p and q are in the same A_{2^n} conjugacy class. □

Consider now the number of cycles in a permutation in S_{2^n} with odd distinct cycles. Since the length of the cycles must sum to 2^n, there must always be an even number of distinct odd cycles. This leads us to a case that will pair two odd distinct cycle $\mathrm{AGL}_n(2)$ conjugacy classes.

Lemma **5.5.5.** *Let b be an affine permutation in $AGL_n(2)$, $n > 2$, with an odd distinct cycle structure β. Let m be the number of cycles in β. If $m \equiv 2 \pmod 4$, then b and b^{-1} are in different A_{2^n} conjugacy classes.*

Proof. Consider the cycles in the permutation b. If every cycle is reversed, we have permutation b^{-1}. We will consider the parity of the permutation σ that reverses every cycle by conjugation.

Let $C_k = (b_1 b_2 \cdots b_k)$ be a cycle of b. Define σ on C_k so that $\sigma(b_i) = b_{k-i+1}$. Then

$$\begin{aligned}
\sigma C_k \sigma^{-1} &= \sigma(b_1 b_2 \cdots b_k)\sigma^{-1} \\
&= (\sigma(b_1)\sigma(b_2)\cdots\sigma(b_k)) \\
&= (b_k \cdots b_2 b_1).
\end{aligned}$$

Since k is odd, the midpoint of the cycle stays fixed, and σ is thus composed of $\frac{k-1}{2}$ transpositions.

Extending σ over all cycles, we see that σ is composed of $\frac{2^n-m}{2}$ transpositions. Since $m \equiv 2 \pmod 4$, $\frac{2^n-m}{2}$ is odd. Therefore σ is an odd permutation. By lemma 5.5.4, b and b^{-1} must be in different A_{2^n} conjugacy classes. $\square$

Since the inversion action stays within $\mathrm{AGL}_n(2)$, there must be an exact pairing between the two A_{2^n} conjugacy classes within $\mathrm{AGL}_n(2)$. Thus all distinct odd cycle types with 2 (mod 4) cycles will split evenly in the A_{2^n} conjugacy classes. Thus the only cycle types which can affect the sum are the distinct odd cycle types with 0 (mod 4) cycles.

The case of $n = 5$ is the first instance where there exists a odd distinct cycle type with 0 (mod 4) cycles. There are two conjugacy classes of size 15237120 with a $(1, 3, 7, 21)$ cycle structure. A parity check between the two conjugacy classes reveals that they differ by an even conjugation. Lemma 5.5.4 indicates that the two conjugacy classes will be in the same A_{2^n} conjugacy class.

Let N_S be the number of affine equivalent 5-bit permutations. Computing the number of affine equivalent 5-bit even permutations N_A yields

$$\begin{aligned} N_A &= \frac{N_s}{2} + \frac{15237120^2 \cdot 3 \cdot 7 \cdot 21}{2|\mathrm{AGL}_n(2)|^2} \\ &= 1284983020561981548. \end{aligned}$$

Is should be noted that we only added half of the summand since the other half was already included in $\frac{N_S}{2}$. The correction factor turns out to only make a difference of 2, since $\frac{N_s}{2} = 1284983020561981546$.

In the cases where $n = 4, 6$, there are no odd distinct cycle types with $0 \pmod 4$ cycles. Thus, the number of affine equivalent even permutations is exactly half for those two cases. For $n = 7$ there are 10 conjugacy classes with odd distinct cycle type and $0 \pmod 4$ cycles.

Table 5.4: Number of Affine Equivalent Even Permutations

n	Affine Equivalent in A_{2^n}
3	2
4	151
5	1284983020561981548
6	38115488450430370396302626146823126191571813695482842576562378932

The fact that there are 151 affine equivalent classes in A_{16} will be used in the classification of 4-bit permutations.

5.6 Open problems

- *Develop a general method for computing the conjugacy classes of $AGL_n(2)$.*

A method was developed in [Hou06], but the results are incomplete. Hou's calculation indicated that the number of affine equivalent 5-bit permutations was 2569966041123938084 instead of 2569966041123963092. A verification check by the author revealed that Hou was missing two conjugacy classes found by MAGMA. While the canonical form (conjugacy class) of linear matrices has been well studied, the problem is surprisingly not completely solved for finding the canonical form of affine transformations.

- *Develop a general method for counting the affine equivalent double cosets in A_{2^n}.* Are there always half as many affine equivalent double cosets in A_{2^n} when n is even ($n > 2$)?

Chapter 6

Hamiltonian Cycles over $\mathbf{GL}_n(2)$

6.1 Results and Application

Identifying whether or not two functions are affine equivalent is critical to the development of a nonlinear hierarchy. This basic computation, which is essentially a double coset membership test, must be done many times, and thus requires a very efficient implementation.

With permutations stored as bit vectors, CNOTs can be applied by simple masking, shifting and XORing. These operations are much faster to implement than generically composing a permutation and a linear function.

Since the double coset test requires exhausting over all bit matrices in $\mathrm{GL}_n(2)$, it is natural to ask whether a Hamiltonian cycle exists over $\mathrm{GL}_n(2)$ where the vertices are the invertible matrices and the edges correspond to adding one row to another.

The major results of this chapter are:

- A proof that the subgroups of upper triangular matrices in $\mathrm{GL}_n(2)$ have Hamiltonian cycles.

- A heuristic algorithm for generating Hamiltonian cycles over $\mathrm{GL}_n(2)$
- The algorithm found Hamiltonian cycles over $\mathrm{GL}_n(2)$ for $n = 2, 3, 4, 5$.

The Hamiltonian cycles found are used to construct a highly efficient double coset test.

6.2 Background

A graph contains a **Hamiltonian cycle** if there exists a sequence of connected edges that pass through each vertex only once, returning to the the start vertex at the end. The Knight's Tour problem of finding a sequence of knight moves that touch each square of the chess board once is a famous Hamiltonian cycle problem. The Gray code is another famous example of a Hamiltonian cycle. In general, determining whether or not a given graph has a Hamiltonian cycle is NP-complete.

Given a group G and a generating set S, the **Cayley graph** $\Gamma = \Gamma(G, S)$ is constructed by assigning each element of G to a vertex, and then connecting two vertices if their difference is in S. Since S is a generating set, $\Gamma(G, S)$ must be connected.

The following lemma and corollary will be used to construct Hamiltonian cycles in this chapter.

Lemma **6.2.1.** *Let $\Gamma(G, S)$ be the Cayley graph for a group G and generating set $S = \{\sigma_0, \sigma_1, \ldots\}$. Let $f : \mathbb{Z}_n \to \{0, 1, 2, \ldots |S| - 1\}$ be a map such that $H = [\sigma_{f(0)}\sigma_{f(1)} \cdots \sigma_{f(n-1)}]$ is a Hamiltonian cycle on Γ. Then any cyclic rotation of H is a Hamiltonian cycle on Γ, i.e. $[\sigma_{f(k)}\sigma_{f(k+1)} \cdots \sigma_{f(k+n-1)}]$ is a Hamiltonian cycle on Γ for all $k \in \mathbb{Z}$.*

Proof. Since $H = [\sigma_{f(0)}\sigma_{f(1)}\cdots\sigma_{f(n-1)}]$ is a Hamiltonian cycle, we may list the elements of G in the following order.

$$\begin{gathered}\sigma_{f(0)}\\ \sigma_{f(0)}\sigma_{f(1)}\\ \sigma_{f(0)}\sigma_{f(1)}\sigma_{f(2)}\\ \vdots\\ \sigma_{f(0)}\sigma_{f(1)}\sigma_{f(2)}\cdots\sigma_{f(n-1)} = e\end{gathered}$$

If we now act on the elements of G by multiplying on the left by $\sigma_{f(0)}^{-1}$, we get another ordering of the elements of the group G.

$$\begin{gathered}e\\ \sigma_{f(1)}\\ \sigma_{f(1)}\sigma_{f(2)}\\ \vdots\\ \sigma_{f(1)}\sigma_{f(2)}\cdots\sigma_{f(n-1)}\end{gathered}$$

Recall that H is a Hamiltonian cycle implies

$$e = \sigma_{f(0)}\sigma_{f(1)}\sigma_{f(2)}\cdots\sigma_{f(n-1)}.$$

Conjugating both sides by $\sigma_{f(0)}$ yields

$$e = \sigma_{f(1)}\sigma_{f(2)}\cdots\sigma_{f(n-1)}\sigma_{f(0)}.$$

Moving the first term e to the end, now gives

$$\begin{array}{l}\sigma_{f(1)}\\ \sigma_{f(1)}\sigma_{f(2)}\\ \vdots\\ \sigma_{f(1)}\sigma_{f(2)}\cdots\sigma_{f(n-1)}\\ \sigma_{f(1)}\sigma_{f(2)}\cdots\sigma_{f(n-1)}\sigma_{f(0)}.\end{array}$$

This ordering is associated with the Hamiltonian cycle $[\sigma_{f(1)}\sigma_{f(2)}\cdots\sigma_{f(n-1)}\sigma_{f(0)}]$.

Since a single circular shift will yield a new Hamiltonian cycle, any circular shift will yield a new Hamiltonian cycle. Thus $[\sigma_{f(k)}\sigma_{f(k+1)}\cdots\sigma_{f(k+n-1)}]$ is a Hamiltonian cycle for any k. □

COROLLARY **6.2.2.** *Let $\Gamma(G, S)$ be the Cayley graph for a group G and generating set S. Let G_1 be a subgroup of G. If H is a Hamiltonian cycle over the vertices in G_1, then any rotation of H is a Hamiltonian cycle over G_1.*

Proof. Apply Lemma 6.2.1 the Cayley graph $\Gamma(G_1, S_1)$ where S_1 is the subset of S used in H. □

COROLLARY **6.2.3.** *Let $\Gamma(G, S)$ be the Cayley graph for a group G and generating set S. Let C_1 be a left coset of G. If H is a Hamiltonian cycle over the vertices in C_1, then any rotation of H is a Hamiltonian cycle over C_1.*

Proof. Any Hamiltonian cycle over a coset will also be a Hamiltonian cycle over the associated subgroup. □

The **Lovász Conjecture** simply states that every finite connected Cayley graph contains a Hamiltonian cycle. Many sub-cases of the the Lovász Conjecture have been proved. These cases mostly deal with special generating sets for the symmetric group S_n. For a survey of these results see [CG96]. The author is not aware of any results over $\mathrm{GL}_n(2)$.

6.3 Existence

The computational effect of an AND, shift and XOR is the same as applying a single CNOT gate, which is the same as the linear operation of adding one row to another. This will be demonstrated in detail later. It is well known from linear algebra that simple row operations will generate any matrix. Thus, the computational speed-up relies on the solution of the following problem.

Problem **6.3.1.** Starting from the identity matrix, is it possible to step through all possible invertible matrices in $\mathrm{GL}_n(2)$ using only a single row operation at each step?

In other words, is there a Hamiltonian cycle over $\mathrm{GL}_n(2)$ with the generating set $S = \{\sigma_{ij}\}$, where σ_{ij} is the operation of adding row i to row j?

For $n = 2$, the problem turns out to be quite easy. Since $\sigma_{ij}^2 = 1$ for all i, j, the only choice is to alternate between σ_{12} and σ_{21}. It turns out that $[\sigma_{12}, \sigma_{21}, \sigma_{12}, \sigma_{21}, \sigma_{12}, \sigma_{21}]$, is a Hamiltonian cycle as follows:

$$\left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right] \xrightarrow{\sigma_{12}} \left[\begin{smallmatrix}1&0\\1&1\end{smallmatrix}\right] \xrightarrow{\sigma_{21}} \left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right] \xrightarrow{\sigma_{12}} \left[\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right] \xrightarrow{\sigma_{21}} \left[\begin{smallmatrix}1&1\\1&0\end{smallmatrix}\right] \xrightarrow{\sigma_{12}} \left[\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right] \xrightarrow{\sigma_{21}} \left[\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right]$$

Finding a solution for $n = 3$ turned out to be not so easy. Early attempts to solve this problem found many near misses. Even though the ability to add any row to any other row gives a lot of freedom, the extra freedom made searching very difficult. Appealing to the Lovász conjecture, a solution should exist for any generating set. Consider the following lemma.

LEMMA **6.3.2.** *Let $G = GL_n(2)$ and let σ_i be the row operation that adds row i to row $i+1$. For σ_n, the row operation will wrap around and add row n to row 1. Then $S = \{\sigma_i \text{ for } i \in [1..n]\}$ is a generating set for G.*

Proof. Each $\sigma_i = \sigma_{i,i+1}$. Noting that $\sigma_{ik}\sigma_{kj}\sigma_{ik}\sigma_{kj} = \sigma_{ij}$, one can build up any σ_{ij}. □

Thus, in the case where $n = 3$, instead of considering $S = \{\sigma_{12}, \sigma_{13}, \sigma_{23}, \sigma_{21}, \sigma_{31}, \sigma_{32}\}$, the Lovász conjecture implies that a Hamiltonian cycle should exist using only $S = \{\sigma_1, \sigma_2, \sigma_3\}$.

Using the reduced generating set, the following 24 long sequence was found. When the sequence is repeated 7 times, it generates all 168 matrices in AGL(3,2).

$$\sigma_1\sigma_2\sigma_1\sigma_2\sigma_1\sigma_2\sigma_1\sigma_3\sigma_1\sigma_2\sigma_1\sigma_2\sigma_1\sigma_2\sigma_1\sigma_3\sigma_1\sigma_2\sigma_1\sigma_2\sigma_1\sigma_3\sigma_1\sigma_2$$

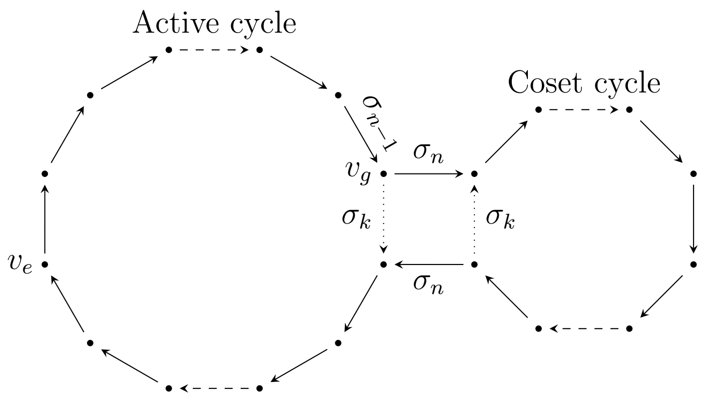
The cases where $n = 2, 3$ are special in that they cannot be found using the heuristic algorithm presented later.

6.4 Borel Subgroup

Removing one generator from $S = \{\sigma_i\}$, namely σ_n, leaves a set that generates the lower triangular matrices, since the rows can only add down. This subgroup, also known as the Borel group, is one generator away from GL$_n$(2). Hamiltonian cycles on cosets of this subgroup could be pieced together to form a Hamiltonian cycle for GL$_n$(2). This is how the heuristic algorithm to be presented later will work.

THEOREM **6.4.1.** *Let B_n be the lower triangular subgroup of $GL_n(2)$ and let $S = \{\sigma_i \text{ for } i \in [1..(n-1)]\}$. Then the Cayley graph $\Gamma(B_n, S)$ has a Hamiltonian cycle.*

Figure 6.1: Extending the Active Cycle



Proof. For $n = 3$, one can verify that $[\sigma_1\sigma_2\sigma_1\sigma_2\sigma_1\sigma_2\sigma_1\sigma_2]$ is a Hamiltonian cycle.

By induction, we assume that a Hamiltonian cycle exists for B_n. Identify B_n with the subgroup of B_{n+1} that is zero everywhere off the diagonal in the final row. The cosets of B_n partition B_{n+1}. Each coset can be traversed using the Hamiltonian cycle for B_n.

Note that σ_n commutes with all of the generators in B_n except for σ_{n-1}. The following process may be used to generate a Hamiltonian cycle for all of B_{n+1}.

1. Start with the Hamiltonian cycle associated with the subgroup B_n. This will be the first state of what will be referred to as the *active cycle*. Note that all the cosets of B_n may also be traversed by the same transition order as the Hamiltonian cycle for B_n. The goal is to connect each coset cycle to the active cycle, eventually enlarging the active cycle to cover all of B_{n+1}.

2. Starting at the identity, e, follow the Hamiltonian path on the active cycle and test each vertex by applying σ_n to see if σ_n transitions to a vertex not on the active cycle.

3. Let v_g be the first vertex on the active cycle such that $v_{g\cdot\sigma_n}$ is not on the active cycle. Since S is a generating set, the only time this can fail is when we have generated the entire Hamiltonian cycle. (Note: In the case where transition away from the active cycle from the initial point e, one of the incoming or outgoing edges to e must not be σ_{n-1}. Thus, the rest of the argument will still apply.)

4. Since σ_n commutes with all of the generators for B_n except σ_{n-1}, the edge transitioning into v_g must be σ_{n-1}. See figure 6.1. Otherwise, commutativity would imply that we would have transitioned away from the active cycle on the previous step.

5. Let σ_k be the edge transitioning out of v_g on the active cycle. Then $k \neq n$, since σ_n must be taking us to a new coset cycle. And $k \neq n-1$, since $\sigma_{n-1}^2 = e$, and the vertices are distinct on a Hamiltonian path. Thus $\sigma_n\sigma_k = \sigma_k\sigma_n$.

6. By corollary 6.2.3, there is a Hamiltonian cycle over the coset and by corollary 6.2.2, we may rotate the cycle so that the last edge is σ_k.

7. Since the coset cycle is a Hamiltonian cycle, the product of all the transitions is the identity (return to the starting point). Thus, if we do not complete the last transition, the product of all the previous transitions must by σ_k^{-1} which is simply σ_k since it has order 2.

8. Extend the active cycle to include the points on the coset cycle by branching out from the active cycle at v_g. Continue around the Hamiltonian cycle from the coset, but stopping before applying the final transition σ_k. Return back to the active cycle via σ_n. The total action is equivalent to $\sigma_n\sigma_k\sigma_n$ which is

equivalent to σ_k due to commutativity. Thus we return at exactly the next vertex on the active cycle and can now complete the trip back to v_e.

9. If the active cycle now covers all of B_{n+1} stop. Otherwise, return to step 2 to extend the active cycle by another coset cycle.

□

6.5 Heuristic Algorithm

Although the evidence is clearly in favor of a Hamiltonian cycle existing for $\mathrm{GL}_n(2)$, the author was unable to find a solution. However, the following heuristic algorithm was used to find Hamiltonian cycles over $\mathrm{GL}_n(2)$ for $n = 4, 5$. In this case note that $\sigma_n = \sigma_{n,1}$. The final σ_n adds the nth position to the first position.

The only difference between this algorithm and the algorithm for finding Hamiltonian cycles over the Borel group is that we cannot prove that this algorithm succeeds. It is possible to build an active cycle that cannot be extended, but is also not the full Hamiltonian cycle.

1. Start with the Hamiltonian cycle associated with the Borel subgroup B_n.

2. Apply σ_n to each vertex v_g. If σ_n transitions to a coset not contained in the active cycle, and the incoming and outgoing transitions for v_g are not both σ_1 and σ_{n-1}, then we have found a vertex from where we can extend the active cycle. NOTE: Such a vertex may not exist. This is where the algorithm could fail.

3. Let σ_k be the transition where $k \neq 1, n-1$. If σ_k is an incoming transition, then choose the vertex immediately preceding v_g to be our branching point.

4. In exactly the same manner as the previous algorithm, insert the new coset cycle in the active cycle in the place of σ_k.

5. If the active cycle now covers all of $\mathrm{GL}_n(2)$ stop. Otherwise, return to step 2 to extend the active cycle by another coset cycle.

6.6 Open Problems

- Prove that a Hamiltonian cycle exists over $\mathrm{GL}_n(2)$ using only row additions. This could include all possible row additions, not just the reduced set considered here.

- Find a "Gray Code" style algorithm for the Hamiltonian cycle. The current method only finds a path whose description is essentially a long description of what move to do at each step. For Gray Code, there is a function $f(t)$ which indicates which bit should be flipped at time t. The ideal answer would be a simple function $f(t) = (a, b)$ indicating the next step would be to add row a to row b.

Chapter 7

Double Coset Representatives

7.1 Results and Application

This chapter defines a **Basis Fixing Permutation (BFP)** as a permutation that fixes the zero vector and each vector of weight one. Each double coset contains at least one, and usually many BFPs. Additional reductions can be applied to find a minimal BFP from a given BFP. Unfortunately, many double cosets still contain multiple minimal BFPs.

This set of minimal BFPs is used to identify a given double coset. Once found, minimal BFPs can quickly discover which class a given permutation is in. Unfortunately, finding all the minimal BFPs is computationally intensive.

The major results of this chapter are:

- Every permutation is affine equivalent to a basis fixing permutation (BFP).
- Each coordinate polynomial of a BFP has a single linear term.
- Given a lexicographic ordering and a BFP, a minimal BFP can be quickly computed.

Affine equivalent representatives for each double coset are used to accelerate the identification algorithms.

7.2 Background

Let us first recall some basic terms used with linear and affine transformations. For the following definitions, let K be a field and the v_i's elements of a vector space over K.

Definition **7.2.1.** The **linear span** of m vectors $v_1, \ldots, v_m \in K^n$ is

$$\left\{ \sum_{i=1}^{m} a_i v_i \,\middle|\, a_i \in K \right\}.$$

Similarly, the **affine span** of $m+1$ vectors $v_0, \ldots, v_n \in K^n$ is

$$\left\{ \sum_{i=0}^{m} a_i v_i \,\middle|\, \sum_{i=0}^{m} a_i = 1, a_i \in K \right\}.$$

Definition **7.2.2.** A set of m vectors $v_1, \ldots, v_m$ is **linearly independent** if

$$\sum_{i=1}^{m} a_i v_i = 0 \implies \forall i, a_i = 0.$$

Similarly, a set of $n+1$ vectors $v_0, \ldots, v_n$ is **affinely independent** if

$$\sum_{i=0}^{n} a_i v_i = 0 \text{ and } \sum_{i=0}^{n} a_i = 0 \implies \forall i, a_i = 0.$$

Definition **7.2.3.** Given n, the **affine basis vectors** are $e_0 = \vec{0}$ and $e_1, \ldots, e_n$, where each e_i is a bit vector with zeros in every position except a 1 in the ith position, for i in $0, 1, \ldots, n$.

7.3 Basis Fixing Permutations

Definition **7.3.1.** A **Basis Fixing Permutation (BFP)** is a function that sends each affine basis vector to itself. Thus, f is basis fixing if $f(\vec{0}) = \vec{0}, f(\vec{e}_1) = \vec{e}_1 \ldots, f(\vec{e}_n) = \vec{e}_n$.

The first major reduction that can be made is realizing that every permutation is affine equivalent to a BFP. This was proved by the author and subsequently discovered to be in [Hou06].

Theorem **7.3.2.** *Every permutation in S_{2^n} is affine equivalent to a basis fixing permutation.*

Proof. Suppose there exists a set of $n+1$ affine independent vectors $x_0, x_1, \ldots, x_n$ such that $p(x_0), p(x_1), \ldots, p(x_n)$ is also affine independent. Then there exist affine maps a_0, a_1 such that for all i, $a_0(e_i) = x_i$ and $a_1(p(x_i)) = e_i$. Thus $a_1 p a_0(e_i) = e_i$ for all i and $a_1 p a_0$ is therefore basis fixing. Thus the proof reduces to finding an affine independent set that maps to an affine independent set via p.

We prove by induction that for every k in $0 \le k \le n$ there exists an affine independent set X_k such that $p(X_k)$ is also affine independent. Clearly for $k = 0, 1$ any X_k and $p(X_k)$ is affine independent. Assume that for $k < n$, X_k is affine independent. Let $\langle X_k \rangle_{af}$ denote the affine span of a $k+1$ element set X_k. Consider the permutation p, restricted as follows:

$$p : V^{2^n} \setminus \langle X_k \rangle_{af} \to V^{2^n} \setminus p(X_k).$$

Since

$$|p(V^{2^n} \setminus \langle X_k \rangle_{af})| + |V^{2^n} \setminus \langle p(X_k) \rangle_{af}| = 2(2^n - 2^k) > 2^n - (k+1) = |V^{2^n} \setminus p(X_k)|,$$

the intersection $p(V^{2^n} \setminus \langle X_k \rangle_{af}) \cap (V^{2^n} \setminus \langle p(X_k) \rangle_{af})$ is non-empty, and thus there exists an x not in the span $\langle X_k \rangle_{af}$ that maps to $p(x)$ not in the span of $\langle p(X_k) \rangle_{af}$. Adding x to X_k, we create a new set $X_{k+1} = X_k \cup x$, such that X_{k+1} and $p(X_{k+1})$ are both affinely independent.

Using the final X_n constructed in this manner, we can now find maps a_0 and a_1 so that $a_1 p a_0$ is a basis fixing permutation. ☐

LEMMA **7.3.3.** *Let p be a BFP. Then each coordinate function p_i of p has the form*

$$p_i(x_1, \ldots, x_n) = x_i + f_i(x_1, \ldots, x_n)$$

where f_i contains no constant or linear terms.

Proof. $p(e_0) = e_0 = \vec{0}$ implies no coordinate function can contain a 1. Given that all coordinate functions have no constant terms, $p(e_i) = e_i$ indicates that the linear term for each p_i is x_i. ☐

7.4 Basis Permuting Permutations

One important subgroup of $\mathrm{GL}_n(2)$ is the group of matrices with a single one in each row and column. This subgroup is isomorphic to S_n acting on $\{1, 2, \ldots, n\}$ via the injective map $\psi : S_n \to \mathrm{GL}_n(2)$ defined by the outer product sum

$$\psi(p) = \sum_{i=1}^{n} \left| e_{p(i)} \right\rangle \langle e_i | \, .$$

Thus $\psi(p)$ is the linear function that maps the basis vector $|e_i\rangle$ to the basis vector $\left|e_{p(i)}\right\rangle$ for each $i \in \{1, 2, \ldots, n\}$.

In $\mathrm{AGL}_n(2)$, there is a subgroup isomorphic to S_{n+1} acting on $\{0, 1, 2, \ldots, n\}$, which allows e_0 to be permuted with the basis vectors. We define this subgroup via the injective map $\phi : S_{n+1} \to \mathrm{AGL}_n(2)$ defined by

$$\phi(q) = \sum_{i=1}^{n} \left(\left| e_{q(i)} \right\rangle \langle e_i | + \left| e_{q(0)} \right\rangle \langle e_i | \right) + \left| e_{q(0)} \right\rangle .$$

The preceding affine form is applied to a vector $|v\rangle$ by first multiplying by the matrix component and then adding the vector component. Thus

$$\phi(q) |v\rangle = \sum_{i=1}^{n} \left(\left| e_{q(i)} \right\rangle \langle e_i | |v\rangle + \left| e_{q(0)} \right\rangle \langle e_i | |v\rangle \right) + \left| e_{q(0)} \right\rangle .$$

Definition **7.4.1.** A **Basis Permuting Permutation (BPP)** is a function that sends each affine basis vector to another. Thus, p is basis permuting if for all i in $0, 1, \ldots, n$, $f(\vec{e}_i) = \vec{e}_j$ for some j in $0, 1, \ldots, n$.

Lemma **7.4.2.** *All of the elements of $\phi(S_{n+1}) < AGL_n(2)$ are basis permuting permutations.*

Proof. Given $q \in Q$, verifying that $q(|e_j\rangle)$ maps to $\left| e_{q(j)} \right\rangle$ is sufficient to prove that q is a BPP. Recall that

$$\langle e_i | |e_j\rangle = \begin{cases} 1 & i = j \neq 0 \\ 0 & \text{otherwise.} \end{cases}$$

When q is evaluated at $e_0 = \vec{0}$, only the affine component $\left| e_{q(0)} \right\rangle$ does not zero out. The result $q(|e_0\rangle) = \left| e_{q(0)} \right\rangle$ is as expected. Evaluating q at e_j where $j \neq 0$ gives:

$$\begin{aligned}
q(|e_j\rangle) &= \left[\sum_{i=1}^{n}\left(\left|e_{q(i)}\right\rangle\langle e_i| + \left|e_{q(0)}\right\rangle\langle e_i|\right) + \left|e_{q(0)}\right\rangle\right](|e_j\rangle) \\
&= \sum_{i=1}^{n}\left(\left|e_{q(i)}\right\rangle\langle e_i|\,|e_j\rangle + \left|e_{q(0)}\right\rangle\langle e_i|\,|e_j\rangle\right) + \left|e_{q(0)}\right\rangle \\
&= \left|e_{q(j)}\right\rangle + \left|e_{q(0)}\right\rangle + \left|e_{q(0)}\right\rangle \\
&= \left|e_{q(j)}\right\rangle .
\end{aligned}$$

Thus, q permutes the affine basis vectors, and is therefore a BPP.

□

LEMMA **7.4.3.** *Any basis fixing permutation conjugated by a basis permuting permutation is a basis fixing permutation.*

Proof. Let q be a BPP and p be a BFP. Then for i in $\{0, 1, \ldots n\}$

$$\begin{aligned}
q^{-1}pq(|e_j\rangle) &= q^{-1}p(\left|e_{q(j)}\right\rangle) \\
&= q^{-1}(\left|e_{q(j)}\right\rangle) \\
&= \left|e_{q^{-1}(q(j))}\right\rangle \\
&= |e_j\rangle .
\end{aligned}$$

Since $q^{-1}pq$ fixes all affine basis vectors, $q^{-1}pq$ is a BFP. □

7.5 Minimal Basis Fixing Permutations

Conjugating by the elements in $Q < \mathrm{AGL}_n(2)$, we can construct many different BFPs from a given BFP. Determining which BFP is minimal depends on the lexicographic order chosen. For Gröbner bases, grevlex is preferred, since it is usually easier to compute an ideal with respect to that order.

Definition **7.5.1.** The **Graded Reverse Lexicographic** or **grevlex** is a polynomial ordering. The ordering of monomials is decided first by degree. If the degree is the same, it decides ties by the degree of the variables in reverse order.

Example **7.5.2.** Polynomials ordered by grevlex.

- $x_1x_3 < x_0x_1x_2$ since the degree$(x_1x_3) = 2 < 3 =$ degree$(x_0x_1x_2)$.
- $x_1x_2 < x_0x_3$ since the degrees are equal, but x_0x_3 has the higher numbered variable (x_3).
- $x_0x_1x_3x_4 < x_0x_2x_3x_4$ since the degrees are equal, but the second monomial contains the highest variable not common to both (x_2).

Definition **7.5.2.** The **Grevlex Permutation Order** is an ordering of BFPs. Compare two permutations p and q, by considering their coordinate functions $(p_1, \ldots, p_n)$ and $(q_1, \ldots, q_n)$. In the first case where p_i and q_i differ, the order is decided by the grevlex order between p_i and q_i.

For a given BFP p, there exists a minimal BFP $q^{-1}pq$ for $q \in Q$ according to the grevlex permutation order. Unfortunately, there can be many such minimal BFPs in a given double coset. The true minimal BFP would be the minimum of all such minimal BFPs.

For computational purposes, the author currently searches for all minimal BFPs, and then identifies the minimum BFP from this set. This information is stored in a table.

Algorithm for permutation identification:

1. Compute BFP from permutation.
2. Compute minimal BFP from BFP.
3. Look up minimum BFP associated with minimal BFP in table.

Once the table is created, permutation identification is quite fast. Unfortunately, constructing the table is computationally intensive.

7.6 Open Problems

- Find an efficient algorithm for computing the minimum BFP for a double coset.
- How efficiently can the parity of a permutation be computed from a BFP?
- Given a basis fixing function $F(x_1, \ldots, x_n) = (x_1 + f_1, \ldots, x_n + f_n)$, how difficult is it to determine whether F is a permutation?
- How hard is it to count the quadratic permutations?

Chapter 8

Multiple Rank Invariant

8.1 Results and Application

When comparing and categorizing permutations, the double coset test must be performed many, many times. If two permutations can be distinguished via a quick test, the double coset test is not needed.

This chapter develops a new rank invariant that can distinguish certain double cosets. The invariant essentially measures how much of a permutation is linear, quadratic, cubic, etc.

The major results of this chapter are:

- A proof that the multiple rank invariant is consistent with affine equivalent double cosets.
- An efficient algorithm for computing the multiple rank invariant.
- A categorization of how many equivalence classes are of each type for $n = 3, 4$.

The multiple rank invariant is used extensively to speed up the computation of the affine equivalence complexity tree.

8.2 Background

Every Boolean permutation can be decomposed into n coordinate functions $(f_1, \ldots, f_n)$. Each coordinate function $f_i : \mathrm{GF}(2)^n \to \mathrm{GF}(2)$ can be expressed as a polynomial over $\mathrm{GF}(x_1, \ldots, x_n)$. Since the value for each x_i is restricted to be exactly 0 or 1, the relationship $x_i^2 = x_i$ holds for each x_i. A polynomial will be called reduced if no variable has degree greater than 1.

8.3 The k-rank

Consider the finite vector space over GF(2) where the independent vectors are the 2^n reduced monomials from $\mathrm{GF}(x_1, \ldots, x_n)$, namely

$$1, \{x_i\}_{i=1}^{n}, \{x_i x_j\}_{i,j=1,i\neq j}^{n}, \ldots x_1 x_2 \cdots x_n.$$

Then similarly to how the rank of a set of vectors is calculated, we can calculate the rank of a set of polynomials.

Example **8.3.1.** Consider the following sets of polynomials.

- The set $\{x_1, x_2, x_1 + x_2\}$ has rank 2, since the third term is the sum of the first two.
- The set $\{x_1, x_2, x_1 x_2\}$ has rank 3, since all three terms are linearly independent.

This now leads to two important definitions.

Definition **8.3.1.** The **k-span** of the variables $X = \{x_1, \ldots, x_n\}$, is the linear span of all monomials of degree k or less. We will denote the k-span by X^k.

DEFINITION **8.3.2.** The **k-rank** of a set of polynomials $P = \{p_1, \ldots, p_m\}$ is $\operatorname{rank}(P) + \operatorname{rank}(X^k) - \operatorname{rank}(P \cup X^k)$. We will denote the k-rank by $R_k(P)$.

The k-rank essentially measures how much of the rank of P is due to polynomials of degree k or less. For the k-rank of a permutation $p = (f_1, \ldots, f_n)$, the set of polynomials will be the set of coordinate functions, i.e. $P = \{f_1, \ldots, f_n\}$.

EXAMPLE **8.3.3.** Consider the following sets of polynomials.

- Let $P_1 = \{x_1, x_2, x_1x_2\}$. Then $R_1(P_1) = 2$, since two of the terms can be expressed by linear polynomials. $R_2(P_1) = 3$ since all polynomials can be expressed by degree two or less polynomials.

- Let $P_2 = \{x_1 + x_1x_2, x_2 + x_1x_2, x_1x_2\}$. Then $R_1(P_2) = 2$ and $R_2(P_2) = 3$. This can be verified by calculation or recognizing that P_2 was derived from P_1 by adding the last term to each of the first two. It can also be seen by reducing by the monomial x_1x_2 and realizing that all of the remaining terms are linear.

EXAMPLE **8.3.4.** Consider the following permutations.

- Let $p_1 = (x_1, x_2, x_3 + x_1x_2)$. Then $P_1 = \{x_1, x_2, x_3 + x_1x_2\}$. As before, $R_1(P_1) = 2$ and $R_2(P_1) = 3$.

- Let $p_2 = (x_1 + x_3 + x_1x_2, x_2 + x_1x_3 + x_1x_2, x_3 + x_1x_2)$. Then $P_1 = \{x_1 + x_3 + x_1x_2, x_2 + x_1x_3 + x_1x_2, x_3 + x_1x_2\}$. In this case $R_1(P_1) = 1$ since $f_1 + f_3$ is linear and $R_2(P_1) = 3$ since all of the coordinate functions are at most quadratic.

8.4 The Invariance of the k-rank

LEMMA **8.4.1.** *All the coordinate functions of an n-bit permutation are linearly independent.*

Proof. Let p be a permutation, and let $P = \{f_1, \ldots, f_n\}$. Suppose P was linearly dependent. Then there would exist a non-zero linear combination L such that $L(f_1, \ldots, f_n) = 0$. Since L is a non-zero vector, it can be extended to an invertible matrix M where L is the first row of M. Note that since M is invertible, M is also a permutation.

Consider now the composition $M \circ P$. Since M and P are both permutations, $M \circ P$ must also be a permutation. However, if we consider the first coordinate function of $M \circ P$ it is $L(f_1, \ldots, f_n) = 0$. Thus $M \circ P$ cannot be a permutation, since every coordinate function of a permutation must be balanced. $\square$

THEOREM **8.4.2.** *The k-rank of a permutation is invariant under the action of $AGL_n(2)$ for all $k \geq 1$.*

Proof. Recall the definition of k-rank for a permutation p:

$$R_k(p) = \text{rank}(p) + \text{rank}(X^k) - \text{rank}(p \cup X^k)$$

We wish to show that $R_k(a \circ p \circ b) = R_k(p)$ for all $a, b \in \text{AGL}_n(2)$. This can be done in two parts. Note that a creates new coordinate functions by summing up various coordinate functions and b substitutes affine combinations of variables in place of the original variables.

First, consider the effect of a. By the previous lemma, ap must have full rank so $\text{rank}(ap) = \text{rank}(p)$. Also note that ap is the linear combination of the coordinate functions of p with the possible addition of the affine vector 1, which is contained in X^k for all k. Thus $\text{rank}(ap \cup X^k) = \text{rank}(p \cup X^k)$. Thus,

$$\begin{aligned} R_k(ap) &= \text{rank}(ap) + \text{rank}(X^k) - \text{rank}(ap \cup X^k) \\ &= \text{rank}(p) + \text{rank}(X^k) - \text{rank}(p \cup X^k) = R_k(p) \end{aligned}$$

Now consider the effect of b. Appealing to the previous lemma again, $\text{rank}(pb) = \text{rank}(p)$. Thus we only need to verify that $\text{rank}(pb \cup X^k) = \text{rank}(p \cup X^k)$.

Notice that the coordinate functions of pb are really the original coordinate functions $f_1, \ldots, f_n$ evaluated at $y_1, \ldots, y_n$ where $y_i = b_i(x_1, \ldots, x_n)$.

Let Y_k be the k-span of the y_i. Then $Y_k \subset X_k$ since any polynomial of degree $k+1$ is the sum of monomials of degree less than or equal to $k+1$. Since b is an invertible affine function, we can write each x_i as a affine function of $y_1, \ldots, y_n$. Thus, by the same argument $X_k \subset Y_k$ and therefore $X_k = Y_k$. Thus,

$$\text{rank}(pb \cup X^k) = \text{rank}(p|_Y \cup Y^k) = \text{rank}(p|_X \cup X^k = \text{rank}(p \cup X^k).$$

This implies,

$$R_k(pb) = \text{rank}(pb) + \text{rank}(X^k) - \text{rank}(pb \cup X^k) = \text{rank}(p) + \text{rank}(X^k) - \text{rank}(p \cup X^k) = R_k(p).$$

Since neither affine function a or b affects the rank, we can apply them in either order and deduce that $R_k(apb) = R_k(p)$.

Therefore the k-rank is invariant under affine equivalence. $\square$

8.5 Multiple Rank Invariant

Before defining the multiple rank invariant, it should be noted that no information is gained using the n-rank invariant.

Lemma **8.5.1.** *No n-bit permutation has a coordinate function with an $x_1x_2\cdots x_n$ term.*

Proof. Considering the truth table of $x_1x_2\cdots x_n$ for $n > 1$, it only takes a non-zero value at one point, namely $(1, 1, \ldots, 1)$. Every other monomial is non-zero at an even number of points. Since the coordinate function of a permutation must be balanced, and therefore have an even number of 0's and 1's (for $n > 1$), $x_1x_2\cdots x_n$ cannot be contained in any coordinate function because it would make the weight odd, and therefore not balanced. $\square$

Thus we only need to consider monomials of degree up to $n-1$ for permutation coordinate functions.

Definition **8.5.1.** The **multiple rank invariant** of an n-bit permutation is an $n-1$ tuple of k invariant differences. $I(p) = (R_1(p), R_2(p)-R_1(p), \ldots, R_{n-1}(p)-R_{n-2}(p)$.

Example **8.5.2.** Consider the following permutations.

- Let $p_1 = (x_1, x_2, x_3+x_1x_2)$. As computed earlier, $R_1(P_1) = 2$ and $R_2(P_1) = 3$. Thus the multiple rank invariant is the tuple $(2, 1)$.
- Let $p_2 = (x_1, x_1 + x_2, x_3 + x_1x_2x_4, x_4 + x_1x_2)$. By inspection, $R_1(P_2) = 2$, $R_2(P_2) = 3$ and $R_3(P_2) = 4$. Thus the multiple rank invariant is the tuple $(2, 1, 1)$.

8.6 Categorization of 3-bit Permutations by MRI

For the four 3-bit equivalence classes, the multiple rank invariant completely distinguishes each class.

Table 8.1: Multiple Rank Invariant of 3-bit Permutations

MRI	# of classes	Description
(3,0)	1	Affine
(2,1)	1	Toffoli
(1,2)	1	
(0,3)	1	

We will discover later that the multiple rank invariant also captures the relative complexity of all 3-bit permutations, ordering them in terms of those requiring 0,1,2 or 3 Toffoli gates.

8.7 Categorization of 4-bit Permutations by MRI

As seen in Table 8.2, the multiple rank invariant separates many of the 302 4-bit equivalence classes, but fails to distinguish among some of the most prevalent permutation types.

When the complexity tree is developed later, we will see that the multiple rank invariant has a rough correlation to the nonlinear complexity of a permutation.

8.8 Open Problems

For $n \leq 4$ the multiple rank invariant is always the same for p and p^{-1}, even when they are in different double cosets. Is this true for all n?

Table 8.2: Multiple Rank Invariant of 4-bit Permutations

MRI	# of Classes	Description
(4,0,0)	1	Affine
(3,1,0)	1	Toffoli
(3,0,1)	1	Triple controlled NOT
(2,2,0)	2	
(2,1,1)	3	
(2,0,2)	4	
(1,3,0)	3	
(1,2,1)	9	
(1,1,2)	20	
(1,0,3)	13	
(0,3,1)	5	
(0,2,2)	39	
(0,1,3)	127	
(0,0,4)	74	

Chapter 9

Additional Invariants

9.1 Results and Application

In addition to the multiple rank invariant, there are a handful of other minor invariants that can be used to refine the double coset identification process. This chapter presents a number of these invariants and their associated proofs.

The major results of this chapter are:

- The parity of a permutation is invariant under affine equivalence.
- Definition of a two-way permutation and its use as an invariant.

These additional invariants are used extensively to speed up computation of the affine equivalence complexity tree.

9.2 Parity

The parity of a permutation is often a useful characteristic, and it will play a central role in classifying the complexity of permutations.

LEMMA **9.2.1.** *Any permutation that does not involve all of the bit wires is an even permutation.*

Proof. Let p be a permutation that does not involve a certain wire. Without loss of generality, we can assume that the high bit is not involved in the permutation. Then p can be decomposed into two disjoint cycle sets: those whose high bit is 0, and those whose high bit is 1. Furthermore, since the high bit is not involved in p, the two cycle sets must have exactly the same structure. Thus the overall permutation must be even. $\square$

COROLLARY **9.2.2.** *For $n > 2$, the parity of a permutation is invariant under affine equivalence.*

Proof. All affine permutations are generated by the NOT gate on any single bit, and by CNOT gates between any two bits. When $n > 2$, there is at least one bit that is not involved in the action of the NOT or the CNOT. Thus by Lemma 9.2.1, CNOT and NOT are both even permutations, and therefore any affine permutation is an even permutation. $\square$

As proved earlier, there are an equal number of even and odd double cosets. The parity test is independent of the multiple rank invariant, distinguishing some permutations with identical MRIs.

9.3 Two-Way Permutations

A one-way permutation is a permutation which is easy to compute, but its inverse is hard to compute. Proving the existence of one-way permutations has proven extremely difficult, and is closely related to the question of $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$. We will not be solving this problem here, but rather introduce the notion of the opposite of a one-way permutation.

Definition **9.3.1.** A permutation p, is a **two-way permutation**, if there exist $a, b \in \mathrm{AGL}_n(2)$ such that $apb = p^{-1}$.

Thus a two-way permutation is affine equivalent to its inverse. This implies that if a method was found to compute p (respectively p^{-1}) faster, that would automatically yield a method for computing p^{-1} (respectively p) faster. Thus no two-way permutation has a chance of being a one-way permutation.

Lemma **9.3.2.** *If p is a two-way permutation, then every permutation affine equivalent to p is a two-way permutation.*

Proof. Assume p is a two-way permutation. Then there exist $a, b \in \mathrm{AGL}_n(2)$ such that $apb = p^{-1}$. Consider any other element in the affine double coset of p, having the form gph. Then

$$(gph)^{-1} = h^{-1}p^{-1}g^{-1} = h^{-1}apbg^{-1} = h^{-1}ag^{-1}(gph)h^{-1}bg^{-1}.$$

Thus the inverse of gph is affine equivalent to gph. Therefore gph is a two-way permutation. □

Thus if an affine equivalent double coset contains a two-way permutation, then the entire double coset is made up of two-way permutations. This implies that "two-way-ness" is invariant over the double coset. Now let us consider the relationship between involutions and two-way permutations.

Lemma **9.3.3.** *If a permutation p affine equivalent to an involution, then p is a two-way permutation.*

Proof. Suppose p is in a double coset containing an involution i. Then there exist $a, b \in \mathrm{AGL}_n(2)$ such that $apb = i$. Thus,

$$(apb)(apb) = i^2 = 1$$

$$pbapba = 1$$

$$bapba = p^{-1}.$$

Therefore, p is affine equivalent to p^{-1} and is therefore a two-way permutation.

□

It is unknown if the converse is true. Table 9.1 shows how the parity and two-way invariants further refine permutation identification.

9.4 Open Problems

- Is every two-way permutation affine equivalent to an involution? For $n \leq 4$ it is true.
- Are there other useful invariants?

Table 9.1: Invariant Table for 4-bit Permutations

	Two-way		not Two-way	
MRI	Even	Odd	Even	Odd
(4,0,0)	1			
(3,1,0)	1			
(3,0,1)		1		
(2,2,0)	2			
(2,1,1)	1	2		
(2,0,2)	3	1		
(1,3,0)	3			
(1,2,1)	1	4	4	
(1,1,2)	11	3	2	4
(1,0,3)	4	9		
(0,3,1)		3	2	
(0,2,2)	12	3	10	14
(0,1,3)	14	35	42	36
(0,0,4)	32	26	6	10

Chapter 10

Complexity Theory

10.1 Results

The goal of this chapter is to illustrate the usefulness of determining the complexity of a function by measuring the nonlinearity of a function. The major results of this chapter are as follows:

- Definition of new complexity class based on nonlinearity of a computation.
- Proofs relating nonlinear complexity to AC and NC.
- Proofs relating the classical and nonlinear versions of P and $\mathsf{P/poly}$.

10.2 Background

An excellent introduction to the complexity classes relevant to circuits is given in [Vol99]. This section provides important complexity class definitions from that book.

To determine a circuit class, we first choose a basis for constructing functions. Let us consider two of the most common bases.

DEFINITION **10.2.1.** $B_0 = \{\neg, \vee^2, \wedge^2\}$ is the *standard bounded fan-in basis*. $B_1 = \{\neg, (\vee^n)_{n\in\mathbb{N}}, (\wedge^n)_{n\in\mathbb{N}}\}$ is the *standard unbounded fan-in basis*.

In B_0 you can only take the AND or OR of two values, whereas in B_1 you may take the AND or OR of as many wires as you wish. The *depth* of a circuit is the length of the longest path from the input bits to the output bits.

DEFINITION **10.2.2.** Let B be a basis and let $s, d : \mathbb{N} \to \mathbb{N}$. We define the following complexity classes:

1. $\mathsf{SIZE}_B(s)$ is the class of all sets $A \subset \{0,1\}^*$ for which there is a circuit family C over basis B of size $O(s)$ that accepts A as an input.

2. $\mathsf{DEPTH}_B(d)$ is the class of all sets $A \subset \{0,1\}^*$ for which there is a circuit family C over basis B of depth $O(d)$ that accepts A as an input.

3. $\mathsf{SIZE\text{-}DEPTH}_B(s,d)$ is the class of all sets $A \subset \{0,1\}^*$ for which there is a circuit family C over basis B of size $O(s)$ and depth $O(d)$ that accepts A as an input.

Two important complexity classes capture the notion of what circuits can be parallelized efficiently: NC and AC.

DEFINITION **10.2.3.** For $i \geq 0$, define

$$\mathsf{NC}^i = \mathsf{SIZE\text{-}DEPTH}_{B_0}(n^{O(1)}, (\log n)^i),$$

and let

$$\mathsf{NC} = \bigcup_{i\geq 0} \mathsf{NC}^i.$$

Similarly, for $i \geq 0$, define

$$\mathsf{AC}^i = \mathsf{SIZE\text{-}DEPTH}_{B_1}(n^{O(1)}, (\log n)^i),$$

and let

$$\mathsf{AC} = \bigcup_{i \geq 0} \mathsf{AC}^i.$$

Thus NC and AC both represent polynomial sized circuits with polylog depth. The only difference is AC allows unbounded fan-in (using basis B_1) as opposed to bounded fan-in for NC (using basis B_0). The following two results are well known in complexity theory.

Theorem **10.2.4.** *For $i \geq 0$,*

$$\mathsf{NC}^i \subseteq \mathsf{AC}^i.$$

Corollary **10.2.5.**

$$\mathsf{NC} = \mathsf{AC}.$$

For a proof of these results, see [Vol99].

10.3 Nonlinear Complexity Class

Similar to the notion of size and depth for classical circuits, we introduce the notions of width and depth for nonlinear reversible circuits. We also need to introduce the notion of a nonlinear basis. Recall that T_{ijk} is a Toffoli gate with control wires i, j acting on the target wire k. Also recall that (i, j, A) represents the controlled affine permutation with i control wires dictating whether or not the $j \times j$ affine transformation A is applied to the next j wires.

Definition **10.3.1.** Let C be a reversible circuit acting on n wires. Then $B_T = \{T_{ijk}$ where $i, j, k \in [0, n)\}$ is the *Toffoli basis* for C. $B_{CA} = \{(i, j, A)$ where $i \in [1, n-1), j \in [1, n-i)$ and $A \in \mathrm{AGL}_n(j)\}$ is the *controlled affine basis* for C.

The *depth* of a nonlinear circuit will be the number of times the nonlinear basis had to be used. Note that multiple nonlinear basis gates could be used in a single round as long as they were operating on independent wires.

Example **10.3.2.** The 6-bit permutation computing $T_{0,1,2}$ and $T_{3,4,5}$ could be done in one round of nonlinearity since the two Toffoli gates are acting on independent sets of wires.

Definition **10.3.2.** Let B be a nonlinear basis and let $w, d : \mathbb{N} \to \mathbb{N}$. We define the following complexity classes:

1. $\mathsf{WIDTH}_B(s)$ is the class of all functions $f : \{0,1\}^* \to \{0,1\}^*$ for which there is a reversible circuit family C over basis B using $O(w)$ wires that computes f.

2. $\mathsf{DEPTH}_B(s)$ is the class of all functions $f : \{0,1\}^* \to \{0,1\}^*$ for which there is a reversible circuit family C over basis B using $O(d)$ rounds of B that computes f.

3. $\mathsf{WIDTH\text{-}DEPTH}_B(s)$ is the class of all functions $f : \{0,1\}^* \to \{0,1\}^*$ for which there is a reversible circuit family C over basis B using $O(w)$ wires and $O(d)$ rounds of B that computes f.

In the same spirit in which NC and AC were defined, we now define the class of nonlinear circuits with polynomial width and polylog depth.

Definition **10.3.3.** For $i \geq 0$, define

$$\mathsf{NLC}^i = \mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^{O(1)}, (\log n)^i),$$

and let

$$\mathsf{NLC} = \bigcup_{i\geq 0} \mathsf{NLC}^i.$$

Note that NLC is defined using the controlled affine basis B_{CA}. Another complexity class similar in nature to NC^i can be defined using the Toffoli basis B_T.

Although NLC may seem overpowered, given its ability to perform arbitrary affine transformations between every nonlinear step, we will see that NLC^i fits nicely within the AC hierarchy.

Lemma **10.3.4.**

$$\mathsf{AC}^i \subseteq \mathsf{NLC}^i.$$

Proof. Let C be a circuit in AC^i with $s(n)$ gates and $d(n)$ depth. Since there are only $s(n)$ gates in the circuit, there are at most $s(n)+n$ different wires to join in a gate at any given time: n for the inputs and $s(n)$ for the output of each gate. Thus the maximal fan-in gate for the circuit has less than $s(n)+n$ inputs.

Construct a nonlinear circuit of width $w(n) = (s(n)+1)(s(n)+n)$. This space will be allocated according to table 10.1.

Table 10.1: Space Requirements

Size	Description
n	Initial input
$s(n)(s(n)+n)$	Wires to hold input for each of the $s(n)$ gates.
$s(n)$	Wires to hold output for each of the $s(n)$ gates.

To populate the circuit with gates, we will follow the flow of the original circuit C. Let G_i be the set of gates computed at depth i in the circuit C. For each gate $g \in G_i$, we construct it using the following steps.

1. Using CNOTs, copy the value of the inputs to g to the input staging area allocated for g.
2. If g is an OR gate, negate all input wires.
3. Apply a multiple controlled NOT from the inputs of g to the output wire allocated for g.
4. If g is an OR gate, negate the output wire.

Note that step 3 is the only nonlinear step. All CNOTs and negations are affine functions, and can be computed in the linear round between each nonlinear round.

Furthermore, since each $g \in G_i$ only depends on values from earlier rounds, all $g \in G_i$ can be computed in the same nonlinear round. Thus the nonlinear circuit will also have depth $d(n)$.

Since $C \in \mathsf{AC}^i$, $s(n)$ is polynomial. Thus $w(n) = (s(n)+1)(s(n)+n)$ is also polynomial. Also, $C \in \mathsf{AC}^i$ implies $d(n) = O((\log n)^i)$. Therefore, $\mathsf{AC}^i \subseteq \mathsf{NLC}^i$. $\square$

Lemma **10.3.5.**

$$\mathsf{NLC}^i \subseteq \mathsf{NC}^{i+1}.$$

Proof. Recall that

$$\mathsf{NLC}^i = \mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^{O(1)}, (\log n)^i).$$

This implies

$$\mathsf{NLC}^i = \bigcup_{k \geq 0} \mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^k, (\log n)^i).$$

We will prove that for each k,

$$\mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^k, (\log n)^i) \subseteq \mathsf{NC}^{i+1}.$$

The construction of NLC^i by union will then imply that $\mathsf{NLC}^i \subseteq \mathsf{NC}^{i+1}$.

Let C be a nonlinear circuit in $\mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^k, (\log n)^i)$. Then C consists of $O((\log n)^i)$ rounds of alternating linearity and nonlinearity on a width of $O(n^k)$ wires. Let us consider the depth in NC terms of the linear and nonlinear rounds separately.

In each linear round, an affine matrix acts on the $O(n^k)$ wires. Since each output wire of the affine transformation, is essentially the parity of some subset of the wires, and PARITY is in NC^1 (polynomial gates, log depth), the affine transformation can be computed with polynomial many gates and $O(\log(n^k)) = O(k \log n) = O(\log n)$ depth.

In each nonlinear round, multiple controlled affine transformations are applied. For each wire i compute the output value a_i as if the affine transformation it is associated with was applied. Similar to the reasoning above, this can be done in $O(\log n)$ depth and polynomial gates. Also, for each i compute the control value c_i which is the AND of the control wires for the affine function associated with i. The AND of $O(n^k)$ inputs can also be computed in $O(\log n)$ depth in NC. Let n_i be the original value of wire i with no transformation applied. The correct output for wire i is then

$$(n_i \land \neg c_i) \lor (a_i \land c_i).$$

Examination verifies that if all the controls are 1, the output of wire i will be a_i, and otherwise n_i. This final computation only requires depth 2, thus the total depth for the nonlinear round is $O(\log n)$.

Since the nonlinear circuit has $O((\log n)^i)$ linear and nonlinear rounds, and each round has NC depth $O(\log n)$, the entire circuit will have NC depth $O((\log n)^i)\cdot O(\log n) = O((\log n)^{i+1})$.

Since the entire circuit uses only polynomially many gates,

$$\mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^k, (\log n)^i) \subseteq \mathsf{NC}^{i+1}.$$

Furthermore, since this inclusion holds for all k,

$$\mathsf{NLC}^i \subseteq \mathsf{NC}^{i+1}.$$

□

Theorem **10.3.6.**

$$\mathsf{NLC} = \mathsf{AC} = \mathsf{NC}.$$

Proof. Combining the two previous lemmas, we get

$$\mathsf{NC}^i \subseteq \mathsf{AC}^i \subseteq \mathsf{NLC}^i \subseteq \mathsf{NC}^{i+1} \subseteq \mathsf{AC}^{i+1} \subseteq \mathsf{NLC}^{i+1}.$$

Since NC, AC and NLC are all infinite unions, this is sufficient to prove $\mathsf{NLC}=$ $\mathsf{AC}=$ NC. □

10.4 Nonlinear Polynomial Complexity

Consider now the complexity class with polynomial width and a polynomial number of nonlinear rounds.

Theorem **10.4.1.**

$$\mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^{O(1)}, n^{O(1)}) = \mathsf{SIZE}_{B_1}(n^{O(1)}).$$

Proof. For each k, let C be a circuit class in $\mathsf{SIZE}_{B_1}(n^k)$. Convert every AND and OR gate into NAND gates using NOT gates on the inputs and outputs as needed. Let $s(n)$ be the number of NAND gates necessary to compute C. Construct a nonlinear circuit of width $n + s(n)$. The first n wires will contain the function input, and the last $s(n)$ will be used to hold the output from each of the $s(n)$ NAND gates. Populate the circuit by computing the NAND gates in the same order as for the circuit computation, storing each output in the $s(n)$ available positions. In the worst case, each NAND gate must be computed in a separate nonlinear depth. Thus the depth of the nonlinear circuit is also $O(n^k)$. This circuit conversion implies

$$\mathsf{SIZE}_{B_1}(n^k) \subseteq \mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^k, n^k).$$

Since this containment holds for any k,

$$\mathsf{SIZE}_{B_1}(n^{O(1)}) \subseteq \mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^{O(1)}, n^{O(1)}).$$

Alternatively, for each k let C be a circuit class in $\mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^k, n^k)$. Let us consider the number of gates required in terms of the linear and nonlinear rounds separately.

In the linear rounds, we need to compute a matrix multiply followed by an affine shift. This can be done in $O((n^k)^2) = O(n^{2k})$ gates.

Similarly, in the nonlinear rounds, compute the potential affine output for each bit a_i for each wire i. This also requires $O((n^k)^2) = O(n^{2k})$ gates. Also, for each i compute the control value c_i which is the AND of the control wires for the affine function associated with i. Finally, let n_i be the original value of wire i with no transformation applied. The correct output for wire i is then

$$(n_i \wedge \neg c_i) \vee (a_i \wedge c_i).$$

Using $O(n^{2k})$ gates for each of the $O(n^k)$ wires potentially requires $O(n^{3k})$ gates.

Combining together all $O(n^k)$ rounds of nonlinearity, we require $O(n^{3k}) \cdot O(n^k) = O(n^{3k}$ gates. Thus,

$$\mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^k, n^k) \subseteq \mathsf{SIZE}_{B_1}(n^{4k}).$$

Since this containment holds for any k,

$$\mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^{O(1)}, n^{O(1)}) \subseteq \mathsf{SIZE}_{B_1}(n^{O(1)}).$$

Combining the two containments, we now conclude

$$\mathsf{WIDTH\text{-}DEPTH}_{B_{CA}}(n^{O(1)}, n^{O(1)}) = \mathsf{SIZE}_{B_1}(n^{O(1)}).$$

□

10.5 Uniform and Non-Uniform Nonlinear Circuits

Theorem 10.4.1 states that any nonlinear circuit with polynomial width and depth can be converted into a circuit with polynomial gates. We can therefore derive a relation between nonlinear circuits and the complexity classes P and $\mathsf{P/poly}$.

The difference between uniform and non-uniform polynomial circuits is subtle yet important. For a circuit class to be *uniform*, all circuits must be described by a finite algorithm, or finite set of instructions. Even though the circuit generated by the algorithm may get larger and larger as n increases, the same finite description is used to generate every circuit.

For *non-uniform* circuit classes, there is no blueprint. Each circuit may be different. Even if each circuit is polynomially sized, the collective description of the circuits must be infinite.

DEFINITION **10.5.1.** Define the following classes:

- P is the class of uniform circuits that run in polynomial time.
- $\mathsf{P/poly}$ is the class of non-uniform circuits that run in polynomial time.
- $\mathsf{nonlinearP}$ is the class of uniform circuits with polynomial width and rounds of nonlinearity,
- $\mathsf{nonlinearP/poly}$ is the class of non-uniform circuits with polynomial width and rounds of nonlinearity,

COROLLARY **10.5.2.**

$$\mathsf{nonlinearP} = \mathsf{P} \textit{ and } \mathsf{nonlinearP/poly} = \mathsf{P/poly}.$$

Proof. Using theorem 10.4.1 we can convert back and forth between polynomial circuits and nonlinear circuits with polynomial width and rounds of nonlinearity. Since the conversion description is finite, any uniformity will be preserved. □

10.6 Open Problems

The results of this chapter show that measuring the complexity of a function or algorithm via its nonlinearity captures most of the essence of "polynomialness". It is hoped that this point of view will be useful in developing new complexity proofs.

As quantum computing was part of the inspiration for finding optimal reversible computation, it is interesting to consider what nonlinearity means in the quantum computing context. For quantum computing, only the CNOT and arbitrary single unitary gates on qubits are needed for computation. It is unclear to the author what subset of this computation should be chosen to form the affine equivalence.

Chapter 11

Computation

Due to the doubly exponential nature of boolean permutations, extremely efficient computational methods are needed to perform many of the calculations in this paper. This chapter outlines a number of the more important methods used. To describe any function $f : V^n \to V^n$, $n2^n$ storage bits are required. Even though n-bit permutations theoretically require only $\lceil \log_2(2^n!) \rceil \approx n2^n - 2^n$ storage bits, it is useful to store the permutations using all $n2^n$ bits, especially since there are cases where we are unsure whether or not we have a permutation. Each of the following algorithms will assume that all $n2^n$ bits are in a single multi-precision integer.

Note that for $n = 4$, $n2^n = 64$, and thus a 4-bit permutation may be stored as a single word on a 64-bit machine. Thus the computation time increases significantly when extending beyond 4-bit permutations.

11.1 Storage of Permutations

There are multiple ways to store bit permutations, and there are algorithmic reasons for preferring each. In this section, we will discuss the three types used in this paper and how to convert between them.

11.1.1 Truth Table Form

We can always describe a permutation using its truth table. Recall the example permutation considered earlier:

Table 11.1: Truth Table: $f = (a_2 \oplus a_0a_1, a_1 \oplus 1, a_2 \oplus a_0a_1 \oplus a_0)$

$a_0a_1a_2$	$b_0b_1b_2$
0 0 0	0 1 0
0 0 1	1 1 1
0 1 0	0 0 0
0 1 1	1 0 1
1 0 0	0 1 1
1 0 1	1 1 0
1 1 0	1 0 0
1 1 1	0 0 1

Given the values in a particular order, storing the inputs is unnecessary and we only need to store the output of the permutation in the correct order. Since the author works on a little endian architecture, it was easier to store the least significant bits on the right. Thus the truth table is stored in the following order:

$$f(111)f(110)f(101)f(100)f(011)f(010)f(001)f(000).$$

Thus the permutation is stored as the binary word

$$w = 001100110011101000111\underbrace{010}_{f(000)}.$$

Extract the value $f(i)$ shifting w by $i * n$ bits to the right and then masking off the right n bits (or low n bits). Note that

$$(1 << n) - 1 = 2^n - 1 = \underbrace{1111111 \cdots 1_2}_{n \text{ ones in binary}} .$$

Thus to extract the value $f(i)$ from the word w, we execute:

```
f(i) = ( w >> (i*n) ) & ((1<<n)-1)
```

11.1.2 Function Table Form

Another way to look at a permutation is essentially turning the truth on its side. Each row is then the truth table of coordinate function. Again, due to little endian-ness, we store the coordinate function of the least significant bit function on the right, i.e. $f_2 f_1 f_0$. Each bit function will also have its least significant bits to the right. Thus for our example, each f_i will be in the following order:

$$f_i(111)f_i(110)f_i(101)f_i(100)f_i(011)f_i(010)f_i(001)f_i(000).$$

The example permutation will be stored as follows:

$$w = 1001101000110011\underbrace{01101010}_{f_0}.$$

Obtain the truth table for the coordinate function f_i shifting w by $i * 2^n$ bits to the right and then masking off the low 2^n bits.

Thus to extract the value $f(i)$ from the word w, we execute:

```
f_i = ( w >> ( i*(1<<n) ) ) & ( ( 1 << (1<<n) ) - 1 )
```

11.1.3 Polynomial Form

Since any boolean function can be represented as a polynomial, we can also specify each coordinate function by indicating which terms are present in the polynomial.

Given i where $0 \leq i < 2^n$, i can represent each possible monomial by considering the binary expansion. The kth bit of i indicates whether or not the monomial contains x_k in the product. In the special case where $i = 0$, the monomial is just 1. An example conversion is given in table 11.2.

Table 11.2: Monomial Conversion Table

Position	Binary	Monomial
0	0 0 0	1
1	0 0 1	a_0
2	0 1 0	a_1
3	0 1 1	a_1a_0
4	1 0 0	a_2
5	1 0 1	a_2a_0
6	1 1 0	a_2a_1
7	1 1 1	$a_2a_1a_0$

Recall that the example permutation has coordinate functions:

$$b_0 = a_2 \oplus a_0a_1 \oplus 1$$

$$b_1 = a_1$$

$$b_2 = a_2 \oplus a_0a_1 \oplus a_0$$

Thus, coordinate function b_0 will have 1's in the positions $0, 3, 4$, and would have binary form 00011001.

Again the polynomials will be stored with the least significant bit functions on the right so the example permutation would be stored as:

$$0001101000000100\underbrace{00011001}_{b_0}.$$

Obtain the polynomial for the coordinate function f_i shifting w by $i * 2^n$ bits to the right and then masking off the low 2^n bits.

Thus to extract the polynomial b_i from the word w, we execute:

```
p_i = ( w >> ( i*(1<<n) ) ) & ( ( 1 << (1<<n) ) - 1 )
```

11.2 Quick double coset test

Recall that two permutations p and q are affine equivalent if there exist affine functions a_0 and a_1 such that $p = a_0 q a_1$. Thus testing two permutations for equivalence is an instance of the Double-Coset Membership (DCM) problem. Generic methods for solving DCM proved too slow for our application, so a DCM test specifically for affine equivalence was developed.

11.2.1 Composition via a Hamiltonian Path

Note that $p = a_0 q a_1$ implies $a_0^{-1} = q a_1 p^{-1}$ or simply that $q a_1 p^{-1}$ is an affine function. Computing two successive permutation compositions is quite demanding, especially if we have to exhaust over all values for a_1. We will try to avoid calculating full compositions by computing $q a_1$ for successive values of a_1 using a gray-code type ordering, and then testing $q a_1 p^{-1}$ for linearity.

Using the Hamiltonian cycles developed in Chapter 6, we can successively step through the values qa_1 for all $a_1 \in \mathrm{AGL}_n(2)$. Each step on the Hamiltonian cycle is a CNOT, which can be implemented using a mask, shift and XOR. Each qa_1 is then composed with p^{-1} and tested for linearity.

11.2.2 Linearity Test

Given a permutation, how difficult is it to determine whether or not the permutation is an affine permutation? Since most of the linear tests are expected to fail, the strategy used for the double coset test tries to find failures quickly.

Instead of computing the entire composition of qa_1p^{-1}, we can compute only 4 values of the truth table, and verify that they satisfy the expected affine relation.

EXAMPLE **11.2.1.** Suppose t was an affine 3-bit permutation. Then

$$t(000) + t(001) + t(100) + t(101) = 000.$$

Essentially, any affine relation that held before t was applied must also hold after t was applied.

For 4-bit permutations, this test discovers 12/13 of nonlinear permutations. (There are 13 possibilities for the fourth value, of which only one satisfies the affine relation.) This fast failure allows us to avoid computing an entire permutation composition most of the time.

11.3 Conclusion

The specialized double coset membership test allows the entire 4-bit permutation space to be explored in a matter of hours, as opposed to early implementations which had an expected runtime of months.

Chapter 12

Classification of 3-bit and 4-bit Permutations

12.1 Summary

The machinery developed in the preceding chapters now gives us the opportunity to examine the nonlinear complexity of various permutations. We will first define some common permutations, and then compute their nonlinear complexity. The assortment of problems chosen aims to provide some insight into what problems are more difficult to compute.

12.2 Permutations of Interest

An n-bit permutation permutes 2^n elements. Some permutations are naturally defined on sets whose size is not exactly a power of two. We are still interested in the complexity of these permutations. In this initial classification of the nonlinear complexity, we will consider permutations to be fixed in the positions which they do not specify.

To save space, we will describe the bit permutations using integers instead of binary. I.e. $6 = 110_2$. Thus, 3-bit permutations will permute the set

$\{0, 1, 2, \ldots, 7\}$ and 4-bit permutations will permute the set $\{0, 1, 2, \ldots, 15\}$.

12.2.1 Modular Addition

Given a positive integer m where $2^{n-1} < m \leq 2^n$ and an integer a where $0 \leq b < m$, define an n-bit permutation π by

$$\pi(x) = \begin{cases} x + b \mod m & \text{if } x < m \\ x & \text{otherwise} \end{cases}$$

Such a permutation will be referred to as the permutation $x + b \mod m$.

Although we will consider π to be constant for values of $x \geq m$, it may be advantageous to allow some permutation of the upper values in finding a permutation with minimal complexity. For simplicity, this paper will only consider permutations with fixed values outside of the modular range.

Example **12.2.1.** The 3-bit permutation $x + 2 \mod 6$.

x	0	1	2	3	4	5	6	7
$x + 2 \mod 6$	2	3	4	5	0	1	6	7

12.2.2 Modular Multiplication

Given a positive integer m where $2^{n-1} < m \leq 2^n$ and an integer a where $\gcd(a, m) = 1$, define an n-bit permutation π by

$$\pi(x) = \begin{cases} ax \mod m & \text{if } x < m \\ x & \text{otherwise} \end{cases}$$

Such a permutation will be referred to as the permutation $ax \mod m$.

Example **12.2.2.** The 3-bit permutation $2x \mod 5$.

x	0	1	2	3	4	5	6	7
$2x \mod 5$	0	2	4	1	3	5	6	7

12.2.3 Modular Affine

Given a positive integer m where $2^{n-1} < m \le 2^n$ and integers a, b where $\gcd(a, m) = 1$ and $0 \le b < m$, define an n-bit permutation π by

$$\pi(x) = \begin{cases} ax + b \mod m & \text{if } x < m \\ x & \text{otherwise} \end{cases}$$

Such a permutation will be referred to as the permutation $ax + b \mod m$.

Example **12.2.3.** The 4-bit permutation $5x + 6 \mod 11$.

e	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$5x + 6 \mod 11$	6	0	5	10	4	9	3	8	2	7	1	11	12	13	14	15

12.2.4 Modular Inverse

Given a positive integer m where $2^{n-1} < m \le 2^n$, define a n-bit permutation π by

$$\pi(x) = \begin{cases} x^{-1} \mod m & \text{if } x < m, (x, m) = 1 \\ x & \text{otherwise} \end{cases}$$

Thus is x is invertible modulo m, $\pi(x) = x^{-1}$, otherwise x is fixed. Such a permutation will be referred to as $x^{-1} \mod m$.

Example **12.2.4.** The 3-bit permutation $x^{-1} \mod 7$.

x	0	1	2	3	4	5	6	7
$x^{-1} \mod 7$	0	1	4	5	2	3	6	7

12.2.5 Modular Exponentiation

Given a prime p where $2^{n-1} < p \leq 2^n$ and a primitive root a, define an n-bit permutation π by

$$\pi(x) = \begin{cases} a^x \mod p & \text{if } 0 < x < p \\ x & \text{otherwise} \end{cases}$$

Such a permutation will be referred to as $a^x \mod p$.

EXAMPLE **12.2.5.** The 3-bit permutation $2^x \mod 5$.

x	0	1	2	3	4	5	6	7
$2^x \mod 5$	0	2	4	3	1	5	6	7

12.2.6 Pseudo-Inverse over GF$(2)^n$

Let $f(x) \in \mathrm{GF}(x)$ be a primitive polynomial of degree n. Then $\mathrm{GF}(x)/f(x)$ is isomorphic to $\mathrm{GF}(2^n)$. Since f is primitive, powers of x generate all $2^n - 1$ nonzero elements of $\mathrm{GF}(2^n)$. The pseudo-inverse is the map $\alpha \mapsto \alpha^{2^n-2}$, which maps each nonzero element to its inverse and maps 0 to itself.

$$\pi(\alpha) = \alpha^{2^n-2}$$

Each element of the finite field will be expressed as a binary integer as follows:

$$\alpha = a_{n-1}x^{n-1} + \cdots + a_1 a + a_0 = (a_{n-1} \cdots a_1 a_0)_2.$$

Such a permutation will be referred to as $x^{-1} \mod f$.

EXAMPLE **12.2.6.** The 3-bit permutation $\alpha^{-1} \mod x^3 + x + 1$.

α	0	1	2	3	4	5	6	7
α^{-1}	0	1	5	7	6	3	4	2

12.3 Complexity Class $\mathsf{NLC}(i, j)$

The complexity trees used in this chapter were generated using the Toffoli gate as the nonlinear transition. In the 4-bit case, where the Toffoli gate can only generate even permutations, a triply controlled NOT or CCCNOT was allowed for the final nonlinear round instead. We will use the following definition in this chapter.

Definition **12.3.1.** The complexity class $\mathsf{NLC}(i, j) = \mathsf{WIDTH\text{-}DEPTH}_B(i, j)$, where the nonlinear basis B is a Toffoli gate and a C^{i-1}-NOT (A NOT gate with $i-1$ controls). Additionally, the C^{i-1}-NOT gate is only allowed in the final nonlinear round.

Thus $\mathsf{NLC}(i, j)$ includes all permutations that can be computed in width less than or equal to i, with at most j nonlinear rounds (as restricted in definition). The theory of the preceding chapters was used to compute the results in the following sections.

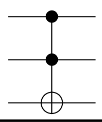
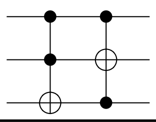
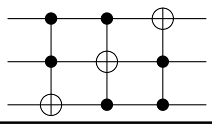
12.4 3-bit Permutations

Since all 2-bit permutations are affine functions, 3-bit permutations are the first nontrivial case to examine. 3-bit permutations are also unique due to the fact that the Tofolli gate is an odd permutation, and therefore generates all of S_{2^3} with $\mathrm{AGL}_3(2)$.

As established earlier, there are four affine equivalence classes of the $8! = 40320$ 3-bit permutations. Using the Toffoli gate as our nonlinear transition, we can generate a canonical representative for each of the three nonlinear classes

(Figure 12.1). We will consider which permutations fall into each complexity level.

Figure 12.1: 3-bit Permutations

$\mathsf{NLC}(3,1)$	$\mathsf{NLC}(3,2)$	$\mathsf{NLC}(3,3)$
		

12.4.1 $\mathsf{NLC}(3,0)$

The simplest 3-bit permutations are the affine permutations or those requiring zero rounds of nonlinearity. $\mathsf{NLC}(3,0)$ contains 1344 permutations. Table 12.1 provides some example permutations.

Table 12.1: $\mathsf{NLC}(3,0)$ Permutations

Permutation	Parameters
$ax \mod 7$	$a = 2, 4$
$5x \mod 8$	
$x^{-1} \mod 7$	

12.4.2 $\mathsf{NLC}(3,1)$

This is the simplest possible nonlinear complexity class, since all 2-bit permutations are affine. This class contains $7 * 1344 = 9408$ 3-bit permutations. Obviously this class includes the Toffoli and Fredkin gates, as well as the permutations in table 12.2.

Table 12.2: $\mathsf{NLC}(3,1)$ Permutations

Permutation	Parameters
$x+1 \mod 8$	
$ax \mod 7$	$a=3,5,6$
$ax \mod 8$	$a=3,7$
$x^{-1} \mod 5$	

12.4.3 $\mathsf{NLC}(3,2)$

$\mathsf{NLC}(3,2)$ is the only nonlinear class of even 3-bit permutations. It is also the largest class containing $14 * 1344 = 18816$ which is almost half of the 3-bit permutations. Example permutations are provided in table 12.3.

Table 12.3: $\mathsf{NLC}(3,2)$ Permutations

Permutation	Parameters
$x+1 \mod 5$	
$x+1 \mod 7$	
$4x \mod 5$	
$5x \mod 6$	
$2^x \mod 5$	
$a^x \mod 7$	a=3,5

12.4.4 $\mathsf{NLC}(3,3)$

$\mathsf{NLC}(3,3)$ contains the 3-bit permutations with maximal nonlinear complexity and has $8 * 1344 = 10752$ members. Example permutations are provided in table 12.4.

Table 12.4: $\mathsf{NLC}(3,3)$ Permutations

Permutation	Parameters
$x+1 \mod 6$	
$ax \mod 5$	$a = 2, 3$
$3^x \mod 5$	
$\alpha^{-1} \mod x^3+x+1$	

12.4.5 No 3-bit Permutations are One-Way

It should be noted that every 3-bit permutation class contains an involution. Therefore, by lemma 9.3.3, all 3-bit permutations are two-way permutations. Thus no 3-bit permutation can be a one-way permutation.

12.5 4-bit Permutations

4 bit permutations contain the first examples of permutations which are not two-way. These classes are of particular interest in the study of one-way permutations, since the inverse of a permutation must be in a different double coset if there is any chance of computing f faster than f^{-1}.

Table 12.5 shows how the invariants sort the 302 equivalence classes at each depth. The table indicates that the Multiple Rank Invariant corresponds loosely with the nonlinear depth. Also it is interesting to note that two-way permutations are represented at every complexity depth.

Table 12.6 illustrates some examples of various simple permutations in each of the seven nonlinear complexity classes for 4-bit permutations.

12.6 Open Problems

- There is one unique class of 4-bit permutations with depth 6. Is any permutation we are familiar with in this class?
- For 4-bit permutations, involutions occur at every nonlinear complexity depth. Does this happen in general?

- Does one of the permutation classes that is not two-way have an implementation with additional scratch space that has provably fewer rounds of nonlinearity than its inverse?

Table 12.5: Invariant Table for each 4-bit Permutations Depth

		Two-way		not Two-way	
Depth	MRI	Even	Odd	Even	Odd
$\mathsf{NLC}(4,0)$	(4,0,0)	1			
$\mathsf{NLC}(4,1)$	(3,1,0)	1			
	(3,0,1)		1		
$\mathsf{NLC}(4,2)$	(2,2,0)	2			
	(2,1,1)	1	2		
$\mathsf{NLC}(4,3)$	(2,0,2)	2	1		
	(1,3,0)	3			
	(1,2,1)	1	4	4	
	(1,1,2)	3	1	2	4
$\mathsf{NLC}(4,4)$	(2,0,2)	1			
	(1,1,2)	7	2		
	(1,0,3)	3	9		
	(0,3,1)		3	2	
	(0,2,2)	7	2	8	14
	(0,1,3)	3	12	14	26
$\mathsf{NLC}(4,5)$	(1,1,2)	1			
	(1,0,3)	1			
	(0,2,2)	5	1	2	
	(0,1,3)	11	23	28	10
	(0,0,4)	31	26	6	10
$\mathsf{NLC}(4,6)$	(0,0,4)	1			

Table 12.6: Depth of Various 4-bit Permutations

Depth	Permutation	Parameters
$\mathsf{NLC}(4,0)$	$ax \mod 15$	$a = 2, 4, 8$
	$9x \mod 16$	
$\mathsf{NLC}(4,1)$	$ax \mod 15$	$a = 7, 11, 13, 14$
	$ax \mod 16$	$a = 5, 13$
$\mathsf{NLC}(4,2)$	$x + 1 \mod 16$	
	$-x \mod 11$	
	$ax \mod 16$	$a = 3, 7, 11, 15$
$\mathsf{NLC}(4,3)$	$x^{-1} \mod 11$	
	$5x \mod 12$	
$\mathsf{NLC}(4,4)$	$x + 1 \mod n$	$n = 9, 15$
	$-x \mod 9$	
	$ax \mod 10$	$a = 3, 7, 9$
	$ax \mod 11$	$a = 2, 3, 4, 5, 6, 7, 8, 9$
	$ax \mod 12$	$a = 7, 11$
	$ax \mod 13$	$a = 5, 8$
	$ax \mod 14$	$a = 9, 11, 13$
	$a^x \mod 13$	$a = 6, 7$
$\mathsf{NLC}(4,5)$	$x + 1 \mod n$	$n = 10, 11, 12, 13, 14$
	$x^{-1} \mod 13$	
	$ax \mod 9$	$a = 2, 4, 5, 7$
	$ax \mod 13$	$a = 2, 3, 4, 6, 7, 9, 10, 11, 12$
	$ax \mod 14$	$a = 3, 5$
	$\alpha^{-1} \mod x^4 + x + 1$	
	$a^x \mod 11$	$a = 2, 6, 7, 8$
	$a^x \mod 13$	$a = 2, 11$
$\mathsf{NLC}(4,6)$		Nothing found

Chapter 13

Case Studies

13.1 Summary

Using the tools developed for classifying and identifying Boolean permutations, we now focus on the study of particular permutations as follows.

- Hiltgen's asymmetric permutation.
- Zech logarithms: $1 + \omega^e = \omega^{Z(e)}$ where $\omega \in \mathrm{GF}(2^n)$.
- Increment: $x + 1 \pmod{2^n}$.
- Multiplication: $(2^{n-1} + 1)x \pmod{2^n}$.
- Addition and Subtraction.

13.2 The Hiltgen function

In 1990, the first ever computationally asymmetric permutation for the gate complexity measure was found [HG92]. The permutation $f(x_1, x_2, x_3, x_4) =$

(y_1, y_2, y_3, y_4) is defined as follows:

$$\begin{aligned} y_1 &= x_1 \oplus x_3 \\ y_2 &= x_2 \oplus x_4(x_1 \oplus x_2) \\ y_3 &= x_3 \oplus x_4(x_1 \oplus x_2) \\ y_4 &= x_4. \end{aligned}$$

Likewise, the inverse permutation $f^{-1}(y_1, y_2, y_3, y_4) = (x_1, x_2, x_3, x_4)$ is defined as follows:

$$\begin{aligned} x_1 &= y_1 \oplus y_3 \oplus y_4(y_1 \oplus y_2 \oplus y_3) \\ x_2 &= y_2 \oplus y_4(y_1 \oplus y_2 \oplus y_3) \\ x_3 &= y_3 \oplus y_4(y_1 \oplus y_2 \oplus y_3) \\ x_4 &= y_4. \end{aligned}$$

By exhaustively enumerating all possible constructions of f and f^{-1}, Hiltgen discovered that f could be constructed using only 5 gates, while f^{-1} required at least 6 gates. Thus, the permutation f is computationally asymmetrical according to the gate complexity measure.

Examination of the Hiltgen permutation reveals that it is affine equivalent to the Toffoli gate. Thus the complexity difference between f and f^{-1} is solely due to the affine component. It is also interesting to note the Hiltgen permutation has very low nonlinear complexity (just one Toffoli gate). This was likely necessary to ensure that all possible constructions could be exhausted.

13.3 Zech Logarithm

In computations over finite fields, field multiplication can be optimized at the expense of field addition. The Zech logarithm is used to compute field addition when multiplication has been optimized using powers of a primitive root [Jun93]. Typically, the Zech logarithm values are precomputed and stored in a table.

Definition **13.3.1.** The **Zech logarithm** $Z(e)$ of ω^e is the discrete logarithm of $1+\omega^e$.

$$1+\omega^e = \omega^{Z(e)}$$

By definition, $Z(0)=\infty$ and $Z(\infty)=0$.

This can be used to compute the sum of two primitive root powers as follows:

$$\omega^a+\omega^b = \omega^a(1+\omega^{b-a}) = \omega^a\omega^{Z(b-a)} = \omega^{a+Z(b-a)}.$$

Since the discrete log ranges from 0 to 2^n-2, we will use the value 2^n-1 as a placeholder for infinity.

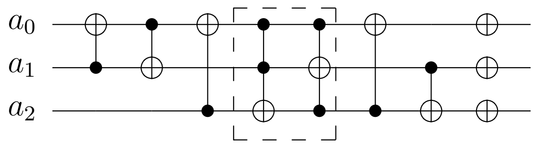
Example **13.3.2.** Let ω be a primitive root of the polynomial x^3+x+1. Then the Zech logarithm will define the following 3-bit permutation. Be aware that $7=\infty$ in the permutation.

e	0	1	2	3	4	5	6	7
$Z(e)$	7	3	6	1	5	4	2	0

Computation reveals that the permutation is affine equivalent to the nonlinear permutation of depth 2. Figure 13.1 illustrates a circuit for $Z(a_0+a_1\cdot 2+a_2\cdot 2^2)$.

Notice that the dashed box contains the only nonlinear part of the permutation.

Figure 13.1: Zech Logarithm for $x^3 + x + 1$



If we consider the other primitive degree 3 polynomial $x^3 + x^2 + 1$, then the Zech logarithm will now define the following different 3-bit permutation.

e	0	1	2	3	4	5	6	7
$Z(e)$	7	5	3	2	6	1	4	0

The Zech logarithm for $x^3 + x^2 + 1$ also has nonlinear depth 2. Thus, the two Zech logarithms are affine equivalent, since there is only one 3-bit equivalence class with nonlinear depth 2.

Example **13.3.3.** Consider now the Zech logarithm associated with the degree 4 primitive polynomial $x^4 + x + 1$. It has the following truth table ($15 = \infty$):

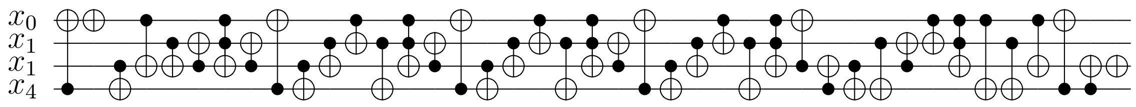
e	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$Z(e)$	15	4	8	14	1	10	13	9	2	7	5	12	11	6	3	0

Identification by computation reveals the Zech logarithm for $x^4 + x + 1$ has a nonlinear depth of 5. The Zech logarithm for $x^4 + x^3 + 1$ has the truth table:

e	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$Z(e)$	15	12	9	4	3	10	8	13	6	2	5	14	1	7	11	0

Computation then reveals that the two Zech logarithms are again affine equiv-

Figure 13.2: Zech Logarithm for $x^4 + x + 1$



alent. In fact,

$$Z_{x^4+x^3+1}(e) = \overline{Z_{x^4+x+1}(\overline{e})},$$

where $\bar{x}$ is the permutation which complements all variables. It is unknown if all the Zech logarithms associated with primitive polynomials are affine equivalent, but it is true for $n \leq 4$.

13.4 Incrementing modulo 2^n

One of the simplest nonlinear permutations we can examine is the simple act of adding 1. Let us first consider the 3-bit case.

x	0	1	2	3	4	5	6	7
$I(x)$	1	2	3	4	5	6	7	0

A computational check reveals that $I(x) = x + 1 \pmod 8$ requires 1 round of nonlinearity and is thus affine equivalent to the single Toffoli gate. Figure 13.3 illustrates a circuit for $I(x) = I(x_0 + 2 \cdot x_1 + 2^2 \cdot x_2)$.

Consider now the incrementation function on 4 bits, $I(x) = x+1 \pmod{16}$. A computational check reveals that $I(x)$ is an odd permutation requiring two rounds of nonlinearity. Thus it must use one Toffoli gate and one triple controlled NOT gate. Figure 13.4 illustrates a circuit for $I(x) = I(x_0 + 2 \cdot x_1 + 2^2 \cdot x_2 + 2^3 \cdot x^3)$.

Figure 13.3: Incrementing modulo 8

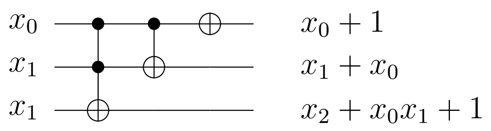
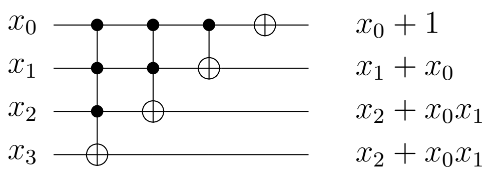


Figure 13.4: Incrementing modulo 16



13.5 Multiplication by $2^{n-1}+1$ modulo 2^n

In exploring affine functions modulo various moduli, it was discovered that the permutation $(2^{n-1}+1)x \pmod{2^n}$ always appeared to be linear. Upon further investigation, the following theorem was proved.

THEOREM **13.5.1.** $(2^{n-1}+1)x \pmod{2^n}$ *is a linear n-bit permutation. Furthermore, it is generated by adding the low bit to the high bit.*

Proof. We will consider the two cases where x is an even and odd number. First, suppose $x = 2k$ is even. Then the multiplication fixes $2k$.

$$(2^{n-1}+1)2k \equiv k2^n + 2k \equiv 2k \pmod{2^n}$$

Suppose now that $x = 2k+1$ is odd. Then

$$(2^{n-1}+1)(2k+1) \equiv k2^n + 2^{n-1} + 2k + 1 \equiv 2^{n-1} + 2k + 1 \pmod{2^n}.$$

Adding 2^{n-1} is the same as toggling the high bit of $2k+1$.

Since odd and even numbers can be distinguished by their low bit, we can combine both cases into the single case of using a single controlled NOT from the low bit to the high bit. Thus,

$$(2^{n-1}+1)x \equiv (2^{n-1}+1)(x_0+2x_1+\cdots+2^{n-1}x_{n-1}) \equiv (x_0+2x_1+\cdots+2^{n-1}(x_0 \oplus x_{n-1})),$$

and therefore $(2^{n-1}+1)x \pmod{2^n}$ is a linear function. □

13.6 Addition and Subtraction modulo 2^n

Addition and subtraction are two of the most common and most useful functions. Despite their apparent simplicity, a great deal of literature and research has shown many surprising ways to improve the computation of addition (i.e. carry-look ahead adders, carry-save adders, constant depth adders).

We will consider addition and subtraction using **Two's complement** which is the most common implementation on computers. Two's complement allows the same circuitry to add signed or unsigned numbers.

Example **13.6.1.** Suppose we were working with 8-bit numbers and wanted to find the two's complement of $40 = 00101000_2$.

00101000_2	Original value of 40
11010111_2	Complement
11011000_2	Add one

The final value 11011000_2 equals -40 in two's complement. Note that

$$00101000_2 + 11011000_2 = 00000000_2,$$

ignoring the overflow carry bit.

The following theorem proves that addition and subtraction are in the same affine equivalence class. Any improvement to one automatically translates into an improvement for the other. Thus addition and subtraction will always have the same nonlinear complexity.

THEOREM **13.6.2.** *For any positive integer n, 2's complement addition and subtraction modulo 2^n are affine equivalent.*

Proof. Recall that in 2's complement, $-a = \overline{a} + 1$, where $\overline{a}$ is a with every bit complemented. Equivalently, $\overline{a} = -a - 1$.

Consider the permutations $+, - : \mathrm{GF}(2)^{2n} \to \mathrm{GF}(2)^{2n}$ where for all $a, b \in \mathrm{GF}(2)^n$,

$$+(a,b) \mapsto (a, a+b \mod 2^n) \text{ and } -(a,b) \mapsto (a, a-b \mod 2^n).$$

Define $A_1, A_2 \in \mathrm{AGL}_n(2)$ where A_1 complements the first n bits and A_2 complements all $2n$ bits. Consider the action of $A_2 \circ (+) \circ A_1$.

$$\begin{aligned}
A_2 \circ + \circ A_1(a,b) &= A_2 \circ +(\overline{a}, b) \\
&= A_2(\overline{a}, \overline{a} + b) \\
&= A_2(\overline{a}, -a - 1 + b) \\
&= A_2(\overline{a}, \overline{a - b}) \\
&= (a, a - b) \\
&= -(a, b)
\end{aligned}$$

Since A_1 and A_2 are affine permutations, $+$ and $-$ are in the same double coset and are therefore affine equivalent. □

Chapter 14

Conclusion

The tools and theory developed in this thesis provide a new method for assessing the complexity of boolean permutations via nonlinearity. Given the universal nature of reversible computation, this implies the complexity of all classical computations can be measured according to their nonlinearity.

The fact that the amount of nonlinearity in a permutation is highly correlated with its complexity should not come as a surprise. Mathematicians are constantly developing the linear theory of a field and then gently extending out into the realm of nonlinearity.

It is hoped that the notion of nonlinearity as complexity will prove fruitful in both the study of complexity theory as well as in the study of optimal functions and algorithm theory.

Further advances in theory may make it possible to classify all 5-bit permutations. In addition, the computational methods for computing the optimal nonlinear complexity may lead to the discovery of a 4-bit or 5-bit permutation that is "mildly one-way", that is requiring one more round of nonlinearity for p^{-1} than for p.

Appendix A

Appendix A: Source Code

```
#!/usr/bin/env python

# _size4.pyx

import random as rand # Avoid conflict with stdlib.h

ctypedef unsigned long long ulong
ctypedef unsigned int uint

cdef extern from "Python.h":
    void* PyMem_Malloc(int)
    void  PyMem_Free(void* p)

cdef extern from "stdlib.h":
    int random()
    ctypedef long size_t
    void* malloc(size_t size)
    void free(void* ptr)

from size import rbin, bin

#############################################################
# BitStatic class
#############################################################

cdef class Size4:
    cdef public uint n, nn, N
    cdef public ulong one, m_identity, p_identity
    cdef public ulong m_mask, r_mask, c_mask, p_mask, p_base
    cdef ulong* x_mask
```

```
def __init__(self, uint n=4):
    cdef ulong i,j
    cdef ulong id, one

    one = 1
    self.n = n
    self.nn = n*n
    self.N = 1ull << n

    # Define the Identity matrix
    id = 0
    for i in range(n):
        id ^= (1ull << (n+1)*i)
    self.m_identity = id

    # Define the Identity permutation
    id = 0
    for i in range( 1 << n ):
        id = id ^ ((i*1ull) << (n*i))
    self.p_identity = id

    # The matrix mask is n*n bits set to 1111..1
    self.m_mask = (1ull << n*n) - 1
    # The row mask is the low n bits set to 111..1
    self.r_mask = (1ull << n) - 1
    # The column mask is n 1's spaced n bits apart
    id = 0
    for i in range(n):
        id ^= (1ull << (i*n))
    self.c_mask = id

    # The permutation mask is 2**n  1's
    self.p_mask = 2*((1ull << (self.N*self.n-1)) - 1) + 1
    self.p_base = self.p_mask // max(self.r_mask, 1)

    # x_mask[i] is the locations of monomials with no x_i.
    self.x_mask = <ulong*> malloc(sizeof(ulong)*self.n)
    for i in range(self.n):
        self.x_mask[i] = (self.p_mask // ((1 << (1<<(i+1)))-1) ) \
 * ((1 << (1<<(i)))-1)

def x_test(cls):
    for i in range(cls.n):
        print rbin(cls.x_mask[i],64)
```

```
def p_id(cls):
    '''
    Mysterious bug
    p_id() != p_identity
    '''
    return int(cls.p_identity)

def p_mk(cls):
    return int(cls.p_mask)

#################################################################
# Print methods
#################################################################
def str_v(cls, v):
    '''Binary form of permutation
    >>> print Size(3).str_v(0)
    000
    >>> print Size(3).str_v(6)
    011
    '''
    return rbin(v, cls.n)

def str_m(cls, m):
    '''Binary form of matrix
    >>> print Size(3).str_m(0421)
    100
    010
    001
    >>> print Size(3).str_m(0153)
    110
    101
    100
    '''
    s = ''
    for i in xrange(cls.n):
        t = (m >> cls.n*i) & cls.r_mask
        s += rbin(t, cls.n) + '\n'
    s = s[:-1]
    return s

def str_a(cls, a):
    '''Binary form of affine matrix
    >>> print Size(3).str_a(07421)
    100
    010
```

```
        001
        111
        >>> print Size(3).str_a(04153)
        110
        101
        100
        001
        '''
        s = ''
        for i in xrange(cls.n + 1):
            t = (a >> cls.n*i) & cls.r_mask
            s += rbin(t, cls.n) + '\n'
        s = s[:-1]
        return s

    def str_p(cls, p):
        '''Binary form of permutation
        >>> s = Size(3)
        >>> print s.str_p(s.p_identity)
        0 1 2 3 4 5 6 7
        >>> print s.str_p(067452301)
        1 0 3 2 5 4 7 6
        '''
        s = str(p & cls.r_mask)
        for i in xrange(1, cls.N):
            s += ' '+str((p >> (i*cls.n)) & cls.r_mask)
        return s

    def str_t(cls, p):
        '''Binary form of permutation
        >>> s = Size(3)
        >>> print s.str_t(s.p_identity),
        000
        001
        010
        011
        100
        101
        110
        111
        >>> print s.str_t(067452301),
        001
        000
        011
        010
        101
        100
        111
        110
        '''
        s = ''
        for i in xrange(cls.N):
            s += bin((p >> (i*cls.n)) & cls.r_mask, cls.n)+'\n'
        return s

    ################################################################
    # Conversion methods
    ################################################################
    '''
    NOTE: Finite Functions acting on [0,1,...,2**n-1]
    0 1 2
    3 4 5 ==> 8 7 6 5 4 3 2 1 0
    6 7 8

    Functions are named as follows
    _ReturnType_InputTypes_FunctionDescription
    Possible types are:
    m matrix
    v vector
    r row vector
    c column vector
    a affine
    b bit
    p permutation
    ? controlled affine (condition off of most significant bits)
    '''

    cpdef ulong r_c(cls, ulong c):
        '''Converts column vector to row vector
        >>> s = Size(3)
        >>> print s.str_v(s.r_c(s.c_mask))
        111
        '''
        c ^= c >> (2*cls.n-2)
        c ^= (c & (cls.r_mask << cls.n)) >> (cls.n - 1)
        c &= cls.r_mask
        return c

    cpdef ulong c_r(cls, ulong r):
        '''Converts row vector to column vector
        >>> s = Size(3)
        >>> print s.str_v(s.r_c(s.c_r(s.r_mask)))
        111
        '''
        r ^= (r >> 1) << cls.n
        r ^= (r >> 2) << 2*cls.n
        r &= cls.c_mask
        return r

    cpdef ulong p_v(cls, ulong v):
        '''Returns permutation equal to adding vector v.
        >>> s = Size(3)
        >>> print s.str_p(s.p_v(1))
        1 0 3 2 5 4 7 6
        '''
        return cls.p_identity ^ (v * cls.p_base)

    cpdef ulong p_m(cls, ulong m):
        '''Returns permutation equal to matrix multiplication by m.
        >>> print Size(3).str_p(Size(3).p_m(0153))
        0 3 5 6 1 2 4 7
        '''
        cdef uint i
        cdef ulong p
        p = 0
        for i in range(cls.N):
            p ^= cls.r_rm_mul(i, m) << (i * cls.n)
        return p

    cpdef ulong p_a(cls, ulong a):
        '''Permutation equal to affine matrix multiplication by a.
        >>> print Size(3).str_p(Size(3).p_a(04153))
        4 7 1 2 5 6 0 3
        '''
        cdef ulong p, i
        p = 0
        for i in range(cls.N):
            p ^= cls.r_ra_mul(i, a) << (i * cls.n)
        return p

    cpdef ulong a_v(cls, ulong v):
        '''Returns affine matrix equal to adding vector v.
        >>> print Size(3).str_a(Size(3).a_v(6))
        100
        010
        001
        011
        '''
        return cls.m_identity ^ (v << cls.nn)

    cpdef ulong m_v(cls, ulong v):
        '''Return matrix with char polynomial associated with v.
        >>> print Size(3).str_m(Size(3).m_v(3))
        010
        001
        110
        '''
        cdef ulong m
        m = cls.m_identity << 1
        m ^= ( v << (cls.n - 1) * cls.n ) & cls.m_mask
        return m

    def a_m(cls, m):
        return m

    # TODO: Tensor two permutations together


    #################################################################
    # Coercion methods
    #################################################################
    cpdef ulong v_p(cls, ulong p):
        '''Returns zero shift of permutation.
        Coerce - Inverts p_v
        >>> s = Size(3)
        >>> print  s.str_v(s.v_p(s.p_identity))
        000
        '''
        return p & cls.r_mask

    cpdef ulong m_p(cls, ulong p):
        '''Matrix function of permutation basis.
        Coerce - Inverts p_m
        >>> s = Size(3)
        >>> print  s.str_m(s.m_p(s.p_identity))
        100
        010
        001
        '''
        cdef uint i
```

```
        m = 0
        for i in range(cls.n):
            m ^= (p >> ( (1 << i) * cls.n) & cls.r_mask) \
<< (i * cls.n)
        return m

    cpdef ulong a_p(cls, ulong p):
        '''Returns affine function of permutation basis.
        Coerce - Inverts p_a
        >>> s = Size(3)
        >>> print  s.str_a(s.a_p(s.p_identity))
        100
        010
        001
        000
        >>> print  s.str_a(s.a_p(067452301))
        100
        010
        001
        100
        '''
        cdef uint i
        cdef ulong v, a
        a = 0
        v = p & cls.r_mask
        for i in range(cls.n):
            a ^= (v ^ (p >> ( (1 << i) * cls.n) & cls.r_mask)) \
<< (i * cls.n)
        return a ^ (v << cls.nn)

    cpdef ulong v_a(cls, ulong a):
        '''Returns affine shift.
        Coerce - Inverts a_v
        >>> s = Size(3)
        >>> print  s.str_v(s.v_a(07421))
        111
        '''
        return (a >> cls.nn) & cls.r_mask

    cpdef ulong m_a(cls, ulong a):
        '''Returns zero shift of permutation.
        Coerce - Inverts convert.a_m
        >>> s = Size(3)
        >>> print  s.str_m(s.m_a(07421))
        100
        010
        001
        '''
        return a & cls.m_mask

    #################################################################
    # Get and set methods
    #################################################################

    cpdef ulong r_pi_get(cls, ulong p, uint index):
        '''Returns the value p(i) from the permutation p.
        >>> s = Size(3)
        >>> print s.str_v(s.r_pi_get(s.p_identity, 4))
        001
        '''
        return (p >> cls.n*index) & cls.r_mask

    cpdef ulong b_vi_get(cls, ulong v, uint index):
        '''
        Candidate static method.
        '''
        return (v >> index) & 1

    cpdef ulong b_mij_get(cls, ulong m, uint i, uint j):
        '''Returns the (i,j) entry of the matrix m.
        >>> s = Size(3)
        >>> print s.b_mij_get(s.m_identity,0,0)
        1
        >>> print s.b_mij_get(s.m_identity,1,0)
        0
        '''
        return (m >> (i*cls.n +j)) & 1

    '''
    _b_ci_get
    _r_mi_get
    _c_mi_get
    _vib_set
    '''

    #################################################################
    # Matrix methods
    #################################################################

    cpdef ulong r_rm_mul(cls, ulong r, ulong m):
        '''Row vector times matrix
        >>> s = Size(3)
        >>> print s.str_v(s.r_rm_mul(1, 0421))
        100
        >>> print s.str_v(s.r_rm_mul(7, 0437))
        000
        >>> print s.str_v(s.r_rm_mul(7, 0467))
        101
        '''
        cdef ulong v
        v = cls.c_r(r)
        m &= v * cls.r_mask
        return cls.r_m_xor_cols(m)

    cpdef ulong r_ra_mul(cls, ulong r, ulong a):
        '''Row vector times affine matrix
        >>> s = Size(3)
        >>> print s.str_v(s.r_ra_mul(1, 03421))
        010
        >>> print s.str_v(s.r_ra_mul(7, 05437))
        101
        >>> print s.str_v(s.r_ra_mul(7, 01467))
        001
        '''
        cdef ulong v,m
        v = cls.c_r(r)
        m = a & (v * cls.r_mask)
        return cls.r_m_xor_cols(m) ^ (a >> cls.nn)

    cpdef ulong r_mr_mul(cls, ulong m, ulong r):
        '''Matrix vector times row
        >>> s = Size(3)
        >>> print s.str_v(s.r_mr_mul(0421, 1))
        100
        >>> print s.str_v(s.r_mr_mul(0665, 7))
        000
        >>> print s.str_v(s.r_mr_mul(0467, 7))
        101
        '''
        cdef ulong v
        m &= r * cls.c_mask
        v = cls.c_m_xor_rows(m)
        return cls.r_c(v)

    cpdef ulong r_ar_mul(cls, ulong a, ulong r):
```

```
    '''Affine matrix times row vecto
    First multiply by matrix and then add affine component
    >>> print Size(3).str_v(Size(3).r_ar_mul(04421, 1))
    101
    >>> print Size(3).str_v(Size(3).r_ar_mul(07566, 1))
    110
    '''
    cdef ulong m, v
    m = a & (r * cls.c_mask) & cls.m_mask
    v = cls.c_m_xor_rows(m)
    return cls.r_c(v) ^ (a >> cls.nn)

cpdef ulong m_mm_mul(cls, ulong m1, ulong m2):
    '''Matrix multiplication
    >>> s = Size(3)
    >>> print s.str_m(s.m_mm_mul(0421, 0421))
    100
    010
    001

    >>> print s.str_m(s.m_mm_mul(311, 311))
    101
    010
    001
    '''
    cdef uint i
    cdef ulong f,r,t
    t = cls.m_m_transpose(m2)
    r = cls.c_m_xor_rows(m1 & t)
    for i in range(1, cls.n):
        f = cls.m_mi_upshift_rows(m1, i)
        f = cls.c_m_xor_rows(f & t)
        f = cls.m_mi_upshift_rows(f, cls.n - i)
        r ^= f << (cls.n - i)
    r = cls.m_m_ishift_rows(r)
    return r

cpdef ulong a_aa_mul(cls, ulong a1, ulong a2):
    '''Affine Matrix multiplication
    >>> print Size(3).str_a(Size(3).a_aa_mul(0421, 0421))
    100
    010
    001
    000
    >>> print Size(3).str_a(Size(3).a_aa_mul(07467, 07467))
    101
    010
    001
    010
    '''
    cdef ulong m,v
    m = cls.m_mm_mul(a1 & cls.m_mask, a2 & cls.m_mask)
    v = cls.r_ra_mul((a1 >> cls.nn) & cls.r_mask, a2)
    return m ^ (v << cls.nn)


cpdef ulong m_m_inv(cls, ulong m):
    '''Assuming the matrix is invertible. Returns the inverse.
    >>> s = Size(3)
    >>> print s.str_m(s.m_m_inv(0421))
    100
    010
    001
    >>> print s.str_m(s.m_m_inv(143))
    010
    001
    111
    >>> print s.str_m(s.m_m_inv(0124))
    001
    010
    100
    >>> print Size(2).str_m(Size(2).m_m_inv(6))
    01
    10
    '''
    cdef ulong m1,m2,sval,oval,val
    cdef uint i,j
    m1 = m
    m2 = cls.m_identity
    for i in range(cls.n):
        for j in range(i, cls.n): # pivot a 1 to (i,i) position
            if cls.b_mij_get(m1, j, i) == 1:
                break
        m1 = cls.m_mij_swap_rows(m1, i, j)
        m2 = cls.m_mij_swap_rows(m2, i, j)
        val = (m1 >> i) & cls.c_mask # which rows added
        val ^= 1ull << i*cls.n # zero out pivot position
        sval = val * ((m1 >> i*cls.n) & cls.r_mask)
        oval = val * ((m2 >> i*cls.n) & cls.r_mask)
        m1 ^= sval
        m2 ^= oval
    return m2

cpdef ulong a_a_inv(cls, ulong a):
    '''Affine inverse. Assume multiplication on right.
    >>> Size(3).a_a_inv(07567)
    1003
    '''
    cdef ulong m,v
    m = cls.m_m_inv(a & cls.m_mask)
    v = cls.r_rm_mul(a >> cls.nn, m)
    return m ^ (v << cls.nn)

cpdef ulong m_m_transpose(cls, ulong m):
    '''Transpose of Bit Matrix
    >>> Size(3).m_m_transpose(0421)
    273
    >>> Size(3).m_m_transpose(0153)
    143
    '''
    cdef ulong f
    cdef uint i
    f = 0
    for i in range(cls.n):
        t = cls.c_mask & (m >> i)
        f ^= cls.r_c(t) << (cls.n * i)
    return f

def m_random(cls):
    '''Returns a random invertible bit matrix
    >>> s = Size(3); m=s.m_random();
    >>> s.m_mm_mul(m, s.m_m_inv(m)) == s.identity
    True
    '''
    while True:
        n = rand.randrange(1<<cls.nn)
        if cls.is_inv_a(n):
            return n

def a_random(cls):
    '''Returns a random invertible affine bit matrix
    >>> s = Size(3); a=s.a_random();
    >>> s.a_aa_mul(a, s.a_a_inv(a)) == s.identity
    True
    '''
    while True:
        n = rand.randrange(1<<cls.nn)
        if cls.is_inv_a(n):
            return n ^ (rand.randrange(cls.N) << cls.nn)


################################################################
# Generator methods
################################################################

"""
def density2(cls):
    '''Generate all weight 2 words.
    >>> for i in Size(4).density2(): print Size(4).str_v(i)
    1100
    1010
    1001
    0110
    0101
    0011
    '''
    m = 3
    for i in xrange(cls.n-1):
        current = (m << i)
        yield current
        for j in xrange(i+1, cls.n-1):
            current ^= m << j
            yield current
    return

def density3(cls):
    '''Generate all weight 2 words.
    >>> for i in Size(5).density3(): print Size(5).str_v(i)
    11100
    11010
    11001
    10110
    10101
    10011
    01110
    01101
    01011
    00111
    '''
    m = 7
    for i in xrange(cls.n-2):
        last = (m << i)
        current = last
        yield current
        for j in xrange(i+1, cls.n-2):
            for k in xrange(j+1, cls.n-1):
                current ^= 3 << k
                yield current
            last ^= 5 << j
            current = last
            yield current
    return
"""

#################################################################
# Helper methods
#################################################################

cpdef ulong r_m_xor_cols(cls, ulong m):
    '''XOR the values in each column to form a row vector.
    >>> s = Size(3)
    >>> s.r_m_xor_cols(s.m_identity) == s.r_mask
    True
    '''
    cdef uint i
    for i in range(1, cls.n):
        m ^= (m >> (cls.n * i) ) & cls.r_mask
    m &= cls.r_mask
    return m

cpdef ulong c_m_xor_rows(cls, ulong m):
    '''XOR the values in each row to form a column vector.
    >>> s = Size(3)
    >>> s.c_m_xor_rows(s.m_identity) == s.c_mask
    True
    '''
    cdef uint i
    for i in range(1, cls.n):
        m ^= (m >> i) & cls.c_mask
    m &= cls.c_mask
    return m

cpdef ulong m_mij_swap_rows(cls, ulong m, uint i, uint j):
    '''
    >>> Size(3).m_mij_swap_rows(0421, 0, 2)
```

```
        84
        '''
        cdef ulong val
        val = (m >> cls.n*i) ^ (m >> cls.n*j)
        val &= cls.r_mask
        m ^= val << cls.n*i
        m ^= val << cls.n*j
        return m

    cpdef ulong m_mi_upshift_rows(cls, ulong m, int i):
        ''' Circular shift of rows upward
        >>> Size(3).m_mi_upshift_rows(0421, 1)
        98
        '''
        return (m>>cls.n*i)^((m<<(cls.n*(cls.n-i)))&cls.m_mask)


    cpdef ulong m_m_ishift_rows(cls, ulong m):
        ''' Circular shift ith row by i
        >>> Size(3).m_m_ishift_rows(0421)
        161
        '''
        cdef uint i
        cdef ulong f
        f = m & cls.r_mask
        for i in range(1, cls.n):
            val = (m >> cls.n*i) & cls.r_mask
            val = cls.r_r_circ_shift(val, i)
            f ^= val << cls.n*i
        return f

    cpdef ulong r_r_circ_shift(cls, ulong r, uint i):
        return ((r>>(cls.n-i)&cls.r_mask)^(r<<(i)))&cls.r_mask

    cpdef ulong m_rr_outer_product(cls, ulong r1, ulong r2):
        ''' Outer product
        >>> s = Size(3)
        >>> print s.str_m(s.m_rr_outer_product(05, 07))
        111
        000
        111
        '''
        r1 = cls.c_r(r1)
        return r1 * r2
```

```
'''
m_m_canonical form
b_vv_dotproduct
'''


#################################################################
# Permutation methods
#################################################################

cpdef ulong p_p_inv(cls, ulong p):
    '''Assuming the function is a permutation. Returns inverse.
    >>> print Size(3).str_p(Size(3).p_p_inv(024710536))
    3 4 7 1 6 2 0 5
    '''
    cdef ulong i
    cdef ulong f
    f = 0
    for i in range(cls.N):
        f ^= i << ( ( (p >> i*cls.n) & cls.r_mask ) * cls.n )
    return f

cpdef ulong p_p_reverse(cls, ulong p):
    '''Reverses permuation as a list
    >>> print Size(3).str_p(Size(3).p_p_reverse(024710536))
    2 4 7 1 0 5 3 6
    '''
    cdef uint i
    cdef ulong f
    f = 0
    for i in range(cls.N):
        f ^= ((p>>i*cls.n)&cls.r_mask)<<(cls.N-i-1)*cls.n
    return f

cpdef ulong p_pp_mul(cls, ulong p0, ulong p1):
    '''Composition function.
    >>> print Size(3).str_p(Size(3).p_pp_mul(024710536,024710536))
    4 0 7 6 3 2 1 5
    >>> print Size(3).str_p(Size(3).p_pp_mul(024710536,053610742))
    3 0 6 2 4 5 1 7
    '''
    cdef uint i
    cdef ulong f, p0i, p1j
    f = 0
    for i in range(cls.N):
        p0i = (p0 >> i*cls.n) & cls.r_mask
        p1j = (p1 >> p0i*cls.n) & cls.r_mask
        f ^= p1j << cls.n*i
    return f

cpdef ulong p_pa_mul(cls, ulong p, ulong a):
    '''Composition function.
    >>> s = Size(3)
    >>> print s.str_p(s.p_pa_mul(053610742,07124))
    5 6 0 7 3 4 1 2
    '''
    cdef uint i
    cdef ulong f, pi
    f = 0
    for i in range(cls.N):
        pi = (p >> i*cls.n) & cls.r_mask
        f ^= cls.r_ra_mul(pi, a) << cls.n*i
    return f

cpdef ulong p_ap_mul(cls, ulong a, ulong p):
    '''Composition function. (Slower than reversed composition)
    >>> s = Size(3)
    >>> print s.str_p(s.p_ap_mul(07124,053610742))
    5 0 6 4 3 7 1 2
    '''
    pa = cls.p_a(a)
    return cls.p_pp_mul(pa,p)

cpdef ulong p_p_fixBasis(cls, ulong p):
    cdef ulong a
    while not cls.is_inv_a(cls.a_p(p)):
        p = cls.p_ap_mul(cls.a_random(), p)
    a = cls.a_a_inv(cls.a_p(p))
    p = cls.p_pa_mul(p,a)
    return p

cpdef uint is_a_pp_mul(cls, ulong p, ulong q):
    cdef ulong v000,v001,v010,v100,v101,v110,v111
    cdef ulong t
    v000 = (q >> (cls.n * (p&cls.r_mask)))
    v001 = (q >> (cls.n * ((p >> cls.n) & cls.r_mask)))
    v010 = (q >> (cls.n * ((p >> 2*cls.n) & cls.r_mask)))
    v011 = (q >> (cls.n * ((p >> 3*cls.n) & cls.r_mask)))
    if ((v000 ^ v001 ^ v010 ^ v011) & cls.r_mask) != 0:
        return 0
    v100 = (q >> (cls.n * ((p >> 4*cls.n) & cls.r_mask)))
    v101 = (q >> (cls.n * ((p >> 5*cls.n) & cls.r_mask)))
    if ((v000 ^ v001 ^ v100 ^ v101) & cls.r_mask) != 0:
        return 0
    v110 = (q >> (cls.n * ((p >> 6*cls.n) & cls.r_mask)))
    if ((v000 ^ v010 ^ v100 ^ v110) & cls.r_mask) != 0:
        return 0
    v111 = (q >> (cls.n * ((p >> 7*cls.n) & cls.r_mask)))
    if ((v000 ^ v001 ^ v110 ^ v111) & cls.r_mask) != 0:
        return 0
    t = cls.p_pp_mul(p,q)
    if cls.is_a_p(t) == 1: # Just test the darn thing then!
        return t
    else:
        return 0


cpdef uint is_inv_a(cls, ulong a):
    '''Determines whether affine transformation is invertible.
    >>> Size(3).is_inv_a(02743)
    False
    >>> Size(3).is_inv_a(07764)
    True
    '''
    ai = cls.a_a_inv(a)
    return cls.a_aa_mul(a,ai) == cls.m_identity

cpdef int cmp_p(cls, ulong p0, ulong p1):
    '''Compare using lexicographic value
    Examples:
    >>> Size(2).cmp_p(0xe4,0xd9)
    True
    '''
    cdef uint i
    cdef ulong val0, val1
    for i in range(cls.N):
        val0 = cls.r_pi_get(p0, i)
        val1 = cls.r_pi_get(p1, i)
        if val0 == val1:
            continue
        else:
            return val0 < val1
    return 0

cpdef ulong p_random_s(cls):
    cdef ulong p
    cdef ulong i,j
    p = cls.p_identity
    for i in range(cls.N):
        j = rand.randrange(i,cls.N)
        p = cls.p_pij_swap(p,i,j)
    return p

cpdef ulong p_random(cls):
    cdef ulong p
    cdef ulong i,j
    p = cls.p_identity
    for i in range(cls.N):
        j = (random() % (cls.N - i)) + i
        p = cls.p_pij_swap(p,i,j)
    return p

cpdef ulong p_pij_swap(cls, ulong p, uint i, uint j):
    cdef ulong val
    val = cls.r_pi_get(p,i) ^ cls.r_pi_get(p,j)
    p ^= val << cls.n*i
    p ^= val << cls.n*j
    return p

cpdef ulong p_p_next(cls, ulong p):
    cdef uint i,j
    cdef ulong pi,pj

    i = cls.N - 1
    pj = cls.r_pi_get(p,i)

    while True:
        i = i -1
        if i < 0:
            return 0
        pi = cls.r_pi_get(p, i)
        if pi < pj:
            break
        pj = pi

    j = cls.N
    while cls.r_pi_get(p, j-1) <= pi:
        j = j - 1

    p = cls.p_pij_swap(p,i,j-1)
    i = i + 1
    j = cls.N -1

    while i<j:
        p = cls.p_pij_swap(p,i,j)
        i += 1
        j -= 1
    return p

cpdef ulong p_p_not(cls, ulong p, ulong target):
    '''NOT Gate
    >>> s = Size(3); p = s.p_identity
    >>> print s.str_p(s.p_p_cnot(p,0))
    1 0 3 2 5 4 7 6
    '''
    return p ^ (cls.p_base << target)

cpdef ulong p_p_cnot(cls, ulong p, ulong source, ulong target):
    '''Control NOT
    >>> s = Size(3); p = s.p_identity
    >>> print s.str_p(s.p_p_cnot(p,0,1))
    0 3 2 1 4 7 6 5
    '''
    cdef ulong d
    d = (p & (cls.p_base << source)) >> source
    return p ^ (d << target)

cpdef ulong p_p_ccnot(cls, ulong p, ulong source1, \
                      ulong source2, ulong target):
    '''Double control NOT
    >>> s = Size(3); p = s.p_identity
    >>> print s.str_p(s.p_p_ccnot(p,0,1,2))
    0 3 2 1 4 7 6 5
    '''
    cdef ulong d
    d = (p & (cls.p_base << source1)) >> source1
    d &= (p & (cls.p_base << source2)) >> source2
    return p ^ (d << target)

cpdef ulong p_p_cccnot(cls, ulong p, ulong source1, \
             ulong source2, ulong source3, ulong target):
    '''Triple control NOT
    >>> s = Size(4); p = s.p_identity
    >>> print s.str_p(s.p_p_cccnot(p,0,1,2,3))
    '''
    cdef ulong d
    d = (p & (cls.p_base << source1)) >> source1
    d &= (p & (cls.p_base << source2)) >> source2
    d &= (p & (cls.p_base << source3)) >> source3
    return p ^ (d << target)

def Test3(cls, p):
    L = []
    L.append(cls.p_identity)
    L.append(cls.p_p_ccnot(L[0],0,1,2))
    L.append(cls.p_p_ccnot(L[1],0,2,1))
    L.append(cls.p_p_ccnot(L[2],1,2,0))

    for i in range(4):
        if cls.DC(p,L[i]):
            return i
    return 'Fail'

def Test4(cls,p):
    if cls.b_p_parity(p) == 0:
        return 'Even: '+str(cls.Test4Even(p))
    else:
        return 'Odd:  '+str(cls.Test4Odd(p))

def order(cls, p):
    '''Find order of permutation
    >>> s=Size(3); s.order(s.p_identity) == 1
    True
    >>> s.order(057136420)
    7
    '''
    d = p
    i = 1
    while True:
        if d == cls.p_identity:
            return i
        d = cls.p_pp_mul(d,p)
        i += 1
        if i > 999:
            return None

#################################################################
# Rank Algorithms
#################################################################
```

```
cpdef uint b_p_parity(cls, ulong p):
    '''Return the parity of the permutation.
    >>> s = Size(4); p = s.p_identity
    >>> s.b_p_parity(p)
    0
    >>> p = s.p_random(); s.b_p_parity(s.p_pp_mul(p,p))
    0
    >>> print Size(3).b_p_parity(067543210) # Toffoli Gate
    1
    '''
    cdef uint a,c,i,j
    a = 0; c = 0
    for j in range(cls.N):
        if (a>>j)&1 == 0:
            c ^= 1
            i = j
            while True:
                a ^= 1<<i
                i = cls.r_pi_get(p,i)
                if i == j:
                    break
    return c # (cls.N-c)%2


cpdef uint n_p_lowbit(cls, ulong p):
    cdef uint n
    n = 0
    if (p & 0xffffffff) == 0:
        n += 32
        p = p >> 32
    if (p & 0xffff) == 0:
        n += 16
        p = p >> 16
    if (p & 0xff) == 0:
        n += 8
        p = p >> 8
    if (p & 0xf) == 0:
        n += 4
        p = p >> 4
    if (p & 0x3) == 0:
        n += 2
        p = p >> 2
    if (p & 0x1) == 0:
        n += 1
    return n
    cpdef uint is_a_p(cls, ulong p):
        '''Determines whether a permutation is an affine function.
        >>> print Size(3).is_a_p(Size(3).p_identity)
        True
        >>> print Size(3).is_a_p(067543210) # Toffoli Gate
        False
        >>> print Size(3).is_a_p(067452301) # NOT Gate on lsb
        True
        '''
        cdef uint i
        cdef ulong a,v
        a = 0
        v = p & cls.r_mask
        for i in range(cls.n):
            a ^= (v ^ (p >> ( (1 << i) * cls.n) & cls.r_mask)) \
<< (i * cls.n)
        a ^= (v << cls.nn)
        for i in range(cls.N):
            if cls.r_ra_mul(i,a)!=((p>>(i*cls.n))&cls.r_mask):
                return 0
        return 1

    cpdef uint is_a_p4(cls, ulong p):
        '''Determines whether a permutation is an affine function.
        >>> print Size(4).is_a_p(Size(4).p_identity)
        True
        >>> print Size(3).is_a_p(067543210) # Toffoli Gate
        False
        >>> print Size(3).is_a_p(067452301) # NOT Gate on lsb
        True
        '''
        p ^= ( p        & 0xf)*0x1111111111111111ull
        p ^= ((p >>  4) & 0xf)*0x1010101010101010ull
        p ^= ((p >>  8) & 0xf)*0x1100110011001100ull
        if (p & 0xffff) != 0:
            return 0
        p ^= ((p >> 16) & 0xf)*0x1111000011110000ull
        if (p & 0xffffffff) != 0:
            return 0
        p ^= ((p >> 32) & 0xf)*0x1111111100000000ull
        return (p == 0)

    cpdef uint is_t_p(cls, ulong p):
        '''Determines whether a permutation is a.e. to Toffoli.
        '''
        cdef uint i
        cdef ulong t,q
        i = cls.n_p_signature(p)
        if i == 11:
            return 1
        else:
            return 0

    cpdef uint is_t3_p(cls, ulong p):
        '''Determines whether a permutation is a.e. to Toffoli.
        '''
        cdef uint i
        cdef ulong t,q
        i = cls.n_p_signature(p)
        if i == 3:
            return 1
        else:
            return 0

    cpdef uint n_p_rank(cls, ulong p):
        '''Determines rank of the four vector truth tables.
        '''
        cdef uint rank, n, a, b
        rank = 0
        while p != 0:
            n = cls.n_p_lowbit(p)
            b = n & 0x3
            a = n - b
            p ^= ((p>>b)&0x1111111111111111ull)*((p>>a)&0xf)
            rank += 1
        return rank

    cpdef ulong p_p_reduce_affine(cls, ulong p):
        p ^= ( p        & 0xf)*0x1111111111111111ull
        p ^= ((p >>  4) & 0xf)*0x1010101010101010ull
        p ^= ((p >>  8) & 0xf)*0x1100110011001100ull
        p ^= ((p >> 16) & 0xf)*0x1111000011110000ull
        p ^= ((p >> 32) & 0xf)*0x1111111100000000ull
        return p

    cpdef ulong p_p_reduce_quadratic(cls, ulong p):
        p ^= ((p >>  3*4) & 0xf)*0x1000100010001000ull
        p ^= ((p >>  5*4) & 0xf)*0x1010000010100000ull
        p ^= ((p >>  6*4) & 0xf)*0x1100000011000000ull
        p ^= ((p >>  9*4) & 0xf)*0x1010101000000000ull
        p ^= ((p >> 10*4) & 0xf)*0x1100110000000000ull
        p ^= ((p >> 12*4) & 0xf)*0x1111000000000000ull
        return p

    cpdef ulong p_p_reduce_cubic(cls, ulong p):
        p ^= ((p >>  7*4) & 0xf)*0x1000000010000000ull
        p ^= ((p >> 11*4) & 0xf)*0x1000100000000000ull
        p ^= ((p >> 13*4) & 0xf)*0x1010000000000000ull
        p ^= ((p >> 14*4) & 0xf)*0x1100000000000000ull
        return p

    def rank_signature(cls, ulong p):
        cdef uint prev, next
        sig = [ cls.b_p_parity(p) ]
        prev = cls.n_p_rank(p)
        p = cls.p_p_reduce_affine(p)
        next = cls.n_p_rank(p)
        sig.append(prev - next)
        prev = next
        p = cls.p_p_reduce_quadratic(p)
        next = cls.n_p_rank(p)
        sig.append(prev - next)
        prev = next
        p = cls.p_p_reduce_cubic(p)
        next = cls.n_p_rank(p)
        sig.append(prev - next)
        prev = next
        return sig

    cpdef uint n_p_signature(cls, ulong p):
        cdef uint prev, next, sig
        p = cls.p_p_reduce_affine(p)
        sig = 4 - cls.n_p_rank(p)
        p = cls.p_p_reduce_quadratic(p)
        sig ^= (4 - sig - cls.n_p_rank(p)) << 3
        return sig

    cpdef uint n_p_signature2(cls, ulong p):
        cdef uint sig
        sig = cls.n_p_signature(p)
        sig ^= cls.DC(p,cls.p_p_inv(p)) << 6
        sig ^= cls.b_p_parity(p) << 7
        return sig
    def L_p_signature(cls, ulong p):
        '''Signature List (Linear, Quadratic, Parity,
CommutatorSize, involution, order3, order4)
        '''
        cdef uint sig
        sig = cls.n_p_signature(p)
        L = []
        L.append(sig&0x7)
        L.append(sig>>3)
        L.append(cls.b_p_parity(p))
        L.append(len(cls.S_p_affineCommutator(p)))
        L.append(cls.DC(p,cls.p_p_inv(p)))
        '''Order does not appear to be working
        L.append(cls.b_p_AE_Order3(p))
        L.append(cls.b_p_AE_Order4(p))
        L.append(cls.b_p_AE_Order5(p))
        L.append(cls.b_p_AE_Order6(p))
        L.append(cls.b_p_AE_Order7(p))
        L.append(cls.b_p_AE_Order8(p))
        L.append(cls.b_p_AE_Order9(p))
        L.append(cls.b_p_AE_Order16(p))
        '''
        return L


    ################################################################
    # Double Coset
    ################################################################

    cpdef uint equivDC(cls, ulong a, ulong b):
        cdef uint h
        cdef ulong t
        for h in range(1<<(cls.nn+cls.n)):
            if cls.is_inv_a(h):
                t = cls.p_pa_mul(a,h)
                t = cls.p_pp_mul(t,b)
                if cls.is_a_p_fastfail(t):
                    return 1
        return 0

    cpdef linear(cls):
        cdef ulong h
        H = set()
        for h in (1<<cls.nn):
            if cls.is_inv_a(h):
```

```
            H.add(h)
        return H


################################################################
# Build up a coset test
################################################################


cpdef uint DC(cls, ulong p, ulong q):
    cdef ulong* coset
    cdef ulong* qlist
    cdef ulong t
    cdef uint i,j
    coset = <ulong*> malloc(sizeof(ulong)*20160)
    qlist = <ulong*> malloc(sizeof(ulong)*16)
    p = cls.p_p_inv(p)
    coset[0] = p
    p = cls.p_p_4bit_linear(p, coset+1)
    for j in range(16):
        qlist[j] = cls.p_ap_mul(cls.a_v(j),q)
    for i in range(20160):
        for j in range(16):
            #if cls.is_a_pp_mul(coset[i],qlist[j]) != 0:
            if cls.is_a_p4(cls.p_pp_mul(coset[i],qlist[j]))!=0:
                free(coset)
                free(qlist)
                return 1
    free(coset)
    free(qlist)
    return 0

cpdef ulong a_pp_DC(cls, ulong p, ulong q):
    cdef ulong* coset
    cdef ulong* qlist
    cdef ulong t, a, pi
    cdef uint i,j
    coset = <ulong*> malloc(sizeof(ulong)*20160)
    qlist = <ulong*> malloc(sizeof(ulong)*16)

    pi = cls.p_p_inv(p) # pi = p^-1
    coset[0] = pi
    cls.p_p_4bit_linear(pi, coset+1)
    for j in range(16):
        qlist[j] = cls.p_ap_mul(cls.a_v(j),q)
    for i in range(20160):
        for j in range(16):
            if cls.is_a_p4(cls.p_pp_mul(coset[i],qlist[j]))!=0:
                t = cls.p_pp_mul(coset[i],qlist[j])
                a = cls.a_a_inv(cls.a_p(t)) << 32
                #a = cls.a_p(cls.p_p_inv(t)) << 32
                #a = cls.a_p(t) << 32
                t = cls.p_pp_mul(p,t)
                t = cls.p_pp_mul(t,cls.p_p_inv(q))
                free(coset)
                free(qlist)
                return a ^ cls.a_p(t)
    free(coset)
    free(qlist)
    return 0

cpdef uint DC_Toff(cls, ulong p, ulong q):
    ''' Tests if p and q affinely differ by a Toffoli.
    There exists a,b,c such that apbqc=T
    '''
    cdef ulong* coset
    cdef ulong* qlist
    cdef ulong t
    cdef uint i,j
    coset = <ulong*> malloc(sizeof(ulong)*20160)
    qlist = <ulong*> malloc(sizeof(ulong)*16)
    p = cls.p_p_inv(p)
    coset[0] = p
    p = cls.p_p_4bit_linear(p, coset+1)
    for j in range(16):
        qlist[j] = cls.p_ap_mul(cls.a_v(j),q)
    for i in range(20160):
        for j in range(16):
            if cls.is_t_p(cls.p_pp_mul(coset[i],qlist[j]))!=0:
                free(coset)
                free(qlist)
                return 1
    free(coset)
    free(qlist)
    return 0

cpdef uint DC_T3(cls, ulong p, ulong q):
    ''' Tests if p and q affinely differ by a CCCNOT.
    There exists a,b,c such that apbqc=T^3
    '''
```

```
        cdef ulong* coset
        cdef ulong* qlist
        cdef ulong t
        cdef uint i,j
        coset = <ulong*> malloc(sizeof(ulong)*20160)
        qlist = <ulong*> malloc(sizeof(ulong)*16)
        p = cls.p_p_inv(p)
        coset[0] = p
        p = cls.p_p_4bit_linear(p, coset+1)
        for j in range(16):
            qlist[j] = cls.p_ap_mul(cls.a_v(j),q)
        for i in range(20160):
            for j in range(16):
                if cls.is_t3_p(cls.p_pp_mul(coset[i],qlist[j]))!=0:
                    free(coset)
                    free(qlist)
                    return 1
        free(coset)
        free(qlist)
        return 0

    #cpdef uint b_p_AE_Involution(cls, ulong p):
    def b_p_AE_Involution(cls, p):
        ''' Tests if p is Affine Equivalent to an involution.
        There exists a,b such that apb=i where i^2=1.
        '''
        cdef ulong* coset
        cdef ulong* qlist
        cdef ulong t
        cdef uint i,j
        coset = <ulong*> malloc(sizeof(ulong)*20160)
        qlist = <ulong*> malloc(sizeof(ulong)*16)
        coset[0] = p
        cls.p_p_4bit_linear(p, coset+1)
        for j in range(16):
            qlist[j] = cls.p_ap_mul(cls.a_v(j),p)
        for i in range(20160):
            for j in range(16):
                t = cls.p_pp_mul(coset[i],qlist[j])
                if cls.is_a_p4(t) != 0:
                    t = cls.p_p_inv(t)
                    t = cls.p_pp_mul(p,t)
                    if cls.p_pp_mul(t,t) == cls.p_identity:
                        free(coset)
                        free(qlist)
                        return 1
        free(coset)
        free(qlist)
        return 0

    # Hamiltonian cycle
    cdef inline ulong a_aij_cnot(cls, ulong a, uint i, uint j):
        return a ^ (((a>>i) & cls.p_base) << j)

    cdef ulong p_p_01(cls, ulong p, ulong* coset):
        p ^= ((p >> 1) & cls.p_base); coset[0] = p # CNOT(1,0)
        p ^= ((p & cls.p_base) << 1); coset[1] = p # CNOT(0,1)
        p ^= ((p >> 1) & cls.p_base); coset[2] = p # CNOT(1,0)
        p ^= ((p & cls.p_base) << 1); coset[3] = p # CNOT(0,1)
        p ^= ((p >> 1) & cls.p_base); coset[4] = p # CNOT(1,0)
        return p

    cdef ulong p_p_10(cls, ulong p, ulong* coset):
        p ^= ((p & cls.p_base) << 1); coset[0] = p # CNOT(0,1)
        p ^= ((p >> 1) & cls.p_base); coset[1] = p # CNOT(1,0)
        p ^= ((p & cls.p_base) << 1); coset[2] = p # CNOT(0,1)
        p ^= ((p >> 1) & cls.p_base); coset[3] = p # CNOT(1,0)
        p ^= ((p & cls.p_base) << 1); coset[4] = p # CNOT(0,1)
        return p

    cdef ulong p_p_20(cls, ulong p, ulong* coset):
        cdef ulong mask
        mask = cls.p_base << 2
        p = cls.p_p_10(p,coset)
        p ^= ((p & mask) >> 1); coset[5] = p  # CNOT(2,1)
        p = cls.p_p_10(p,coset+6)
        p ^= ((p & mask) >> 2); coset[11] = p # CNOT(2,0)
        p = cls.p_p_10(p,coset+12)
        p ^= ((p & mask) >> 1); coset[17] = p # CNOT(2,1)
        p = cls.p_p_10(p,coset+18)
        return p

    cdef ulong p_p_21(cls, ulong p, ulong* coset):
        cdef ulong mask
        mask = cls.p_base << 2
        p = cls.p_p_01(p,coset)
        p ^= ((p & mask) >> 2); coset[5] = p  # CNOT(2,0)
        p = cls.p_p_01(p,coset+6)
        p ^= ((p & mask) >> 1); coset[11] = p # CNOT(2,1)
        p = cls.p_p_01(p,coset+12)
        p ^= ((p & mask) >> 2); coset[17] = p # CNOT(2,0)
        p = cls.p_p_01(p,coset+18)
        return p

    cdef ulong p_p_0212(cls, ulong p, ulong* coset):
        cdef uint i,j
        for i in range(7):
            j = ((i>>2)^i)&1 # produces pattern 0101101
            if j==0:
                p = cls.p_p_20(p,coset+i*24)
            else:
                p = cls.p_p_21(p,coset+i*24)
            if i<6:
                p = cls.a_aij_cnot(p,j,2); coset[(i+1)*24-1] = p
        return p

    cdef ulong p_p_30(cls, ulong p, ulong* coset):
        p = cls.p_p_0212(p,coset)
        p = cls.a_aij_cnot(p,3,2); coset[168-1] = p
        p = cls.p_p_0212(p,coset+168)
        p = cls.a_aij_cnot(p,3,1); coset[2*168-1] = p
        p = cls.p_p_0212(p,coset+2*168)
        p = cls.a_aij_cnot(p,3,2); coset[3*168-1] = p
        p = cls.p_p_0212(p,coset+3*168)
        p = cls.a_aij_cnot(p,3,0); coset[4*168-1] = p
        p = cls.p_p_0212(p,coset+4*168)
        p = cls.a_aij_cnot(p,3,2); coset[5*168-1] = p
        p = cls.p_p_0212(p,coset+5*168)
        p = cls.a_aij_cnot(p,3,1); coset[6*168-1] = p
        p = cls.p_p_0212(p,coset+6*168)
        p = cls.a_aij_cnot(p,3,2); coset[7*168-1] = p
        p = cls.p_p_0212(p,coset+7*168)
        return p

    cdef ulong p_p_31(cls, ulong p, ulong* coset):
        p = cls.p_p_0212(p,coset)
        p = cls.a_aij_cnot(p,3,2); coset[168-1] = p
        p = cls.p_p_0212(p,coset+168)
        p = cls.a_aij_cnot(p,3,0); coset[2*168-1] = p
        p = cls.p_p_0212(p,coset+2*168)
        p = cls.a_aij_cnot(p,3,2); coset[3*168-1] = p
        p = cls.p_p_0212(p,coset+3*168)
        p = cls.a_aij_cnot(p,3,1); coset[4*168-1] = p
        p = cls.p_p_0212(p,coset+4*168)
        p = cls.a_aij_cnot(p,3,2); coset[5*168-1] = p
```

```
        p = cls.p_p_0212(p,coset+5*168)
        p = cls.a_aij_cnot(p,3,0); coset[6*168-1] = p
        p = cls.p_p_0212(p,coset+6*168)
        p = cls.a_aij_cnot(p,3,2); coset[7*168-1] = p
        p = cls.p_p_0212(p,coset+7*168)
        return p

    cdef ulong p_p_4bit_linear(cls, ulong p, ulong* coset):
        p = cls.p_p_30(p,coset)
        p = cls.a_aij_cnot(p,0,3); coset[1*1344-1] = p
        p = cls.p_p_31(p,coset+1344)
        p = cls.a_aij_cnot(p,2,3); coset[2*1344-1] = p
        p = cls.p_p_31(p,coset+2*1344)
        p = cls.a_aij_cnot(p,0,3); coset[3*1344-1] = p
        p = cls.p_p_30(p,coset+3*1344)
        p = cls.a_aij_cnot(p,1,3); coset[4*1344-1] = p
        p = cls.p_p_31(p,coset+4*1344)
        p = cls.a_aij_cnot(p,2,3); coset[5*1344-1] = p
        p = cls.p_p_31(p,coset+5*1344)
        p = cls.a_aij_cnot(p,0,3); coset[6*1344-1] = p
        p = cls.p_p_30(p,coset+6*1344)
        p = cls.a_aij_cnot(p,1,3); coset[7*1344-1] = p
        p = cls.p_p_31(p,coset+7*1344)
        p = cls.a_aij_cnot(p,0,3); coset[8*1344-1] = p
        p = cls.p_p_31(p,coset+8*1344)
        p = cls.a_aij_cnot(p,2,3); coset[9*1344-1] = p
        p = cls.p_p_30(p,coset+9*1344)
        p = cls.a_aij_cnot(p,1,3); coset[10*1344-1] = p
        p = cls.p_p_31(p,coset+10*1344)
        p = cls.a_aij_cnot(p,0,3); coset[11*1344-1] = p
        p = cls.p_p_31(p,coset+11*1344)
        p = cls.a_aij_cnot(p,2,3); coset[12*1344-1] = p
        p = cls.p_p_30(p,coset+12*1344)
        p = cls.a_aij_cnot(p,0,3); coset[13*1344-1] = p
        p = cls.p_p_30(p,coset+13*1344)
        p = cls.a_aij_cnot(p,3,2) # One wasted step.
        p = cls.a_aij_cnot(p,2,3); coset[14*1344-1] = p
        p = cls.p_p_30(p,coset+14*1344)
        return p

    ################################################################
    # Set Manipulators
    ################################################################

    cdef ulong* A_S(cls, S):
        cdef ulong* A
        cdef uint i,n
        n = len(S)
        A = <ulong*> malloc(sizeof(ulong)*n)
        i = 0
        for x in S:
            A[i] = x
            i += 1
        return A

    cdef S_An(cls, ulong* A, uint n):
        cdef uint i
        S = set()
        for i in range(n):
            S.add(A[i])
        return S

    cdef ulong* A_An_inv(cls, ulong* A, uint n):
        cdef uint i
        for i in range(n):
            A[i] = cls.p_p_inv(A[i])
        return A

    cdef ulong* A_p_leftAffineCoset(cls, ulong p):
        cdef ulong* coset
        cdef uint i,j
        coset = <ulong*> malloc(sizeof(ulong)*20160*16)
        coset[0] = p
        p = cls.p_p_4bit_linear(p, coset+1)
        for i in range(16):
            for j in range(20160):
                coset[20160*i+j] = coset[j] ^ (cls.p_base * i)
        return coset

    def S_p_leftAffineCoset(cls, ulong p):
        cdef ulong* coset
        coset = cls.A_p_leftAffineCoset(p)
        return cls.S_An(coset,322560)

    cdef ulong* A_p_leftLinearCoset(cls, ulong p):
        cdef ulong* coset
        cdef uint i,j
        coset = <ulong*> malloc(sizeof(ulong)*20160)
        coset[0] = p
        p = cls.p_p_4bit_linear(p, coset+1)
        return coset

    def S_p_leftLinearCoset(cls, ulong p):
        cdef ulong* coset
        coset = cls.A_p_leftAffineCoset(p)
        return cls.S_An(coset,20160)

    cdef ulong* A_p_affineCommutator(cls, ulong p):
        cdef ulong* S
        cdef ulong* C
        cdef uint i, count
        S = <ulong*> malloc(sizeof(ulong)*20160*16)
        C = <ulong*> malloc(sizeof(ulong)*20160*16+1)
        S = cls.A_p_leftAffineCoset(p) # pH
        p = cls.p_p_inv(p)
        count = 1
        for i in range(322560):
            if cls.is_a_pp_mul(S[i],p) != 0: # pHp^-1
                C[count] = cls.p_pp_mul(S[i],p)
                count += 1
        C[0] = count - 1
        return C

    def S_p_affineCommutator(cls, ulong p):
        cdef ulong* S
        cdef uint n
        S = cls.A_p_affineCommutator(p)
        return cls.S_An(S+1,S[0])

    cdef ulong* A_p_linearCommutator(cls, ulong p):
        cdef ulong* S
        cdef ulong* C
        cdef uint i, count
        S = <ulong*> malloc(sizeof(ulong)*20160)
        C = <ulong*> malloc(sizeof(ulong)*20160+1)
        S = cls.A_p_leftLinearCoset(p) # pH
        p = cls.p_p_inv(p)
        count = 1
        for i in range(20160):
            if cls.is_a_pp_mul(S[i],p) != 0: # pHp^-1
                if cls.v_p(cls.p_pp_mul(S[i],p)) == 0:
                    C[count] = cls.p_pp_mul(S[i],p)
                    count += 1
        C[0] = count - 1
        return C
```

```
def S_p_linearCommutator(cls, ulong p):
    cdef ulong* S
    cdef uint n
    S = cls.A_p_linearCommutator(p)
    return cls.S_An(S+1,S[0])

def S_p_affineTransversal(cls, ulong p):
    cdef ulong t
    H = cls.S_p_affineCommutator(p)
    G = cls.S_p_leftAffineCoset(cls.p_identity)
    for h in H:
        G.remove(h)
    H.remove(cls.p_identity) # t*id will be removed by pop()
    T = set([ cls.p_identity ])
    while len(G) > 0:
        t = G.pop()
        for h in H:
            G.remove(cls.p_pp_mul(t,h))
        T.add(t)
    return T

def S_p_linearTransversalToff(cls):
    '''Affine shift commute through a Toffoli gate into linear.
    Thus, the transversal can be only linear functions.
    '''
    cdef ulong t,p
    p = 0xbedcfa9836547210 # Toffoli

    H = cls.S_p_linearCommutator(p)
    G = cls.S_p_leftLinearCoset(cls.p_identity)
    for h in H:
        G.remove(h)
    H.remove(cls.p_identity) # t*id will be removed by pop()
    T = set([ cls.p_identity ])
    while len(G) > 0:
        t = G.pop()
        for h in H:
            G.remove(cls.p_pp_mul(t,h))
        T.add(t)
    return T

def is_pS_DC(cls, ulong p, S):
    cdef ulong s
    cdef uint sig
    sig = cls.n_p_signature(p)
    for s in S:
        if cls.n_p_signature(s) != sig:
            continue
        if cls.DC(p,s) != 0:
            return 1
    return 0

def p_pS_DC(cls, ulong p, S):
    cdef ulong s
    cdef uint sig
    sig = cls.n_p_signature(p)
    for s in S:
        if cls.n_p_signature(s) != sig:
            continue
        if cls.DC(p,s) != 0:
            return s
    return 0

cdef ulong is_pAn_DC(cls, ulong p, ulong* A, uint n):
    cdef ulong* coset
    cdef ulong t
    cdef uint i,j,k
    coset = <ulong*> malloc(sizeof(ulong)*20160)
    p = cls.p_p_inv(p)
    coset[0] = p
    p = cls.p_p_4bit_linear(p, coset+1)
    for i in range(20160):
        for j in range(16):
            t = coset[i] ^ (cls.p_base * j)
            for k in range(n):
                if cls.is_a_pp_mul(t,A[k]) != 0:
                    free(coset)
                    return 1
    free(coset)
    return 0

'''
Is Basis Fixing
Is Normal (zero fixing)
Is involution
Cycle Index Polynomial
Conjugation
Random
Hilbert polynomial
```

```
    One and two variable
    '''

if __name__ == '__main__':
    import doctest, sys
    doctest.testmod(sys.modules[__name__])
```

BIBLIOGRAPHY

[AtSotCL51] H. H. Aiken and the Staff of the Computation Laboratory, *Synthesis of electronic computing and control circuits*, 1951, pp. 231–278.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust, *The magma algebra system i: the user language*, J. Symb. Comput. **24** (1997), no. 3-4, 235–265.

[Ben73] C. H. Bennett, *Logical reversibility of computation*, IBM J. Res. Develop. **17** (1973), 525–532. MR MR0449020 (56 #7325)

[CG96] Stephen J. Curran and Joseph A. Gallian, *Hamiltonian cycles and paths in cayley graphs and digraphs – a survey*, Discrete Mathematics **156** (1996), no. 1-3, 1 – 18.

[HG92] A. P. Hiltgen and J. Ganz, *On the existence of specific complexity – asymmetric permutations*, Technical Report, Signal and Information Proc. Lab., ETH-Zurich (1992).

[Hil93] Alain P. L. Hiltgen, *Constructions of feebly-one-way families of permutations*, Advances in cryptology—AUSCRYPT '92 (Gold Coast, 1992), Lecture Notes in Comput. Sci., vol. 718, Springer, Berlin, 1993, pp. 422–434. MR MR1292706 (96e:94014)

[Hol05] Derek F. Holt, *Handbook of computational group theory (discrete mathematics and its applications)*, Chapman & Hall/CRC, January 2005.

[Hou06] Xiang-dong Hou, *Affinity of permutations of* $\mathbb{F}_2^n$, Discrete Appl. Math. **154** (2006), no. 2, 313–325. MR MR2194404 (2006j:06024)

[Jac74] Nathan Jacobson, *Basic algebra I*, W. H. Freeman and Company, San Francisco, CA, USA, 1974, ISBN 0-7167-0453-6 (v. 1.).

[Jun93] Dieter Jungnickel, *Finite fields*, Bibliographisches Institut, Mannheim, 1993, Structure and arithmetics. MR MR1238714 (94g:11109)

[KMS07] Samuel A. Kutin, David Petrie Moulton, and Lawren M. Smithline, *Computation at a distance*, 2007.

[Lor64] C.S. Lorens, *Invertible boolean functions*, Electronic Computers, IEEE Transactions on **EC-13** (1964), no. 5, 529–541.

[LPS87] Martin W. Liebeck, Cheryl E. Praeger, and Jan Saxl, *A classification of the maximal subgroups of the finite alternating and symmetric groups*, J. Algebra **111** (1987), no. 2, 365–383. MR MR916173 (89b:20008)

[Mas96] James L. Massey, *The difficulty with difficulty*, Presented EUROCRYPT 1996, in Zaragoza, Spain, 1996.

[MDM07] D. Maslov, G. W. Dueck, and D. M. Miller, *Techniques for the synthesis of reversible toffoli networks*, ACM Trans. Des. Autom. Electron. Syst. **12** (2007), no. 4, 42.

[Moo52] E. F. Moore, *A table for four-relay two-terminal contact networks*, 1952.

[Sco87] W. R. Scott, *Group theory*, second ed., Dover Publications Inc., New York, 1987. MR MR896269 (88d:20001)

[Vol99] Heribert Vollmer, *Introduction to circuit complexity*, Texts in Theoretical Computer Science. An EATCS Series, Springer-Verlag, Berlin, 1999, A uniform approach. MR MR1704235 (2001b:68047)