Clemson University

TigerPrints

All Dissertations Dissertations

May 2020

Codes and Sequences for Information Retrieval and Stream Ciphers

Travis Alan Baumbaugh

Clemson University, tabaumbaugh@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Recommended Citation

Baumbaugh, Travis Alan, "Codes and Sequences for Information Retrieval and Stream Ciphers" (2020). *All Dissertations*. 2620.

https://tigerprints.clemson.edu/all_dissertations/2620

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

Codes and Sequences for Information Retrieval and Stream Ciphers

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematical Sciences

by
Travis Alan Baumbaugh
May 2020

Accepted by:
Dr. Felice Manganiello, Committee Chair
Dr. Neil Calkin
Dr. Shuhong Gao
Dr. Kevin James

Abstract

Given a self-similar structure in codes and de Bruijn sequences, recursive techniques may be used to analyze and construct them. Batch codes partition the indices of code words into m buckets, where recovery of t symbols is accomplished by accessing at most τ in each bucket. This finds use in the retrieval of information spread over several devices. We introduce the concept of optimal batch codes, showing that binary Hamming codes and first order Reed-Muller codes are optimal. Then we study batch properties of binary Reed-Muller codes which have order less than half their length.

Cartesian codes are defined by the evaluation of polynomials at a subset of points in $\mathbb{F}_q^\mu$. We partition $\mathbb{F}_q^\mu$ into buckets defined by the quotient with a subspace V. Several properties equivalent to $V \cap \langle e_i, e_j \rangle = \{0\} \, \forall \, i,j \in [\mu]$ are explored. With this framework, a code in $\mathbb{F}_q^{\mu-1}$ capable of reconstructing μ indices is expanded to one in $\mathbb{F}_q^\mu$ capable of reconstructing $\mu+1$ indices. Using a base case in $\mathbb{F}_q^3$, we are able to prove batch properties for codes in $\mathbb{F}_q^\mu$. We generalize this to Cartesian Codes with a limit on the degree ρ of the polynomials.

De Bruijn sequences are cyclic sequences of length q^n that contain every q-ary word of length n exactly once. The pseudorandom properties of such sequences make them useful for stream ciphers. Under a particular homomorphism, the preimages of a binary de Bruijn sequence form two cycles. We examine a method for identifying points where these sequences may be joined to make a de Bruijn sequence of order n. Using the recursive structure of this construction, we are able to calculate sums of subsequences in $\mathcal{O}\left(n^4 \log(n)\right)$ time, and the location of a word in $\mathcal{O}\left(n^5 \log(n)\right)$ time. Together, these functions allow us to check the validity of any potential toggle point, which provides a method for efficiently generating a recursive specification. Each successful step takes $\mathcal{O}\left(k^5 \log(k)\right)$, for k from 3 to n.

Dedication

In memory of my father, who always encouraged me to pursue my interests.

Acknowledgments

I am grateful to Clemson University for hosting the REUs at which some of this work was completed. The REUs and the funding for the remainder of the research were made possible by an NSF Research Training Group (RTG) grant (DMS #1547399) promoting Coding Theory, Cryptography, and Number Theory at Clemson.

I would also like to thank my advisor Felice Manganiello for providing guidance on the problems and pushing me to complete milestones. I would like to acknowledge my committee of Neil Calkin, Shuhong Gao, and Kevin James for their work in reviewing the material, especially Neil Calkin for providing context and insight for working with recursive sums.

My family has been very supportive along the way, especially my parents who encouraged my to pursue my academic career and my brother who was kind enough to look over this paper.

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

Dedication . . . . **iii**

Acknowledgments . . . . **iv**

1 Introduction **1**
- 1.1 Coding Theory 2
- 1.2 Information Retrieval 5
- 1.3 Stream Ciphers and Feedback Sequences 7

2 Batch Code Constructions **10**
- 2.1 Background 13
- 2.2 Preliminary Results 15
- 2.3 Hamming Codes 20
- 2.4 Reed-Muller Codes 23
- 2.5 Cartesian Codes 34
- 2.6 Conclusions 45

3 De Bruijn Sequences **46**
- 3.1 Introduction 46
- 3.2 Background and Notation 48
- 3.3 Preliminary Results 52
- 3.4 Indexing 63
- 3.5 Recursive Sums 68
- 3.6 Complexity 84
- 3.7 Conclusions 88

Appendices **90**
- A De Bruijn Seqeunce generator code 91
- B Toggle Checking Code 97

Bibliography **101**

Chapter 1

Introduction

In the processing of data in modern computers, there are numerous operations that we may wish to carry out. One such operation is the encoding of the data in such a way that it may be retrieved through various means. This may enable recovery in light of potential corruption or enable simultaneous access by multiple parties. These operations fall under the category of coding theory. Another modification of data we may wish to perform is encryption, wherein the goal is to obfuscate the data in such a way that it cannot be recovered except by another party with a matching key. These operations are part of the mathematical study of cryptography.

In both cases, it is often useful to consider the structure of the data and operations involved as being built from similar, smaller structures. In this way, algorithms may be defined recursively so that we can rigorously prove their validity for a large class of data without having to handle many individual cases. We shall see how this use of recursion applies in particular to the construction of batch codes for use in information retrieval, and in the generation of de Bruijn sequences for use in stream ciphers.

1.1 Coding Theory

The foundations for information theory were laid in 1948 by Claude Shannon [Shannon, 1948]. From this comes the field of coding theory, which is concerned with the use of codes for various purposes, including error correction, data storage, and cryptography. For a comprehensive introduction to coding theory, see [MacWilliams and Sloane, 1977]. Here, a code is defined as follows:

Definition 1.1. *A* code $\mathcal{C}$ *of length* n *over an alphabet* Σ *is a subset of* Σ^n*.*

Throughout this paper, we are concerned primarily with a subset of codes known as linear codes:

Definition 1.2. *A code* $\mathcal{C}$ *that is a* k*-dimensional linear subspace of* $\mathbb{F}_q^n$ *(where* $\mathbb{F}_q$ *is the finite field of order* q*), and which has minimum distance* d*, is referred to as an* $[n, k, d]$ linear code*.*

Unless otherwise specified, any code to which we refer can be assumed to be a linear code. To discuss and compare codes, it is useful to build up several tools.

Definition 1.3. *The* Hamming weight *of a codeword* $c \in \mathcal{C}$ *is*

$$w(c) = |\{i \in [n] \mid c_i \neq 0\}|\,.$$

The Hamming distance was introduced in [Hamming, 1950], and can be defined in terms of the weight as follows:

Definition 1.4. *For two codewords* $c, c' \in \mathcal{C}$*, the* Hamming distance $D(c, c')$ *between the two words is defined by*

$$D(c, c') = w(c - c') = |\{i \in [n] \mid c_i \neq c'_i\}|\,.$$

The Hamming distance between two codewords is the number of positions in which the two words differ. From this, we get the notion of the minimum distance of a code.

Definition 1.5. *The* minimum distance $d = d(\mathcal{C})$ *of a code* $\mathcal{C}$ *is*

$$d = \min_{c, c' \in \mathcal{C} \mid c \neq c'} D(c, c').$$

That is, it is the minimum Hamming distance between two distinct codewords of $\mathcal{C}$. An alternate characterization of the minimum distance d is in terms of the weight of code words:

Corollary 1.6. *The minimum distance d of a linear code $\mathcal{C}$ may be determined by*

$$d = \min_{c \in \mathcal{C} | c \neq \underline{0}} \{w(c)\},$$

where $\underline{0} = (0, 0, \ldots, 0) \in \mathbb{F}_q^n$ is the codeword of all 0s.

In addition to taking the difference between two codewords, we can look at another binary operation known as the dot product.

Definition 1.7. *For two codewords $u, v \in \mathcal{C}$, the* dot product *$u \cdot v$ is*

$$u \cdot v = uv^T = \sum_{i=1}^{n} u_i v_i.$$

This allows us to define an important code related to $\mathcal{C}$.

Definition 1.8. *The* dual code *of $\mathcal{C}$, denoted $\mathcal{C}^\perp$, is the code*

$$\mathcal{C}^\perp = \{x \in \mathbb{F}_q^n \mid x \cdot u = 0 \, \forall u \in \mathcal{C}\}.$$

For working with the field $\mathbb{F}_q^n$, we introduce the *canonical basis.*

Definition 1.9. *For any $i \in [n]$, let $e_i \in \mathbb{F}_q^n$ be the vector $e_i = (a_1, \ldots, a_n)$, where $a_i = 1$ and $a_j = 0$ for all $j \neq i$.*

That is, e_i is the vector with a 1 in position i and 0 everywhere else. We also introduce notation for subspaces of $\mathbb{F}_q^n$.

Definition 1.10. *For any set $U \subset \mathbb{F}_q^n$, let $\langle U \rangle$ denote the linear subspace of $\mathbb{F}_q^n$ generated by U. If $U = \{u_1, u_2, \ldots, u_k\}$, then we write $\langle u_1, \ldots, u_k \rangle$ for convenience.*

In the discussion of linear codes, we will often be interested in two types of matrices: the generator matrix and the parity check matrix of the code.

Definition 1.11. *The* generator matrix G *for an* $[n, k, d]$ *linear code* $\mathcal{C}$ *over a field* $\mathbb{F}_q$ *is a matrix* $G \in \mathbb{F}_q^{k \times n}$ *such that*

$$\mathcal{C} = \{xG \mid x \in \mathbb{F}_q^k\}.$$

In other words, the generator matrix G is such that the code $\mathcal{C}$ is the row space of G.

Definition 1.12. *A* parity check matrix H *for an* $[n, k, d]$ *linear code* $\mathcal{C}$ *is a full-rank matrix* $H \in \mathbb{F}_q^{(n-k) \times n}$ *such that* H *is a generator matrix for* $\mathcal{C}^\perp$.

This means that $Hc^T = \underline{0}^T \ \forall c \in \mathcal{C}$, where here $\underline{0} \in \mathbb{F}_q^{n-k}$.

Of particular use in this dissertation are several particular categories of codes. First are a class of codes known as cyclic codes.

Definition 1.13. *A* cyclic code *is a code* $\mathcal{C}$ *such that for every codeword* $c = (c_1, \ldots, c_n) \in \mathcal{C}$*, the codeword* $c' = (c_2, \ldots, c_n, c_1)$ *is also a codeword. That is,* $c' \in \mathcal{C}$.

In Chapter 2 Section 2.3, we examine properties of Hamming codes:

Definition 1.14. *For some* $s \geq 2$*, let* $H \in \mathbb{F}_2^{2^s - 1 \times s}$ *be a matrix whose columns are all of the nonzero vectors of* $\mathbb{F}_2^s$*. Let* $n = 2^s - 1$*. We use* H *as our parity check matrix and define the* binary Hamming code*:*

$$\mathcal{H}_s \coloneqq \{c \in \mathbb{F}_2^n \mid cH^T = 0\}$$

We also examine Reed-Muller codes, first introduced by [Muller, 1954] and decoded efficiently by [Reed, 1954]. These can be defined as evaluations of multivariate polynomials of limited degree at all points in a vector space. First, we need the notation for the polynomials used.

Definition 1.15. *Let* $\mathbb{F}_q[X_1, \ldots, X_\mu]^\rho$ *be the set of all multivariate polynomials over* $\mathbb{F}_q$ *of total degree at most* ρ.

Then, we may define Reed-Muller codes from these sets of polynomials.

Definition 1.16. *Let* $\mathbb{F}_q[X_1, \ldots, X_\mu]$ *be the ring of polynomials in* μ *variables with coefficients in*

$\mathbb{F}_q$ *and let* $\mathbb{F}_q^\mu = \{P_1, \ldots, P_n\}$ *(so* $n = q^\mu$*). The q-ary* Reed-Muller code, $\mathcal{RM}_q(\rho, \mu)$ *is defined as:*

$$\mathcal{RM}_q(\rho, \mu) \coloneqq \{(f(P_1), \ldots, f(P_n)) \mid f \in \mathbb{F}_q[X_1, \ldots, X_\mu]^\rho\}.$$

Finally, there is the class of Cartesian codes, defined in [López et al., 2014].

Definition 1.17. *Let* $\mathbb{F}_q$ *be an arbitrary field, and* $A_1, \ldots, A_\mu$ *be finite non-empty subsets of* $\mathbb{F}_q$*. We define* $X = A_1 \times \ldots \times A_\mu \subseteq \mathbb{F}_q^\mu$*. Take the polynomial ring* $S = \mathbb{F}_q[x_1, \ldots, x_\mu]$ *and define* $S_{\leq\rho}$ *to be the* $\mathbb{F}_q$ *vector space of all polynomials in* S *with degree at most* ρ*. If* $P_1, \ldots P_n$ *are the points of* X*, we define the map:*

$$\begin{aligned} ev_\rho : S_{\leq\rho} &\to \mathbb{F}_q^{|X|} \\ f &\mapsto (f(P_1), \ldots, f(P_n)). \end{aligned}$$

We define the affine Cartesian code of degree ρ*, denoted* $\mathcal{C}_X(\rho)$*, to be the image of* ev_ρ*.*

1.2 Information Retrieval

One particular application of coding theory is in designing systems for information retrieval. In this setting, the idea is to take the original information, encode it some way, and then spread the resulting data among several devices. This is done in such a way that various desired properties with respect to the retrieval of the information are met.

Introduced in [Fazeli et al., 2015], PIR codes apply coding theory to the problem of Private Information Retrieval, where the party retrieving data does not wish to reveal which piece of data they are recovering. Previously, this was done with replication [Chor et al., 1998]. However, this meant that the storage overhead (ratio of data) was at least 2 to guarantee information-theoretic security. By requiring only computational security (making the problem computationally intractable), lower ratios were possible [Kushilevitz and Ostrovsky, 1997]. Introducing coding as a method allows for ratios arbitrarily close to 1 while maintaining information theoretic security.

If instead we are concerned with multiple users being able to retrieve data simultaneously,

we may turn to batch codes, introduced in [Ishai et al., 2004]. They can be defined briefly as follows.

Definition 1.18. *An (n, k, t, m, τ) batch code over an alphabet Σ encodes a string $x \in \Sigma^k$ into an m-tuple of strings, called buckets, of total length n such that for each t-tuple of distinct indices, $i_1, \ldots, i_t \in [k]$, the entries $x_{i_1}, \ldots, x_{i_t}$ can be decoded by reading at most τ symbols from each bucket.*

We can view the buckets as servers and τ as a bandwidth limit on each server. This original definition applies to a single user reconstructing t bits of information. Removing the requirement that these indices are distinct naturally generalizes to the concept of multiset batch codes. Going further, if all indices to be recovered are the same, this corresponds to PIR codes. Other schemes dealing with multiple requests are addressed in [Ramakrishnan and Wootters, 2018]. In this research, the queries are considered to happen at the same time, while the asynchronous case is considered in [Riet et al., 2018]. Combinatorial batch codes are replication-based codes using various combinatorial objects that allow for efficient decoding procedures. They are introduced in [Paterson et al., 2009] and further studied in [Bujtás and Tuza, 2011], [Bhattacharya et al., 2012], and [Silberstein and Gál, 2016].

Properties relevant to batch codes include locality and availability. We define them as in [Dimakis et al., 2011]:

Definition 1.19. *A code has* locality *r if any entry may be recovered by reading a set of entries of size at most r.*

Definition 1.20. *A code with locality r has* availability *δ if there exist δ disjoint recovery sets of size at most r for each index.*

That is, a code having locality r and availability δ means we have the opportunity to reconstruct a particular bit of data in δ different ways using a set of size r. Some bounds on the size of locally repairable codes are examined in [Cadambe and Mazumdar, 2015]. These are built up to bounds on the batch properties given small values of t and r in [Thomas and Skachek, 2017]. Further connections between batch codes and locally repairable codes are given in [Skachek, 2018].

When we only consider reconstructing any given bit multiple times, this corresponds to the properties as a PIR code. In the case of the batch properties, however, we consider the scenario

in which some bits may differ. We study the properties of Hamming, Reed-Muller, and Cartesian codes as batch codes in Chapter 2.

1.3 Stream Ciphers and Feedback Sequences

If the goal is encrypting data to prevent third parties from reading it without permission rather than to encoding data to ensure recovery, we move from the field of coding theory into the field of cryptography. In this field, ciphers are used to turn *plaintext* messages into *ciphertext* using a *key* in such a way that it is computationally infeasible to determine the plaintext from the ciphertext (and possibly other information an eavesdropper might acquire) without the key. Of course, it must also be quite feasible to determine the plaintext from the ciphertext *with* the key.

One potential cipher which is easily understood is the one-time pad (OTP), first described in 1882 by Frank Miller [Miller, 1882]. The concept is that the plaintext is represented as a string of numbers (usually in binary), and a sequence of random numbers is added onto this string to get the ciphertext. On the other end, the message is decrypted by subtracting off the same sequence of random numbers. In [Shannon, 1949], Claude Shannon proved that this cipher achieves perfect secrecy, and is in fact the only type of cipher that can do so.

It is usually impractical for both parties to agree on a sequence of truly random numbers the length of the message, and so OTP is rarely used. An exception has been in war, when perfect secrecy may be desired despite the inconvenience. In World War II, for instance, the SIGSALY system used duplicate phonographic records with random noise to scramble and unscramble audio [Boone and Peterson, 2000].

This impracticality means that it is far easier to use pseudorandom numbers instead. If both communicating parties can agree on a way to generate a sequence of pseudorandom numbers, then this sequence may be used to mask the data in what is known as a *stream cipher*. However, this is not without risks, as the lack of true randomness means perfect secrecy is no longer guaranteed. Thus, long key streams without bias in the pseudorandom numbers are required, and the same key stream must not be reused.

We now introduce a more formal definition of a (synchronous) stream cipher.

Definition 1.21. *A* stream cipher *is a cipher in which a string of symbols $k = (k_1, k_2, \dots)$ from a group G is agreed upon in some way by the participants. A plaintext message $p = (p_1, \dots, p_m) \in G^m$ is encrypted as $c = \mathrm{Enc}_k(p)$, with $c = (c_1, \dots, c_m) \in G^m$ defined by*

$$c_i = p_i + k_i \; \forall \, 1 \leq i \leq m.$$

A ciphertext message c is then decrypted as $p' = \mathrm{Dec}_k(c)$, where $p' = (p'_1, \dots, p'_m)$ is defined by

$$p'_i = c_i - k_i = p_i + k_i - k_i = p_i \; \forall \, 1 \leq i \leq m,$$

thus recovering the original message p.

This definition does not specify how the key stream is generated or agreed upon, as there are various methods. Due to their simplicity, Linear Feedback Shift Registers (LFSRs) are often a component in generating such streams.

Definition 1.22. *A* Linear Feedback Shift Register *is a register of values $x = (x_1, x_2, \dots, x_n) \in \mathbb{F}_q^n$ for some field $\mathbb{F}_q$, combined with a linear feedback function $f(x_1, \dots, x_n)$, where at each time step t, $x^{(t)}$ is updated to $x(t+1)$, where*

$$x_i^{t+1} = \begin{cases} x_{i+1}^{(t)} & \text{if } 1 \leq i \leq n-1 \\ f(x_1^{(t)}, \dots, x_n^{(t)}) & \text{if } i = n+1 \end{cases}.$$

That is, there is some initial state $x^{(1)} = (x_1^{(1)}, \dots, x_n^{(1)})$, and at each time step, the symbols are all shifted left, with a new symbol added onto the end that is a linear function of the symbols at the previous time step. With this definition in place, we can introduce the concepts of an LFSR sequence.

Definition 1.23. *An* LFSR sequence *is a sequence $s = (s_1, s_2, \dots)$ defined by*

$$s_i = x_1^{(i)} \forall 1 \leq i \in \mathbb{N},$$

where $x^{(1)} \in \mathbb{F}_q^n$ is some initial state and the states $x^{(t)}$ for $t > 1$ are determined as above.

Thus, it is the sequence of elements that appear in the first position of the state throughout the time steps.

Due to the linear structure of the function f, the options are limited and there are attacks that analyze the resulting output to determine the function f and thus the rest of the sequence. One way to improve security is by dropping the requirement that f be a linear function. That is, we have the following definition.

Definition 1.24. *A* Non-linear Feedback Shift Register (NLFSR) *is a shift register as defined above, but where $f(x_1, \ldots, x_n)$ is any function. An* NLFSR sequence *is a sequence as defined above but where states $x^{(t)}$ for $t > 1$ are determined by an NLFSR as opposed to an LFSR.*

Note that while the name includes "Non-linear," by definition they may actually be linear: it is simply not *required* to be linear. This makes all LFSRs a subset of NLFSRs.

Because there are exactly q^n possible states, and each state proceeds to the next in a deterministic manner, the sequence generated by an NLFSR will be periodic, with length at most q^n. Sequences with this maximum possible length are also known as de Bruijn sequences. We can define these briefly as follows:

Definition 1.25. *A* de Bruijn sequence *of order n over an alphabet K of size q is a periodic sequence s of period q^n such that for every word $w = (w_1, \ldots, w_n) \in K^n$, there exists a unique t such that $1 \leq t \leq q^n$ and $(s_t, s_{t+1}, \ldots, s_{t+n-1}) = (w_1, \ldots, w_n) = w$.*

In other words, every word of length n appears exactly once in the sequence when only one period is considered. There are a variety of ways to generate sequences with these properties, and we prove the viability of one such construction in Chapter 3.

Chapter 2

Batch Code Constructions

Batch codes may be used in information retrieval when multiple users want to access potentially overlapping requests from a set of devices while achieving a balance between minimizing the load on each device and minimizing the number of devices used. We can view the buckets as servers and the symbols used from each bucket as the load on each server. In the original scenario, a single user is trying to reconstruct t bits of information. This definition naturally generalizes to the concept of multiset batch codes which have nearly the same definition, but where the indices chosen for reconstruction are not necessarily distinct.

The family of codes known as batch codes was introduced in [Ishai et al., 2004]. They were originally studied as a scheme for distributing data across multiple devices and minimizing the load on each device and total amount of storage consumed. In this chapter, we study $[n, k, t, m, \tau]$ batch codes, where n is the code length, k is the dimension of the code, t is the number of entries we wish to retrieve, m is the number of buckets, and τ is the maximum number of symbols used from each bucket for any reconstruction of t entries. We seek to minimize the number of devices in the system and the load on each device while maximizing the amount of reconstructed data. That is, we want to minimize $m\tau$ while maximizing t.

This corresponds to t users who each wish to reconstruct a single element, among which there may be duplicates. This is similar to private information retrieval (PIR) codes, which differ in

that t duplicates of the same element must be reconstructed. Other schemes dealing with multiple requests are addressed in [Ramakrishnan and Wootters, 2018]. For batch and PIR codes where the queries do not all necessarily occur at the same time, see [Riet et al., 2018]. Restricted recovery set sizes are considered in [Thomas and Skachek, 2017]. Another notable type of batch code defined in [Ishai et al., 2004] is a primitive multiset batch code where the number of buckets is $m = n$.

Much of the related research involves primitive multiset batch codes with a systematic generator matrix. In [Ishai et al., 2004], the authors give results for some multiset batch codes using subcube codes and Reed-Muller codes. They use a systematic generator matrix, which often allows for better parameters. Their goal was to maximize the efficiency of the code for a fixed number of queries t. The focus of research on batch codes then shifted to combinatorial batch codes. These were first introduced by [Paterson et al., 2009]. They are replication-based codes using various combinatorial objects that allow for efficient decoding procedures. We do not consider combinatorial batch codes but some relevant results can be found in [Paterson et al., 2009], [Bujtás and Tuza, 2011], [Bhattacharya et al., 2012], and [Silberstein and Gál, 2016].

In order to reduce wait time for multiple users, we may look at locally repairable codes with availability as noted in [Dimakis et al., 2011]. A locally repairable code, with locality r and availability δ, provides us the opportunity to reconstruct a particular bit of data using δ disjoint sets of size at most r [Skachek, 2018]. When we only need to reconstruct this one bit multiple times, this gives us properties of the code as a Private Information Retrieval (PIR) code. However, the research in this chapter covers the scenario in which some bits may differ.

The Hamming weights of Cartesian Codes are studied in [Beelen and Datta, 2018]. This is a generalization of work in [Heijnen and Pellikaan, 1998], and in a similar fashion, the work in Section 2.5 aims to expand the study of batch properties from Reed-Muller codes as studied in [Baumbaugh et al., 2018] to the broader class of Cartesian Codes. In the same manner, we begin by examining codes with $\tau = 1$. The even broader family of generalized affine Cartesian codes, specifically those with complementary duals, are studied in [López et al., 2019].

In Section 2.1, we formally introduce batch codes. We then introduce the concepts of locality and availability of a code and summarize some results from previous work on batch codes. This is followed in Section 2.2 by the introduction of the concept of optimal batch codes and some other

preliminary results for working with batch codes, including the batch properties of a code $\mathcal{C}$ given that $\mathcal{C}^{\perp}$ is of a $(u \mid u+v)$-code construction with determined batch properties.

After this background and preliminaries are the results of joint work with Diaz, Friesenhahn, and Vetter during an REU at Clemson, published as [Baumbaugh et al., 2018]. We studied the batch properties of binary Hamming codes and Reed-Muller codes. Section 2.3 focuses on batch properties of binary Hamming codes. We show that Hamming codes are optimal $(2^{s-1}, 2^s-1-s, 2, m, \tau)$ batch codes for $m, \tau \in \mathbb{N}$ such that $m\tau = 2^{s-1}$.

Section 2.4 provides batch properties of Reed-Muller codes. Specifically, Section 2.4.2 gives the locality and availability properties of first-order Reed-Muller codes over any finite field. We find that the locality of $\mathcal{RM}_q(1,\mu)$ is 2 when $q \neq 2$ and 3 when the $q = 2$. Furthermore, we also show that its availability is $\left\lfloor \frac{q^{\mu}-1}{2} \right\rfloor$ when $q \neq 2$, whereas when $q = 2$, the availability is $\frac{2^{\mu}-1}{3}$ if μ is even and at least $\frac{2^{\mu}-4}{4}$ otherwise. In Section 2.4.3, we show that binary first-order Reed-Muller codes are optimal batch codes for $t = 4$. We first look at the specific $\mathcal{RM}(1,4)$ case and achieve parameters $(16, 5, 4, m, \tau)$ such that $m\tau = 10$. We then prove a general result that any Reed-Muller code with $\rho = 1$ and $\mu \geq 4$ has batch properties $(2^{\mu}, \mu+1, 4, m, \tau)$ for any $m, \tau \in \mathbb{N}$ such that $m\tau = 10$.

We next generalize our study of Reed-Muller codes and look at properties of $\mathcal{RM}(\rho,\mu)$ for all values of ρ and conclude our study by presenting batch properties $(2^{\mu}, k, 4, m, \tau)$ such that $m\tau = 10 \cdot 2^{2\rho-2}$ for $\mathcal{RM}(\rho',\mu)$ where $\mu \in \{2\rho+2, 2\rho+3\}$ and $\rho' \leq \rho$.

Finally, in Section 2.5, which is joint work with REU students Colgate and Jackman, we study the batch properties of Cartesian codes. First, in Subsection 2.5.1, we define the possible recovery sets for a point in $\mathbb{F}_q^n$. In Subsection 2.5.2, we define buckets as cosets of a subset V of $\mathbb{F}_q^n$ and show that under several equivalent conditions, this allows for queries of size $t = n+1$. The specific case with $V = \langle(1,1,\ldots,1)\rangle$ is considered in Subsection 2.5.3. In the end, this is generalized to Cartesian codes with a restriction on the degree of the polynomials in Subsection 2.5.4.

2.1 Background

Batch codes were introduced in [Ishai et al., 2004]. The relationship between batch codes and locally repairable codes (as well as PIR codes) is studied in [Skachek, 2018]. Throughout this chapter, by "batch codes" we refer specifically to multiset batch codes, defined by [Ishai et al., 2004]. To build up to this definition, we first introduce several notions. First, we have some notation:

Definition 2.1. *For any $n \in \mathbb{N}$, the set $[n]$ is defined as $[n] = \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$. That is, $[n] = \{1, 2, \ldots, n\}$.*

Definition 2.2. *A* bucket configuration *$B_1, \ldots, B_m$ is a partition on index set I. For each $k \in [m]$, the B_k is referred to as a* bucket.

Definition 2.3. *For any index $i \in I$, a* recovery set R_i is a set such that, for any codeword $c \in \mathcal{C}$, the value of c_i may be recovered by reading the symbols $\{c_j \mid j \in R_i\}$.

At this point, we note two distinct categories of recovery sets. If $R_i = \{i\}$, then we refer to R_i as *direct access*. If instead $i \notin R_i$, then we refer to R_i as an *indirect* recovery set. We also note that while any set containing a recovery set is technically a recovery set, these shall not be considered proper recovery sets in the remainder of the paper. Now we deal with multiple recovery sets at the same time for a query of indices that are not necessarily distinct.

Definition 2.4. *Given a query $Q = (i_1, \ldots, i_t) \in I^t$, we say that a set of recovery sets $R^Q = \{R_{i_1}, \ldots, R_{i_t}\}$ is a* query recovery set *with property τ for Q if*

1. $\left| \left(\bigcup_{s=1}^{t} R_{i_s} \right) \cap B_k \right| \leq \tau \ \forall \ k \in [m]$, *and*
2. $R_{i_r} \cap R_{i_s} = \emptyset \ \forall \ r, s \in [t]$ *where $r \neq s$.*

Definition 2.5. *We say that a bucket configuration $B_1, \ldots, B_m$ is t, τ* valid *if, for all queries $Q = (x_1, \ldots, x_t) \in I^t$, there exists a query recovery set R^Q with property τ.*

Now, with the building blocks in place, we may more rigorously define batch codes.

Definition 2.6. *A $[n, k, t, m, \tau]$ linear* batch code *$\mathcal{C}$ over $\mathbb{F}_q$ is a linear code $\mathcal{C}$ of length n and dimension k, together with a t, τ valid bucket configuration $B_1, \ldots, B_m$.*

In [Thomas and Skachek, 2017], recovery sets with restricted sizes are considered. These are similar to codes with locality and availability, which can be defined in our language as follows:

Definition 2.7. *For any $r \geq 1$, a code $\mathcal{C}$ has* locality r *if, for every $i \in I$, there exists an indirect recovery set R_i such that $|R_i| \leq r$. The smallest r for which this holds is the* minimal locality *of $\mathcal{C}$.*

Definition 2.8. *A code $\mathcal{C}$ with locality $r \geq 1$ has* availability $\delta \geq 1$ *if, for every $i \in I$, there exist δ pairwise-disjoint indirect recovery sets $R_{i,1}, \ldots, R_{i,\delta}$, where $|R_{i,k}| \leq r$ for each $k \in [\delta]$.*

Next, the focus of research turned to linear batch codes, which use classical error-correcting codes. The following general results are proven in [Lipmaa and Skachek, 2015]:

Theorem 2.9. *Let $\mathcal{C}$ be an $[n, k, t, n, 1]$ linear batch code over $\mathbb{F}_2$ with generator matrix G. Then, G is a generator matrix of the classical error-correcting $[n, k, d]_2$ linear code where $d \geq t$.*

Theorem 2.10. *Let $\mathcal{C}_1$ be an $[n_1, k, t_1, n_1, 1]_q$ linear batch code and $\mathcal{C}_2$ be an $[n_2, k, t_2, n_2, 1]_q$ linear batch code. Then, there exists an $[n_1 + n_2, k, t_1 + t_2, n_1 + n_2, 1]_q$ linear batch code.*

Theorem 2.11. *Let $\mathcal{C}_1$ be an $[n_1, k_1, t_1, n_1, 1]_q$ linear batch code and $\mathcal{C}_2$ be an $[n_2, k_2, t_2, n_2, 1]_q$ linear batch code. Then, there exists an $[n_1 + n_2, k_1 + k_2, \min(t_1, t_2), n_1 + n_2, 1]_q$ linear batch code.*

Often, we wish to focus on that case $\tau = 1$ for simplicity. The following lemmas, proven in [Ishai et al., 2004], allow us to do this:

Lemma 2.12. *An $[n, k, t, m, \tau]$ batch code for any τ implies an $[n\tau, k, t, m\tau, 1]$ batch code.*

Lemma 2.13. *An $[n, k, t, m, 1]$ batch code implies an $[n, k, t, \lceil \frac{m}{\tau} \rceil, \tau]$ batch code.*

We now give a description of the $(u \mid u + v)$-code construction and the related generator matrix, which can be found in [MacWilliams and Sloane, 1977]. These will be useful in particular for dealing with binary Reed-Muller codes.

Definition 2.14. *Given two linear codes $\mathcal{C}_1, \mathcal{C}_2$ with identical alphabets and block lengths, we may construct a new code $\mathcal{C}$ defined by*

$$\mathcal{C} \coloneqq \{(u \mid u + v) \mid u \in \mathcal{C}_1,\ v \in \mathcal{C}_2\},$$

where $\mid$ represents concatenation. We call this the $(u \mid u + v)$-construction.

Lemma 2.15. *Let G, G_1, and G_2 be the generator matrices for the codes $\mathcal{C}$, $\mathcal{C}_1$, and $\mathcal{C}_2$, respectively, where $\mathcal{C}$ is obtained from $\mathcal{C}_1$ and $\mathcal{C}_2$ via the $(u \mid u+v)$-construction. Then we have*

$$G := \begin{pmatrix} G_1 & G_1 \\ 0 & G_2 \end{pmatrix}$$

Finally, we introduce a property that will be useful for dealing with Cartesian codes. The proof may be seen in [Phillips, 2003, Ch 1].

Lemma 2.16. *For any finite field $\mathbb{F}_q$, if $x_1, x_2, \ldots, x_n \in \mathbb{F}_q$ are distinct, then for any $y_1, \ldots, y_n \in \mathbb{F}_q$, there exists unique polynomial $f \in \mathbb{F}_q[x]$ of degree at most $n-1$ such that $f(x_i) = y_i$ for all $i \in [n]$.*

We note that the proof of this lemma is constructive, and so such a unique polynomial not only exists, but may be determined in the following manner. We construct Lagrange basis polynomials.

Definition 2.17. *For any $j \in [n]$, the* Lagrange basis polynomial *$\ell_j(x)$ is defined by*

$$\ell_j(x) := \prod_{i \neq j \in [n]} \frac{x - x_i}{x_j - x_i}.$$

These polynomials are then taken together in the proper linear combination.

Definition 2.18. *The* Lagrange interpolation polynomial *$L(x)$ is defined by*

$$L(x) = \sum_{j \in [n]} y_j \ell_j(x).$$

It may be verified by examination that evaluating this polynomial at $x_1, \ldots, x_n$ will result in the values $y_1, \ldots, y_n$ as required.

2.2 Preliminary Results

We begin by examining how recovery sets may be formed using the dual of a code.

Lemma 2.19. *If $\mathcal{C}$ is a linear code with dual $\mathcal{C}^\perp$, then for any $h \in \mathcal{C}^\perp$, and any $i \in \text{supp}(h)$, $\text{supp}(h) \setminus \{i\}$ is a recovery set for i.*

Proof. Let $R_i = \text{supp}(h) \setminus \{i\}$. To see that this is a recovery set, we must consider any $c \in \mathcal{C}$. Since $h \in \mathcal{C}^\perp$, we have by definition that $c \cdot h = 0$. Thus, we may write

$$\begin{aligned} 0 &= c \cdot h \\ 0 &= \sum_{j \in \text{supp}(h)} c_j h_j \\ c_i h_i &= -\sum_{j \neq i \in \text{supp}(h)} c_j h_j \\ c_i &= -\sum_{j \neq i \in \text{supp}(h)} c_j h_j h_i^{-1}, \end{aligned}$$

and so c_i may be recovered by reading the symbols $\{c_j | j \in R_i\}$, and R_i is a recovery set by definition. □

We also find that the inverse is true.

Lemma 2.20. *If $\mathcal{C}$ is a linear code with dual $\mathcal{C}^\perp$, then for any $i \in I$, let R_i by any (proper) indirect recovery set for i. Then there exists some $h \in \mathcal{C}^\perp$ such that $R_i \cup \{i\} = \text{supp}(h)$.*

Proof. If R_i as defined above is an indirect recovery set for index i, then by Definition 2.3, for any $c \in \mathcal{C}$, it must hold that c_i may be recovered from $\{c_j \mid j \in R_i\}$. Let ϕ be the function used to recover c_i. That is:

$$c_i = \phi\left(c_j \mid j \in R_i\right).$$

This must also hold for any other $c' \in \mathcal{C}$, and so we may write

$$c'_i = \phi\left(c'_j \mid j \in R_i\right).$$

Finally, we note that this also holds for $c - c'$, and so we have

$$(c - c')_i = \phi\left((c - c')_j \mid j \in R_i\right).$$

Putting these together, we write

$$\begin{aligned}\phi\left((c-c')_j \mid j \in R_i\right) &= (c-c')_i \\ &= c_i - c'_i \\ &= \phi\left(c_j \mid j \in R_i\right) - \phi\left(c'_j \mid j \in R_i\right).\end{aligned}$$

Furthermore, for any $\alpha \in \mathbb{F}_q$, we have that

$$\phi\left((\alpha c)_j \mid j \in R_i\right) = (\alpha c)_i = \alpha c'_i = \phi\left(\alpha c_j \mid j \in R_i\right) - \phi\left(c'_j \mid j \in R_i\right).$$

This makes ϕ a linear function of the inputs, and so we may write $\phi\left(c_j \mid j \in R_i\right) = \sum_{j \in R_i} \alpha_j c_j$, where $\alpha_j \in \mathbb{F}_q$ for all $j \in R_i$. We can then write

$$\begin{aligned}c_i &= \sum_{j \in R_i} \alpha_j c_j \\ 0 &= -c_i + \sum_{j \in R_i} \alpha_j c_j,\end{aligned}$$

and so if we let $h_i = -1$, $h_j = \alpha_j$ for all $j \in R_i$, and $h_j = 0$ for $j \notin R_i \cup \{i\}$, we have $h \cdot c = 0$. This holds for all $c \in \mathcal{C}$, and so by definition, $h \in \mathcal{C}^{\perp}$, and we have $\text{supp}(h) \subseteq R_i \cup \{i\}$. If $\text{supp}(h)$ is a proper subset of $R_i \cup \{i\}$, then this means $\alpha_j = 0$ for some $j \in [n]$. Then j may be removed from R_i and we would still have a recovery set for i. This would in turn mean that R_i was not a proper recovery set to begin with. Thus, we must instead have equality, so $\text{supp}(h) = R_i \cup \{i\}$. □

To examine the batch properties of the $(u \mid v+u)$-code construction, we first introduce a general result for codes that are subsets of other codes.

Theorem 2.21. *Let $\mathcal{C}_1, \mathcal{C}_2$ be codes of length n and dimension k_1 and k_2, respectively such that $\mathcal{C}_1 \subseteq \mathcal{C}_2$. If $\mathcal{C}_2$ is a $[n, k_2, t, m, \tau]$ batch code, then $\mathcal{C}_2$ is a $[n, k_1, t, m, \tau]$ batch code.*

Proof. Note that $\mathcal{C}_2^{\perp} \subseteq \mathcal{C}_1^{\perp}$ because $\mathcal{C}_1 \subseteq \mathcal{C}_2$. Any recovery set R_i for $\mathcal{C}_2$ corresponds to a dual codeword h in $\mathcal{C}_2^{\perp}$. But then we also have $h \in \mathcal{C}_1^{\perp}$. By Lemma 2.19, $R_i = \text{supp}(h) \setminus \{i\}$ is then also a recovery set for i in $\mathcal{C}_1$. Thus, for any query Q, a recovery set R^Q for $\mathcal{C}_2$ will also be a query recovery set for $\mathcal{C}_1$. In turn, a bucket configuration $B_1, \ldots, B_m$ that is t, τ valid will also be t, τ valid for $\mathcal{C}_1$,

and so $\mathcal{C}_1$ is at least a $[n, k_1, t, m, \tau]$ batch code. $\square$

We now introduce results for a $(u \mid u+v)$-code construction.

Theorem 2.22. *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be $[n, k_1]$ and $[n, k_2]$ codes, respectively, such that $\mathcal{C}_2^\perp \subseteq \mathcal{C}_1^\perp$. Let $\mathcal{C}$ be a code such that $\mathcal{C}^\perp$ is a $(u \mid u+v)$-code construction of $\mathcal{C}_1^\perp$ and $\mathcal{C}_2^\perp$. If $\mathcal{C}_2$ is a $[n, k_2, t, m\tau]$ batch code, then $\mathcal{C}$ is a $[2n, k_1+k_2, t, m, \tau]$ batch code.*

Proof. The first two parameters of $\mathcal{C}$ follow from the definition of a $(u \mid u+v)$ construction. Let $\mathcal{C}$ be constructed as described, and let $\mathcal{C}_2$ be an $[n, k_2, t, m, \tau]$ batch code. This means that there exists a bucket configuration $B_1, \ldots, B_m$ that is t, τ valid. Let $B_k' = B_k \cup (n + B_k)$ for $k \in [n]$. That is, construct buckets containing the original indices in B_k and those indices plus n. We will show that this is t, τ valid for $\mathcal{C}$. Consider any query $Q = (i_1, \ldots, i_t) \in [2n]^t$, and let $i_s' = i_s$ if $i_s \in [n]$ and $i_s' = i_s - n$ otherwise. Thus, $Q' = (i_1', \ldots, i_t') \in [n]^t$, and so there exist t disjoint recovery sets $S_{i_1'}, \ldots, S_{i_t'}$ for those indices, the union of which consists of at most τ entries in each of the m buckets $B_1, \ldots, B_m$.

For any $s \in [t]$, if $S_{i_s'}$ is a direct access, then let R_{i_s} be a direct access. That is, $R_{i_s} = \{i_s\}$. If it is not direct access, let $R_{i_s} = S_{i_s'}$ if $i_s \in [n]$ and $R_{i_s} = S_{i_s'} + n$ otherwise. We claim that such an R_{i_s} is a recovery set for i_s. Note that $S_{i_s'} \cup \{i_s'\}$ is the support of some vector $v \in \mathcal{C}_2^\perp$, and since $\mathcal{C}_2^\perp \subseteq \mathcal{C}_1^\perp$, we have that $(v \mid 0), (0 \mid v) \in \mathcal{C}^\perp$ by construction. Hence, in the first case, $i_s \in \operatorname{supp}(v \mid 0) = \operatorname{supp}(v)$, and so R_{i_s} is a recovery set. In the second case, $i_s \in \operatorname{supp}(0 \mid v) = \operatorname{supp}(v) + n$, and so R_{i_s}, as defined, is also a recovery set for i_s.

Since the original $S_{i_s'}$ are all disjoint, so are the R_{i_s}, and so we have t disjoint recovery sets, the union of which consists of at most τ elements from each of m buckets $B_1', \ldots, B_m'$, and this bucket configuration is t, τ valid for $\mathcal{C}$, which means $\mathcal{C}$ is a $[2n, k_1+k_2, t, m, \tau]$ batch code. $\square$

Next, we have a lemma showing that we need only consider the locality of a single index.

Lemma 2.23. *Let $\mathcal{C} \subseteq \mathbb{F}_q^n$ be a linear code and let d' be the minimum distance of $\mathcal{C}^\perp$. If $\mathcal{C}^\perp$ is*

generated by its minimum weight codewords and

$$\bigcup_{\lambda \in \mathcal{C}^{\perp}} \mathrm{supp}(\lambda) = [n], \tag{2.1}$$

then $\mathcal{C}$ has all symbol locality $d' - 1$.

Proof. Condition (2.1) implies that no coordinate of $\mathcal{C}$ is independent of the others. If the minimum weight codewords generate $\mathcal{C}^{\perp}$, then each index $i \in I$ (where I is the index set of $\mathcal{C}$) is in the support of at least one minimum weight codeword h_i of $\mathcal{C}^{\perp}$. Since $\mathrm{supp}(h) = d'$, $R_i = \mathrm{supp}(h) \setminus \{i\}$ is a set of size $d' - 1$, and Lemma 2.19 proves that this R_i is a recovery set, so i has a recovery set of size $d' - 1$. since this holds for any $i \in I$, this implies the all symbol locality of $\mathcal{C}$ is $d' - 1$. □

We note that Condition (2.1) is reasonable to expect for a code. Without it, the code $\mathcal{C}$ would have nonrecoverable coordinates. Finally, we will give a bound that relates the locality property of a linear code to its batch properties.

Lemma 2.24. *Let $\mathcal{C}$ be an $[n, k, t, m, \tau]$ linear batch code with minimal locality r. It holds that*

$$m\tau \geq (t-1)r + 1. \tag{2.2}$$

Proof. We consider such a code $\mathcal{C}$. If for each $i \in I$, there exists some indirect recovery set R_i such that $|R_i| \leq r - 1$ elements, then by the definition of locality, $\mathcal{C}$ has locality $r - 1$, a contradiction to r being the minimal locality. Therefore, there exists some $i \in I$ such that any indirect recovery set R_i has $|R_i| > r - 1$. That is, $|R_i| \geq r$. If we wish to recover this entry t times, then we may use one direct access and $t - 1$ disjoint indirect recovery sets, each of size at least r. This requires reading at least $(t-1)r + 1$ entries, and since we may read at most τ entries from each of the m buckets, we must have that $m\tau \geq (t-1)r + 1$. □

From the perspective of individual devices storing bits of data, $m\tau$ represents the total amount of data read to provide t pieces of the original data. To minimize bandwidth usage in the case where the entries of a codeword represent nodes on a network, we must minimize $m\tau$. This gives rise to the following condition:

Definition 2.25. *A* $[n, k, t, m, \tau]$ *linear batch code* $\mathcal{C}$ *with minimal locality* r *is* optimal *if it satisfies Condition* (2.2) *with equality.*

In the following section, we will see that Hamming codes are optimal linear batch codes.

2.3 Hamming Codes

2.3.1 Definition and Properties

Hamming codes were first introduced in 1950 by Richard Hamming [Hamming, 1950]. In what follows, we consider binary Hamming codes over $\mathbb{F}_2$. The parameters of binary Hamming codes are shown in [MacWilliams and Sloane, 1977, Ch 1]. There, we see that Hamming codes are perfect codes, that is, they achieve the highest possible rate (ratio of message rate to block length) for codes with minimum distance 3. This minimum distance allows the detection of 2 errors and the correction of 1 error. While this may seem limited, it is very useful in situations where the likelihood of errors is low and a small data overhead is desired.

One such application is in computer memory, where the likelihood that any given bit is flipped is very low. This was the motivation for the creation of Hamming codes [Hamming, 1950], and accounted for some of their earliest uses [Dimsdale and Weinberg, 1960]. They are especially useful in applications where the accuracy of the data is critical, such as scientific applications. This was the case with the Cassini-Huygens spacecraft, which recorded a four-fold increase of single-bit errors during a solar proton event [Swift and Guertin, 2000].

We now provide the standard definition of the Hamming code in the binary case:

Definition 2.26. *For some* $s \geq 2$*, let* $H \in \mathbb{F}_2^{2^s-1 \times s}$ *be a matrix whose columns are all of the nonzero vectors of* $\mathbb{F}_2^s$*. Let* $n = 2^s - 1$*. We use* H *as our parity check matrix and define the* binary Hamming code*:*

$$\mathcal{H}_s := \{c \in \mathbb{F}_2^n \mid cH^T = 0\}$$

Due to [Blahut, 2003] Theorem 5.5.1, $\mathcal{H}_s$ is a $[2^s - 1, 2^s - 1 - s, 3]$ cyclic code. Its dual code,

the simplex code, is a $[2^s-1, s, 2^{s-1}]$ cyclic code. For cyclic codes, the locality can be derived from the following in [Huang et al., 2016].

Lemma 2.27. *Let $\mathcal{C}$ be an $[n, k, d]$ cyclic code, and let d' be the minimum distance of its dual code $\mathcal{C}^{\perp}$. Then, the code $\mathcal{C}$ has all symbol locality $d' - 1$.*

This is essentially a special case of Lemma 2.23, relying on each entry being in the support of a minimal weight dual codeword. From this lemma, we know that the locality of $\mathcal{H}_s$ is $2^{s-1}-1$. In the next section, we present the batch properties of binary Hamming Codes.

2.3.2 Batch Properties

Theorem 2.28. *A binary $[n = 2^s - 1, k = 2^s - 1 - s]$ Hamming code is an optimal $[2^s-1, 2^s-1-s, 2, m, \tau]$ batch code for any $m, \tau \in \mathbb{N}$ such that $m\tau = 2^{s-1}$.*

Proof. Let $\mathcal{H}$ be a binary Hamming code with $n = 2^s - 1$, with parity check matrix H. Note that by Lemma 2.24, $m\tau \geq (2-1)(2^{s-1}-1)+1 = 2^{s-1}$. The buckets for $m = 2^{s-1}$, $\tau = 1$ are constructed as follows: The parity check matrix H has columns $h_j \in \mathbb{F}_2^s$ for $1 \leq j \leq n$. If $h_a + h_b = \mathbf{1}$ (the all ones column), then we place a and b into the same bucket. Note that because $h_\ell = \mathbf{1}$ in H, ℓ is placed into its own bucket.

Consider the query $Q = (a, b)$. If a and b are in separate buckets, then the direct access recovery sets $R_a = \{a\}$ and $R_b = \{b\}$ form a query recovery set which trivially satisfied both properties of Definition 2.4. Otherwise, a and b are in the same bucket. We may take $R_a = \{a\}$ and since h_b is not all zeros, consider any d such that entry d of h_b is 1. Then if we let r_d be row d of H, we have that $b \in \text{supp}(r_d)$. If $a \in \text{supp}(r_d)$, then entry d of h_a is also 1, a contradiction to h_a and h_b being complements, and thus being in the same bucket. Therefore, $a \notin \text{supp}(r_d)$. In fact, the same process shows that for any bucket $B_k = \{i, j\}$, $|B_k \cap \text{supp}(r_d)| \leq 1$. That is, it contains at most 1 element from each bucket. Thus, we take $R_b = \text{supp}(r_d) \setminus \{b\}$, which is a recovery set by Lemma 2.19, and we see that R_a, R_b satisfy both properties as well.

Every bucket has cardinality 2 aside from the bucket corresponding to the all ones column in H, so this construction gives us exactly $m = 2^{s-1}$ buckets. Thus, we have shown that the batch

properties hold for $m = 2^{s-1}$ and $\tau = 1$. Further, Lemma 2.13 implies that this is true for any m, τ such that $m\tau = 2^{s-1}$. Since this satisfies $m\tau \geq 2^{s-1}$ with equality, $\mathcal{H}$ is an optimal batch code.

Furthermore, the locality of $\mathcal{H}$ is $2^{s-1} - 1$, and therefore, $t = 2$ is also maximal. Suppose instead that we could have $t \geq 3$. Then, in particular, each entry must be reconstructible at least 3 times. We may use one direct access, but then there must be at least 2 other reconstruction sets used which are disjoint and of size $2^{s-1} - 1$. These would correspond to two codewords in the dual code of weight 2^{s-1} with the intersection of their support being only the given entry. The sum of these codewords will thus have weight $2^{s-1} + 2^{s-1} - 2 = 2^s - 2$. However, the all ones vector is also in the dual code. Adding this vector to the sum will produce a codeword of weight one, a contradiction. Thus, $t = 2$ is maximal. □

Example 2.1. *We now give an example for $s = 3$. This Hamming Code is a $[7, 4]$-linear code, and the dual code is a $[7, 3]$-linear code.*

The parity check matrix H is as follows:

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Thus, the buckets are:

$$\{1, 6\}, \{2, 5\}, \{3, 4\}, \{7\}$$

We note that we are actually able to obtain any pair of bits in the codeword, not just those corresponding to copies of the original information that was encoded. Additionally, we note that although these codes are optimal, we may wish to find batch codes where $t > 2$. These larger values have practical applications where the goal is to quickly distribute data, such as the use case where there are more than two users. Thus, our research moved on to Reed-Muller codes, where we were able to obtain larger t values.

2.4 Reed-Muller Codes

2.4.1 Definition and Background

Reed-Muller codes are well known linear codes. We give some basic properties of these codes, but an interested reader can find more information in [Assmus and Key, 1992].

Definition 2.29. *Let $\mathbb{F}_q[x_1, \ldots, x_\mu]^\rho$ be the set of all multivariate polynomials over $\mathbb{F}_q$ of total degree at most ρ.*

Definition 2.30. *Let $\mathbb{F}_q[x_1, \ldots, x_\mu]$ be the ring of polynomials in μ variables with coefficients in $\mathbb{F}_q$ and let $\mathbb{F}_q^\mu = \{P_1, \ldots, P_n\}$ (so $n = q^\mu$). The q-ary* Reed-Muller code, *$\mathcal{RM}_q(\rho, \mu)$ is defined as:*

$$\mathcal{RM}_q(\rho, \mu) \coloneqq \{(f(P_1), \ldots, f(P_n)) \mid f \in \mathbb{F}_q[x_1, \ldots, x_\mu]^\rho\}.$$

Lemma 2.31 ([Assmus and Key, 1992])**.** *If $\rho < \mu(q-1)$, the dual of a Reed-Muller code $\mathcal{RM}_q(\rho, \mu)$ is $\mathcal{RM}_q(\rho, \mu)^\perp = \mathcal{RM}_q(\mu(q-1) - 1 - \rho, \mu)$.*

2.4.2 Locality and availability properties of $\mathcal{RM}_q(1, \mu)$

Reed-Muller codes for which $\rho = 1$ are known as first-order Reed-Muller codes. We look at the properties using the polynomial evaluation definition of Reed-Muller codes. We begin with a result in the q-ary case.

Theorem 2.32. *Let $\mathbb{F}_q^\mu = \{P_i \mid 1 \leq i \leq q^\mu = n\}$ be the set of evaluation points for $\mathcal{RM}_q(1, \mu)$. Then $(\lambda_1, \ldots, \lambda_n) \in \mathcal{RM}_q(1, \mu)^\perp$ if and only if*

$$\sum_{i=1}^{n} \lambda_i P_i = 0 \text{ and } \sum_{i=1}^{n} \lambda_i = 0. \tag{2.3}$$

Proof. First, if $(\lambda_1, \ldots, \lambda_n)$ is in the dual code, then by definition,

$$\sum_{i=1}^{n} \lambda_i f(P_i) = 0 \tag{2.4}$$

for every polynomial $f \in \mathbb{F}_q[x_1, \ldots, x_\mu]^1$. In particular, note that for any $1 \leq k \leq \mu$, if we define $f_k \in \mathbb{F}_q[x_1, \ldots, x_\mu]$ by $f_k(x_1, \ldots, x_\mu) = x_k$, we have

$$\sum_{i=1}^{n} \lambda_i f_k(P_i) = \sum_{i=1}^{n} \lambda_i p_{i,k} = 0,$$

where $p_{i,k}$ is the kth entry of point P_i. We may gather these equations together for $1 \leq k \leq \mu$ to write the linear combination

$$\sum_{i=1}^{n} \lambda_i P_i = 0.$$

We then consider $f_0 = 1$, and Equation (2.4) becomes $\sum_{i=1}^{n} \lambda_i = 0$, so Equation (2.3) is satisfied.

For the other direction, assume that

$$\sum_{i=1}^{n} \lambda_i P_i = 0 \text{ and } \sum_{i=1}^{n} \lambda_i = 0.$$

Then in particular, for any $1 \leq k \leq \mu$, we have $\sum_{i=1}^{n} \lambda_i p_{i,k} = 0$, and we consider any polynomial $f = a_0 + a_1 x_1 + \cdots + a_\mu x_\mu \in \mathbb{F}_q[x_1, \ldots, x_\mu]^1$. By linearity, we have

$$\sum_{i=1}^{n} \lambda_i f(P_i) = \sum_{i=1}^{n} \lambda_i \left[a_0 + \sum_{k=1}^{\mu} a_k p_{i,k} \right] = a_0 \sum_{i=1}^{n} \lambda_i + \sum_{k=1}^{\mu} a_k \sum_{i=1}^{n} \lambda_i p_{i,k} = 0,$$

and thus $(\lambda_1, \ldots, \lambda_\mu) \in \mathcal{RM}_q(1, \mu)^\perp$. □

From Theorem 2.32 we obtain the following corollaries:

Corollary 2.33. *The minimum distance of $\mathcal{RM}_q(1, \mu)^\perp$ is 4 if $q = 2$ and 3 otherwise.*

Proof. Let $q = 2$ and suppose by way of contradiction that the minimum weight is 2. In that case, there would have to exist two distinct points that sum to zero. This is not possible, and thus the minimum weight must be greater than 2. Note that the only choice of λ_i is 1, and thus the sum $\sum_{i=1}^{n} \lambda_n$ is 0 if and only if $\text{supp}(\lambda)$ is even. Therefore, the weight of the codewords is a multiple of 2, which rules out 3 as the minimum weight. The following points are in $\mathcal{P}$ (for $\mu \geq 2$):

$$P_0 = (0, 0, 0, \ldots, 0)^T, P_1 = (1, 0, 0, \ldots, 0)^T, P_2 = (0, 1, 0, \ldots, 0)^T, \text{ and } P_3 = P_1 + P_2.$$

These points satisfy the conditions, and thus the minimum distance for characteristic 2 is 4.

For $q \neq 2$, let $P_1 = (0,0,0,0,\dots,0)^T, P_2 = (1,0,0,0,\dots,0)^T, P_3 = (-a,0,0,0,\dots,0)^T \in \mathbb{F}_q^\mu$ and the entries of λ corresponding to the positions of P_1, P_2, and P_3 be $-(a+1)$, a, and 1, respectively, with all other entries 0. Then, If $a \neq -1, 0$, λ satisfies Equations (2.3).

Suppose there exists a $\lambda \in \mathcal{RM}_q(1,\mu)^\perp$ with weight 2. Then we have two distinct points $P_i, P_j \in \mathbb{F}_q^\mu$ and $\lambda_i, \lambda_j \in \mathbb{F}_q$ such that $\lambda_j = -\lambda_i$, with $\lambda_k = 0$ for all $k \neq i, j$. Our two conditions imply:

$$\lambda_i P_i - \lambda_i P_j = 0 \implies P_i = P_j,$$

a contradiction to the two points being distinct. Therefore, the minimum distance for characteristic $q \geq 3$ is 3. □

Corollary 2.34. *When $q = 2$ and $\rho \leq \mu - 2$, every codeword in $\mathcal{RM}(\rho,\mu)$ satisfies Equation* (2.3).

Proof. Since $\mathcal{RM}(\rho_1,\mu) \subset \mathcal{RM}(\rho_2,\mu)$ if $\rho_1 < \rho_2$, any codeword in $\mathcal{RM}(\rho,\mu)$ is also in $\mathcal{RM}(\mu-2,\mu)$. By Lemma 2.31, the dual code $\mathcal{RM}(1,\mu)^\perp = \mathcal{RM}(\mu-2,\mu)$, so applying Theorem 2.32 to these codewords guarantees that Equation (2.3) holds. □

We may now move on to examining the locality and availability of first-order Reed-Muller codes.

Theorem 2.35. *Let $q \neq 2$. Then $\mathcal{RM}_q(1,\mu)$ has locality 2 and availability $\delta = \left\lfloor \frac{q^\mu - 1}{2} \right\rfloor$.*

Proof. Let $P_a \in \mathbb{F}_q^\mu$ be an evaluation point. Then consider any $\alpha \in \mathbb{F}_q$ such that $\alpha \neq 0, -1$. We have that $1 + \alpha + (-\alpha - 1) = 0$, and will find corresponding points to use in the reconstruction of P_a. For any choice of $P_b \in \mathbb{F}_q^\mu$ such that $P_b \neq P_a$, let

$$P_c = (\alpha + 1)^{-1}(P_a + \alpha P_b).$$

Upon rearrangement, we have that $P_a + \alpha P_b + (-\alpha - 1)P_c = 0$. We claim that $P_c \neq P_a, P_b$. If $P_c = P_a$, then our equation becomes $P_a + \alpha P_b + (-\alpha - 1)P_a = 0$, which simplifies to $\alpha P_b - \alpha P_a = 0$, which would contradict $P_b \neq P_a$. Likewise, $P_c = P_b$ would imply $P_a + \alpha P_b + (-\alpha - 1)P_b = 0$, which

becomes $P_a - P_b = 0$, another contradiction. With a fixed P_a, we may choose any of the remaining $q^\mu - 1$ points as P_b, and P_c may then be calculated. This, however, will count each pair P_b, P_c twice, and so there are $\left\lfloor \frac{q^\mu - 1}{2} \right\rfloor$ choices of distinct pairs P_b, P_c for P_a. Each of these corresponds to a unique $\lambda \in \mathcal{RM}_q(1,\mu)^\perp$ of weight 3 that can be used to recover c_a, and pairwise intersections of the supports of these vectors contain only $\{a\}$. Thus, the locality is 2 and the availability is $\left\lfloor \frac{q^\mu - 1}{2} \right\rfloor$. $\square$

We introduce an intermediate result which helps build up to the availability for $\mathcal{RM}(1,\mu)$:

Lemma 2.36. *Given any point $P_a \in \mathbb{F}_2^{\ell+2}$, we may write $P_a = (\overline{P_a}^T \mid b, c)^T$ for some $\overline{P_a} \in \mathbb{F}_2^\ell$ and $b, c \in \mathbb{F}_2$. If there are m disjoint sets of three points that sum to $\overline{P_a}$, then there are $4m$ disjoint sets of three points which sum to P_a.*

Proof. Let $P_a, \overline{P_a}, b$, and c be defined as above. Then, for any set of three points $\{S_1, S_2, S_3\}$ such that $S_1 + S_2 + S_3 = \overline{P_a}$, we may write:

$$\begin{aligned}
(S_1^T|b,c)^T + (S_2^T|b,c)^T + (S_3^T|b,c)^T &= P_a \\
(S_1^T|\bar{b},c)^T + (S_2^T|b,\bar{c})^T + (S_3^T|\bar{b},\bar{c})^T &= P_a \\
(S_1^T|b,\bar{c})^T + (S_2^T|\bar{b},\bar{c})^T + (S_3^T|\bar{b},c)^T &= P_a \\
(S_1^T|\bar{b},\bar{c})^T + (S_2^T|\bar{b},c)^T + (S_3^T|b,\bar{c})^T &= P_a,
\end{aligned} \tag{2.5}$$

where $\bar{b} = b + 1$ and $\bar{c} = c + 1$. Because S_1, S_2, and S_3 are all distinct, it can be seen by examining the combinations of $b, c, \bar{b}$, and $\bar{c}$ that each of the points on the left sides in (2.5) is distinct. Each row is thus a set of three distinct points which sum to P_a, and these sets are pairwise disjoint. Furthermore, for any other set $\{S_1', S_2', S_3'\}$ such that $S_1' + S_2' + S_3' = \overline{P_a}$, the only way to have a nonempty intersection between these 4 sets and the 4 sets constructed in the same way from $\{S_1', S_2', S_3'\}$ is if there exists some $i, j \in [3]$ such that $S_i = S_j'$, which means that the two original sets are not disjoint. Thus, given m original disjoint sets that sum to $\overline{P_a}$, this method generates $4m$ disjoint sets of three points that sum to P_a. $\square$

With this building block, we may now move on to examine the availability.

Theorem 2.37. $\mathcal{RM}(1,\mu)$ *has availability $\delta = \frac{2^\mu - 1}{3}$ when μ is even.*

Proof. We use an inductive argument on μ. For a given $\mu \geq 2$ and any $P_a \in \mathbb{F}_2^{\mu}$, we must show that there are $\frac{2^{\mu}-1}{3}$ disjoint sets of three points in $\mathbb{F}_2^{\mu}$ that sum to P_a.

It is easy to verify the claim for $\mu = 2$ since there is only one equation for which this is true: If $P_a = (b, c)$, then

$$(b, \bar{c})^T + (\bar{b}, c)^T + (\bar{b}, \bar{c})^T = (b, c)^T.$$

Now assume the claim holds for $\mu = 2k$, and we claim that it also holds for $\mu = 2k + 2$. For any $P_a \in \mathbb{F}_q^{2k+2}$, we write $P_a = (\overline{P_a}^T \mid b, c)^T$. Then we have $\frac{2^{2k}-1}{3}$ disjoint sets of three points that all sum to $\overline{P_a} = \mathbb{F}_2^{2k}$ by our induction hypothesis, and by Lemma 2.36, we have $4\frac{2^{2k}-1}{3} = \frac{2^{2k+2}-4}{3}$ disjoint sets of three points that sum to P_a. Note that the point $\overline{P_a}$ cannot be in any of the sets, or this would imply that the remaining two distinct points sum to 0, a contradiction. Thus, we also consider the sum

$$(\overline{P_a}^T | \bar{b}, \bar{c})^T + (\overline{P_a}^T | \bar{b}, c)^T + (\overline{P_a}^T | b, \bar{c})^T = P_a, \tag{2.6}$$

and can see that each of the points in (2.6) is distinct from any of those that appear in (2.5). Thus, this introduces one more set for a total of

$$\frac{2^{2k+2}-4}{3} + 1 = \frac{2^{2k+2}-1}{3}.$$

Since this works for any P_a, the availability holds for $\mu = 2k + 2$. By induction, it thus holds for any even $\mu \geq 2$. Finally, note that this number is guaranteed to be an integer because

$$2^{2k+2} \equiv 4^{k+1} \equiv 1^{k+1} \equiv 1 \pmod 3.$$

□

We also note that since there are 2^{μ} symbols, if we take out the symbol being recovered itself, then there can be at most $\frac{2^{\mu}-1}{3}$ triplets from which to make recovery sets. Thus, this is the maximal availability that may be attained.

Theorem 2.38. *$\mathcal{RM}(1, \mu)$ has availability δ at least $\frac{2^{\mu}-4}{4}$ when μ is odd.*

Proof. We again prove this by induction on μ. For $\mu = 3$, if $P_a = (b, c, d)^T$, then

$$(\bar{b}, c, d)^T + (b, \bar{c}, d)^T (\bar{b}, \bar{c}, d)^T = P_a.$$

No combination of the four remaining points of $\mathbb{F}_2^3$ sum to P_a. Thus, we have availability $1 = \frac{2^3-4}{4}$, and so we have our base case.

Now assume that for $\mu = 2k+1$, where $k \geq 1$, we have that the availability of $\mathcal{RM}(1, \mu)$ is at least $\frac{2^\mu - 4}{4}$. Further assume that if $P_a = (\overline{P_a}^T \mid b, c)^T \in \mathbb{F}_q^{\mu+2}$, there are at least 3 points that are not used in any recovery set for $\overline{P_a} \in \mathbb{F}_q^\mu$.

As before, we may use the $\frac{2^\mu-4}{4}$ disjoint sets that sum to $\overline{P_a}$ to make $4\frac{2^\mu-4}{4} = \frac{2^{\mu+2}-16}{4}$ disjoint sets that sum to P_a by Lemma 2.36. We also have points T_1, T_2, and T_3 that are not used in $\mathbb{F}_2^\mu$, and so we may write the following:

$$\begin{aligned}
(\overline{P_a}^T|\bar{b}, c)^T + (T_1^T|b, \bar{c})^T + (T_1^T|\bar{b}, \bar{c})^T &= P_a \\
(\overline{P_a}^T|b, \bar{c})^T + (T_2^T|\bar{b}, \bar{c})^T + (T_2^T|\bar{b}, c)^T &= P_a \\
(\overline{P_a}^T|\bar{b}, \bar{c})^T + (T_3^T|\bar{b}, c)^T + (T_3^T|b, \bar{c})^T &= P_a
\end{aligned}$$

As before, $\overline{P_a}$ cannot appear in any of the previous sets, and all of these points are distinct, so this gives us 3 additional recovery sets, leading to a total of $\frac{2^{\mu+2}-16}{4} + 3 = \frac{2^{\mu+2}-4}{4}$. We also note that $3\left(\frac{2^{\mu+2}-4}{4}\right) + 3 = 3(2^\mu - 1) + 3 = 3 \cdot 2^\mu < 4 \cdot 2^\mu = 2^{\mu+2}$, and so there are at least 3 points not used in any of the recovery sets for P_a. This means that both assumptions hold for $\mu + 2$, and so by induction, we have that for every $k \geq 1$, when $\mu = 2k+1$, the availability of $\mathcal{RM}(1, \mu)$ is at least $\frac{2^\mu-4}{4}$. □

Thus we have achieved a lower bound on δ. Note, however, that we have not shown that this is necessarily an optimal construction.

We next move to the study the batch properties of binary Reed-Muller codes. That is, we take $q = 2$, and as before, instead of writing $\mathcal{RM}_2(\rho, \mu)$, we omit the 2 for convenience. In this

binary case, Reed-Muller codes can be equivalently defined using the $(u \mid u+v)$-code construction, as in [MacWilliams and Sloane, 1977, Ch 13].

Definition 2.39. *Let $\mu, \rho \geq 1$ with $\rho < \mu$. A binary Reed-Muller code $\mathcal{RM}(\rho, \mu)$ is defined as follows:*

$$\mathcal{RM}(\rho, \mu) \coloneqq \{(u \mid u+v) \mid u \in \mathcal{RM}(\rho, \mu-1),\ v \in \mathcal{RM}(\rho-1, \mu-1)\}$$

where $\mathcal{RM}(0, \mu) \coloneqq \{(0,0,\ldots,0),(1,1,\ldots,1)\} \subset \mathbb{F}_2^{2^\mu}$, and $\mathcal{RM}(\mu,\mu) \coloneqq \mathbb{F}_2^{2^\mu}$.

As a consequence, if $G_{\rho,\mu}$ is the generator matrix of the code $\mathcal{RM}(\rho, \mu)$, then

$$G_{\rho,\mu} \coloneqq \begin{pmatrix} G_{\rho,\mu-1} & G_{\rho,\mu-1} \\ 0 & G_{\rho-1,\mu-1} \end{pmatrix}, \tag{2.7}$$

where $G_{0,\mu} = (1,1,\ldots,1) \in \mathbb{F}_2^{1\times 2^\mu}$ and $G_{\mu,\mu} = I_{2^\mu}$.

These binary Reed-Muller codes are particularly useful because their duals satisfy Theorem 2.22. Before we begin examining the batch properties that result because of this, we introduce an intermediate result due to this same recursive structure.

Lemma 2.40. *Let $a \in \mathcal{RM}(\rho-1, \mu-2)$. Then*

$$(a|a|0|0), (a|0|a|0), (a|0|0|a), (0|a|a|0), (0|a|0|a), (0|0|a|a) \in \mathcal{RM}(\rho, \mu),$$

where, for ease of notation, $0 \in \mathbb{F}_2^{\mu}$.

Proof. By (2.7), we have that

$$G = \begin{pmatrix} G_{\rho,\mu-1} & G_{\rho,\mu-1} \\ 0 & G_{\rho-1,\mu-1} \end{pmatrix},$$

where G is the generator of $\mathcal{RM}(\rho, \mu)$. Applying (2.7) again, we obtain that

$$G = \begin{pmatrix} G_{\rho,\mu-2} & G_{\rho,\mu-2} & G_{\rho,\mu-2} & G_{\rho,\mu-2} \\ 0 & G_{\rho-1,\mu-2} & 0 & G_{\rho-1,\mu-2} \\ 0 & 0 & G_{\rho-1,\mu-2} & G_{\rho-1,\mu-2} \\ 0 & 0 & 0 & G_{\rho-2,\mu-2} \end{pmatrix}. \tag{2.8}$$

From the second and third block rows in matrix (2.8), we see that for any $a \in \mathcal{RM}(\rho-1,\mu-2)$, the second row implies $(0|a|0|a) \in \mathcal{RM}(\rho,\mu)$, and the third row implies $(0|0|a|a) \in \mathcal{RM}(\rho,\mu)$. Note that our code is linear, and thus $(0|a|0|a)+(0|0|a|a) = (0|a|a|0) \in \mathcal{RM}(\rho,\mu)$. Finally, note that since $\mathcal{RM}(\rho-1,\mu-2) \subseteq \mathcal{RM}(\rho,\mu-2)$, the first row implies $(a|a|a|a) \in \mathcal{RM}(\rho,\mu)$, and so combining this with the previous vectors, we find that $(a|a|0|0), (a|0|a|0), (a|0|0|a) \in \mathcal{RM}(\rho,\mu)$. □

2.4.3 Batch Properties of $\mathcal{RM}(1,\mu)$

We begin with what essentially forms the base case, $\mathcal{RM}(1,4)$.

Theorem 2.41. *The linear code $\mathcal{RM}(1,4)$ is a $[16,5,4,m,\tau]$ batch code for any $m,\tau \in \mathbb{N}$ such that $m\tau = 10$.*

Proof. First, note that the dual code of $\mathcal{RM}(1,4)$ is $\mathcal{RM}(2,4)$, which informs us how to reconstruct elements of the codewords The generator matrix for $\mathcal{RM}(1,4)$ can be recursively constructed as follows by (2.7):

$$G_{1,4} = \begin{pmatrix} G_{1,3} & G_{1,3} \\ 0 & G_{0,3} \end{pmatrix}.$$

It can be verified by brute force that any query of 4 coordinates of a codeword in $\mathcal{RM}(1,4)$ is possible with the following partition into buckets:

$$\{1\}, \{2\}, \{3\}, \{4\}, \{5,6\}, \{7,8\}, \{9,11\}, \{10,12\}, \{13,16\}, \{14,15\}.$$

That is, these buckets are 4, 1-valid. In this case, $m = 10$ and $\tau = 1$. By Lemma 2.13, this holds for any $m,\tau \in \mathbb{N}$ such that $m\tau = 10$. □

Next, we show how to extend this construction to $\mathcal{RM}(1,\mu)$ for any $\mu \geq 4$.

Theorem 2.42. *Any first-order Reed-Muller code, $\mathcal{RM}(1,\mu)$, with $\mu \geq 4$, has batch properties $(n,k,4,m,\tau)$ for any $m,\tau \in \mathbb{N}$ such that $m\tau = 10$.*

Proof. We will proceed by induction. We have just shown that it holds for the base case where $\mu = 4$. Now, assume that for some $\mu \geq 4$, we have that $\mathcal{RM}(1, \mu)$ has batch properties $(n, k, 4, m, \tau)$. By Lemma 2.31, we see that the dual code of $\mathcal{C} = \mathcal{RM}(1, \mu+1)$ is $\mathcal{C}^{\perp} = \mathcal{RM}(\mu-1, \mu+1)$. Further, by Definition 2.39, $\mathcal{C}^{\perp}$ is the $(u \mid u+v)$-code construction of $\mathcal{C}_1^{\perp} = \mathcal{RM}(\mu-1, \mu)$ and $\mathcal{C}_2^{\perp} = \mathcal{RM}(\mu-2, \mu)$. Since $\mathcal{RM}(\mu-2, \mu) \subseteq \mathcal{RM}(\mu-1, \mu)$, we have $\mathcal{C}_2^{\perp} \subseteq \mathcal{C}_1^{\perp}$, meaning we may apply Theorem 2.22. Since $\mathcal{C}_2 = \mathcal{RM}(1, \mu)$, we know that $\mathcal{C}$ is also an $(n, k, 4, m, \tau)$ batch code. Thus, these properties hold for $\mu + 1$. By induction, for $\mu \geq 4$, $\mathcal{RM}(1, \mu)$ has batch properties $(n, k, 4, m, \tau)$ for any $m, \tau \in \mathbb{N}$ such that $m\tau = 10$. □

From this, we may make the following observation:

Theorem 2.43. *First-order binary Reed-Muller codes are optimal for $\mu \geq 4$.*

Proof. Since the locality of these codes is $r = 3$, for $t = 4$, we have $m\tau = 10 = (4 - 1) \cdot 3 + 1 = (t - 1)r + 1$, and thus (2.2) is satisfied with equality, and we have optimal batch properties. □

We now use this result to examine the batch properties for a broader class of $\mathcal{RM}(\rho, \mu)$. Specifically, we shall examine when $\rho \geq 1$ and $\mu \geq 2\rho + 2$.

Theorem 2.44. *Let $\rho \geq 1$ and $\mu \in \{2\rho+2, 2\rho+3\}$. Then, for $\rho' \leq \rho$, $\mathcal{RM}(\rho', \mu)$ is a $(n, k, 4, m, \tau)$ linear batch code for any $m, \tau \in \mathbb{N}$ such that $m\tau = 10 \cdot 2^{2\rho-2}$.*

Proof. We focus on the case where $\mu = 2\rho + 2$, as the case $\mu = 2\rho + 3$ proceeds with similar steps.

If $\mathcal{RM}(\rho, 2\rho + 2)$ is an $(n, k, 4, m, \tau)$ linear batch code for any $m, \tau \in \mathbb{N}$ such that $m\tau = 10 \cdot 2^{2\rho-2}$, then it follows from Theorem 2.21 that for any $\rho' \leq \rho$, the code $\mathcal{RM}(\rho', 2\rho + 2)$ is an $(n, k, 4, m, \tau)$ batch code for any $m, \tau \in \mathbb{N}$ such that $m\tau = 10 \cdot 2^{2\rho-2}$. Thus, we need only prove that this holds for $\mathcal{RM}(\rho, 2\rho + 2)$.

We proceed by induction on ρ. Note that by Theorem 2.42, the claim is true for $\rho = 1$, the base cases with $\mu = 4, 5$. Now assume that the claim holds for some $\rho \geq 1$. We show that it also holds for $\mathcal{RM}(\rho + 1, 2(\rho + 1) + 2) = \mathcal{RM}(\rho + 1, 2\rho + 4)$. By assumption, $\mathcal{RM}(\rho, 2\rho + 2)$ is a $(n, k, 4, m, \tau)$ linear batch code for any $m, \tau \in \mathbb{N}$ such that $m\tau = 10 \cdot 2^{2\rho-2}$. In particular, we may choose $\tau = 1$ and have $m = 10 \cdot 2^{2\rho-2}$ buckets.

We now examine $\mathcal{RM}(\rho+1, 2\rho+4)$. By Lemma 2.31, the dual code of $\mathcal{RM}(\rho+1, 2\rho+4)$ is $\mathcal{RM}(\rho+2, 2\rho+4)$. By Lemma 2.40, for any $a \in \mathcal{RM}(\rho+1, 2\rho+2) = \mathcal{RM}(\rho, 2\rho+2)^{\perp}$, we have

$$(a|a|0|0), (a|0|a|0), (a|0|0|a), (0|a|a|0), (0|a|0|a), (0|0|a|a) \in \mathcal{RM}(\rho+2, 2\rho+4).$$

This provides a way to produce parity check equations for $\mathcal{RM}(\rho+1, 2\rho+4)$ from those for $\mathcal{RM}(\rho, 2\rho+2)$, which in turn provides a way to make recovery sets for the former from those for the latter, as each vector corresponds to a recovery set for every index in its support by Lemma 2.19.

For each bucket $B_k = \{i_1, \ldots, i_\ell\}$ for $\mathcal{RM}(\rho, 2\rho+2)$, define four buckets for $\mathcal{RM}(\rho+1, 2\rho+4)$ as $B_{k,1} = B_k$, $B_{k,2} = B_k + n = \{i_1+n, \ldots, i_\ell+n\}$, $B_{k,3} = B_k + 2n$, and $B_{k,4} = B_k + 3n$. This results in $4 \cdot 10 \cdot 2^{2\rho-2} = 10 \cdot 2^{2\rho} = 10 \cdot 2^{2(\rho+1)-2}$ buckets. We must show that any query of 4 indices may be recovered by drawing at most 1 entry from each bucket.

Consider any query of 4 indices $Q = (i_1, i_2, i_3, i_4) \in [4n]^4$. We write $[4n] = \bigcup_{m=0}^{3}([n] + mn)$. Let $m_s = \left\lfloor \frac{i_s-1}{n} \right\rfloor$ for $s \in [4]$ and let $i'_s = i_s - m_s n$, so that $i'_1, i'_2, i'_3, i'_4 \in [n]$. Then define $Q' = (i'_1, i'_2, i'_3, i'_4) \in [n]^4$. By the induction hypothesis, for this query Q', we have a query recovery set $R^Q = \{R_{i'_1}, R_{i'_2}, R_{i'_3}, R_{i'_4}\}$ satisfying Definition 2.4:

1. $\left| \left(\bigcup_{s=1}^{4} R_{i'_s} \right) \cap B_k \right| \leq 1 \ \forall\ k \in [m]$, and
2. $R_{i'_r} \cap R_{i'_s} = \emptyset \ \forall\ r, s \in [4]$ where $r \neq s$.

Each $R_{i'_s}$ is either $\{i'_s\}$ or $\operatorname{supp} a_s \setminus \{i'_s\}$ for some vector $a_s \in \mathcal{RM}(\rho+1, 2\rho+2)$ (the dual of $\mathcal{RM}(\rho, 2\rho+2)$) with $i'_s \in \operatorname{supp}(a)$. If $R_{i'_s} = \{i'_s\}$, then let $R'_{i_s} = R_{i'_s} + m_s n = \{i'_s + m_s n\} = \{i_s\}$. That is, for any direct access of an index in $\mathcal{RM}(\rho, 2\rho+2)$, we will use direct access in $\mathcal{RM}(\rho, 2\rho+4)$. Otherwise, since a is a vector in $\mathcal{RM}(\rho, 2\rho+2)$ with support $R_{i'_s} \cap \{i'_s\}$, we know by Lemma 2.40 that there is a vector a'_s with support $(\operatorname{supp}(a_s) + m_s n) \cup (\operatorname{supp}(a_s) + m'_s n)$ in $\mathcal{RM}(\rho+2, 2\rho+4)$ for any $m'_s \in \{0, 1, 2, 3\}$ such that $m'_s \neq m_s$. Because $i'_s \in \operatorname{supp}(a_s)$, we find $i_s = i'_s + m_s n \in \operatorname{supp}(a_s) + m_s n$, so $i_s \in \operatorname{supp}(a'_s)$, which means $T'_{i_s, m'_s} = \operatorname{supp}(a'_s) \setminus \{i_s\}$ is a valid recovery set for i_s. We must now show that the correct choice of values m'_s will lead to a query recovery set with the properties required.

From the $R_{i'_s}$ and T_{i_s,m'_s} as defined above, we now seek to construct recovery sets R'_{i_s} for Q. Note that since indices are being recreated from $d = |\{m_1, m_2, m_3, m_4\}|$ different quarters of $[4n]$, we can take at least d of the recovery sets R'_{i_s} to be direct access, even if $R_{i'_s}$, as defined above, was not. Further, assume that we take as many recovery sets to be singletons as possible.

We claim that under this assumption, no recovery set will contain more than one index in each bucket. Certainly a direct access cannot, and if we take $R'_{i_s} = T_{i_s,m'_s}$ and there is some bucket accessed twice, this would imply that there is some bucket that $(\text{supp}(a_s)+m_s n)\cup(\text{supp}(a_s)+m'_s n)$ contains two indices from. Since all indices in $\text{supp}(a_s) + m_s n$ come from buckets of the form B_{k,m_s+1}, and all indices in $\text{supp}(a_s) + m'_s n$ come from buckets of the form B_{k,m'_s+1}, and we have $m_s \neq m'_s$, the only way this can happen is if there is some B_k such that $\text{supp}(a_s)$ contains two indices in B_k. We know $\text{supp}(a_s) \setminus \{i'_s\}$ is a valid recovery set, so it contains at most one index from each B_k. Thus, the bucket from which two indices occurred in $\text{supp}(a_s)$ would have to be the one in which i'_s lies. But this means that no other recovery set uses that bucket B_k, and so no T_{i_r,m'_r} for $r \in [4]$, $r \neq s$ will use B_{k,m_s+1}. Since B_{k,m_s+1} contains i_s, we could have done direct access for i_s, a contradiction.

As a result, we have at most $4-d$ recovery sets which are not direct, and therefore must be of the form $R'_{i_s} = T_{i_s,m'_s} = \text{supp}(a'_s) \setminus \{i_s\}$, which requires reading $\text{supp}(a_s) + m'_s n$, a subset in a different quarter from the first. Assume without loss of generality that these are $R'_{i_1}, \ldots, R'_{i_{4-d}}$. Then we may let $m'_1, \ldots, m'_{d-4}$ be the elements of $\{0,1,2,3\} \setminus \{m_1, m_2, m_3, m_4\}$. Since these are distinct, the sets $\text{supp}(a_s) + m'_s n$ lie in distinct quarters, and so there are no intersections between them. This means the only way some R'_{i_s} and R'_{i_r} could have a nonempty intersection is if there is an intersection between $\text{supp}(a_s)+m_s n$ and $\text{supp}(a_r)+m_r n$. In particular, this means that $m_s = m_r$, and so there is an intersection between $\text{supp}(a_r)$ and $\text{supp}(a_s)$. By Condition (2) for $R^{Q'}$, we have that $\text{supp}(a_r) \setminus i'_r$ and $\text{supp}(a_s) \setminus i'_s$ have an empty intersection, so the intersection must have been at either i'_r or i'_s. This implies the intersection for $\text{supp}(a_s)+m_s n$ and $\text{supp}(a_r)+m_r n$ was at i_r or i_s. Since R'_{i_s} does not contain i_s, and R_{i_r} does not contain i_r, this intersection is impossible. Thus, Condition (1) also holds for our R^Q.

We have already covered the fact that none of the $\text{supp}(a_s) + m'_s n$ will contain more than one index in each bucket, and since these are in separate quarters, the only way $\bigcup_{s=1}^{4} R'_{i_s}$ would

contain more than one index in a bucket would be if some indices i_r and i_s are being recovered in the same quarter and $[(\text{supp}(a_r) + m_r n) \setminus \{i_r\}] \cup [(\text{supp}(a_s) + m_s n) \setminus \{i_s\}]$ consists of more than 1 index in some bucket. However, this would mean that $(\text{supp}(a_r) \setminus i'_r) \cup (\text{supp}(a_s) \setminus i'_s)$, which is the same as $R_{i'_r} \cap R_{i'_s}$, has indices in the same bucket. This would violate Condition (1) for $R^{Q'}$. Thus, we instead have that Condition (1) holds for R^Q.

Since this can be done for any query $Q = (i_1, i_2, i_3, i_4) \in [4n]^4$, we find that $\mathcal{RM}(\rho+1, 2\rho+4)$ is a $(4n, k', 4, 10 \cdot 2^{2(\rho+1)-2}, 1)$ batch code, and by Lemma 2.13, we know that $\mathcal{RM}(\rho+1, 2(\rho+1)+2)$ is a $(4n, k', 4, m, \tau)$ batch code for any $m, \tau \in \mathbb{N}$ such that $m\tau = 10 \cdot 2^{2(\rho+1)-2}$. This completes the induction step, and so for any $\rho \geq 1$, $\mathcal{RM}(\rho, 2\rho+2)$ is a $(4n, k', 4, m, \tau)$ batch code for any $m, \tau \in \mathbb{N}$ such that $m\tau = 10 \cdot 2^{2\rho-2}$. □

Moving onward, we switch gears to focus on another type of codes for which we studied the batch properties.

2.5 Cartesian Codes

Throughout this section, we will take $\tau = 1$, while the number of buckets m will not be defined directly but rather be a consequence of the subspace construction given in Section 2.5.2. This is followed by a proof of several equivalent conditions to $V \cap \langle e_i, e_j \rangle = \{0\} \, \forall \, i, j \in [\mu]$, where e_i and e_j are canonical basis vectors as in Definition 1.9. This is a condition we require of any subspace V used in the construction. This in turn leads to a proof of a method to expand a code in $\mathbb{F}_q^{\mu-1}$ into a code over $\mathbb{F}_q^{\mu}$. Finally, in Section 2.5.4, these methods are applied to general affine Cartesian Codes. We use the definition of a Cartesian Code as described in [López et al., 2014]:

Definition 2.45. *Let $\mathbb{F}_q$ be an arbitrary field, and $A_1, \ldots, A_\mu$ be finite non-empty subsets of $\mathbb{F}_q$. We define $X = A_1 \times \ldots \times A_\mu \subseteq \mathbb{F}_q^\mu$. Take the polynomial ring $S = \mathbb{F}_q[x_1, \ldots, x_\mu]$ and define $S_{\leq \rho}$ to be the $\mathbb{F}_q$ vector space of all polynomials in S with degree at most ρ. If $P_1, \ldots P_n$ are the points of*

X, we define the map:

$$ev_{rho} : S_{\leq\rho} \to \mathbb{F}_q^{|X|}$$
$$f \mapsto (f(P_1), \ldots, f(P_n)).$$

We define the affine Cartesian code of degree ρ, denoted $\mathcal{C}_X(\rho)$, to be the image of ev_ρ.

We now introduce an important lemma which will help us reconstruct values:

Lemma 2.46. *For any point $x = (a_1, \ldots, a_\mu) \in X$, and any $i \in [\mu]$, let*

$$R_{x,i} = \{(b_1, \ldots, b_\mu) \mid b_i \in A_i, b_j = a_j \forall j \neq i\} \setminus \{x\}.$$

If $d + 1 < |A_i|$, then for any $f \in S_{\leq\rho}$, the value of $f(x)$ can be recovered using the values $f(R_{x,i})$.

Proof. For any $x \in X$ as above, let $R_{x,i}$ be defined as above and consider any $f \in S_{\leq\rho}$, where $\rho + 1 \leq |A_i|$. We write $f_i(x_i) = f(a_1, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_\mu) \in \mathbb{F}_q[x_i]$. By the construction of $R_{x,i}$, we have that $f(R_{x,i}) = f_i(A_i \setminus \{a_i\})$. Since f is a polynomial of total degree at most ρ, $f_i(x_i)$ is a polynomial of degree at most ρ in x_i. If $\rho + 1 < |A_i|$, then $\rho \leq |A_i| - 2$, so f_i is of degree at most $|A_i| - 2$. By Lemma 2.16, we may find a unique polynomial $g(x_i) \in \mathbb{F}_q[x_i]$ of degree at most $|A_i \setminus \{a_i\}| - 1 = |A_i| - 2$ such that $g(a) = f_i(a)$ for all $a \in A_i \setminus \{a_i\}$, and so we must have $g = f_i$. We find that $g(a_i) = f_i(a_i) = f(a_1, \ldots, a_\mu) = f(x)$, and so we can recover $f(x)$. □

We thus concentrate only on cases where $\rho+1 < |A_i|$ for all $i \in [\mu]$. If we take $\mathbb{F}_q = \mathbb{F}_q$, then $|A_i| \leq q$ for all $i \in [\mu]$, and so we must have $\rho + 1 < q$, or $\rho < q - 1$. We thus initially consider the Cartesian code $C_Y(\rho)$, where $Y = \mathbb{F}_q \times \cdots \times \mathbb{F}_q = \mathbb{F}_q^\mu$, and $\rho < q - 1$. This is in fact the Reed-Muller code $\mathcal{RM}_q(\rho, \mu)$.

2.5.1 Cartesian Recovery Sets

For any $x \in Y = \mathbb{F}_q^\mu$, we define the following recovery sets:

Definition 2.47. *The recovery sets of* $x \in \mathbb{F}_q^\mu$ *are:*

$$R_{x,0} = \{x\}$$
$$R_{x,i} = (x + \langle e_i \rangle) \setminus \{x\} \qquad \forall i \in [\mu].$$

As before, we refer to $R_{x,0}$ as a *direct access* of x and $R_{x,i}$ for $i > 0$ as an *indirect recovery* of x. Indirect recovery sets correspond to one-dimensional vector subspaces of $\mathbb{F}_q^\mu$, where i is the only dimension that has varying entries.

Corollary 2.48. *For any query* $Q = (x_1, \ldots, x_{\mu+1}) \in (\mathbb{F}_q^\mu)^{\mu+1}$, *using the indices in a query recovery set* $R^Q = \{R_{x_1,i_1}, \ldots, R_{x_{\mu+1},i_{\mu+1}}\}$, *it is possible to recover* $f(x_1), \ldots, f(x_{\mu+1})$.

Proof. For any $s \in [\mu+1]$ such that $i_s = 0$, we note that $R_{x_s,i_s} = R_{x_s,0} = \{x_s\}$, and so this is direct access, and we may simply calculate $f(x_s)$. That these are recovery sets for x_s such that $i_s \neq 0$ follows from Lemma 2.46, noting that with $X = Y$, the two definitions of R_{x_j,i_j} coincide. □

We will leave off the Q in R^Q when the context makes the query unambiguous. To be more precise about batch properties, we restate the conditions that every query recovery set must satisfy for a bucket configuration to be valid with $t = n + 1$ and $\tau = 1$, the parameters we will be using throughout this section:

1. $\left| \left(\bigcup_{s=1}^{\mu+1} R_{x_s,i_s} \right) \cap B_k \right| \leq 1 \ \forall \ k \in [m]$, and
2. $R_{x_r,i_r} \cap R_{x_s,i_s} = \emptyset \ \forall \ r, s \in [\mu+1]$ where $r \neq s$.

The first condition corresponds to using at most $\tau = 1$ indices in any given bucket, while the second corresponds to having non-overlapping recovery sets.

2.5.2 Subspace Bucket Construction

With requirements for valid bucket configurations addressed, we now define the bucket configuration used in this section.

Definition 2.49. *For any subspace V of $\mathbb{F}_q^\mu$, consider the quotient space $\mathbb{F}_q^\mu/V$. The equivalence classes $[a] = a + V$ partition $\mathbb{F}_q^\mu$. We define a subspace bucket construction to be one where the buckets are these equivalence classes.*

Definition 2.50. *We denote by "$\sim$" the equivalence relation imposed by V. That is, $x \sim y$ if and only if $[x] = [y]$.*

With this bucket configuration, we have $m = \frac{|\mathbb{F}_q^\mu|}{|V|} = \frac{q^\mu}{q^{\dim(V)}} = q^{\mu - \dim V}$. This construction provides us with a great deal of symmetry and structure, which allows us to approach determining the validity of a given subspace bucket construction with the following tools.

For any $a \in \mathbb{F}_q^\mu$, the set $[a] = a + V$ is all elements in the same bucket as a by definition. For ease of notation, we introduce the following definition.

Definition 2.51. *For any subset $U \subseteq \mathbb{F}_q^n$, let*

$$[U] = \{[x] \mid x \in U\}.$$

That is, $[U]$ is the set of all buckets corresponding to points in U. It is important to keep in mind that it is a set of sets, not a set of points. With this notation, we now note an important result with respect to recovery sets for equivalent points.

Lemma 2.52. *With the subspace construction, if $x \sim y$, then $[R_{x,i}] = [R_{y,i}]$ for all $i \in [\mu]$.*

This means that under the equivalence relation, the recovery sets for elements in the same bucket are the same. This identical use of buckets for the recovery sets leads to the following:

Corollary 2.53. *For any query $Q = (x_1, \ldots, x_{\mu+1}) \in (\mathbb{F}_q^\mu)^{\mu+1}$, if there is some $s \leq \mu + 1$ such that $x_1 \sim \cdots \sim x_s$, let $Q' = (x'_1, \ldots, x'_{\mu+1})$, where $x'_r = x_1$ for $1 \leq r \leq s$ and $x'_r = x_r$ otherwise. Any query recovery set $R^{Q'}$ is a query recovery set for Q, and any query recovery set R^Q is a query recovery set for Q'.*

In other words, we may effectively treat recovering multiple points in the same bucket as recovering the same point multiple times. This leads naturally to notation for all recovery sets of a point.

Definition 2.54. *For any point $x \in \mathbb{F}_q^n$, let*

$$R_x = \{R_{x,0}, \ldots, R_{x,\mu}\}.$$

For convenience, we also introduce notation for the union of all such recovery sets for an element.

Definition 2.55. *The* star *of an element x, denoted E_x, is defined as*

$$E_x = \bigcup_{i=0}^{\mu} R_{x,i}.$$

We now reach the central theorem which will be used to verify the validity of subspace bucket constructions.

Theorem 2.56. *The following are equivalent:*

(a) V has minimum distance $d \geq 3$.

(b) $V \cap \langle e_i, e_j \rangle = \{0\} \, \forall \, i, j \in [\mu]$.

(c) For any $x \in \mathbb{F}_q^\mu$, and any $a, b \in E_x$, $a \neq b \implies [a] \neq [b]$.

(d) For any $x \in \mathbb{F}_q^\mu$, R_x is a query recovery set for $Q = (x, \ldots, x)$.

Proof. First, since E_x is just the union of the sets in $R_x = \{R_{x,i} \mid 0 \leq i \leq \mu\}$, if R_x is a query recovery set, Condition (1) implies that each point is in a separate bucket, so for all $a, b \in E_x$, $a \neq b \implies [a] \neq [b]$. Similarly, if each point in E_x is in a different bucket, then Condition (1) is satisfied, and the sets $R_{x,0}, \ldots, R_{x_\mu}$ are all disjoint by construction, so Condition (2) is satisfied. This makes R_x a query recovery set for $Q = (x, \ldots, x)$. This means (c) $\Leftrightarrow$ (d).

Next, we show that (c) $\implies$ (b). Assume for contradiction that (c) holds, but (b) does not. This would mean that there are some $i, j \in [\mu]$ such that $V \cap \langle e_i, e_j \rangle \neq \{0\}$. Thus, consider any $a \in V \cap \langle e_i, e_j \rangle$ such that $a \neq 0$. Fix any $x \in \mathbb{F}_q^\mu$. We have $a = \alpha e_i + \beta e_j$, with α, β not both 0. We write $0 \neq \alpha e_i + \beta e_j$, so $-\beta e_j \neq \alpha e_i$, and $x - \beta e_j \neq x + \alpha e_i$. However, we know that

$x+\alpha e_i, x-\beta e_j \in E_x$, and since $(x+\alpha e_i)-(x-\beta e_j) = \alpha e_i+\beta e_j = a \in V$, we have $[x+\alpha e_i] = [x-\beta e_j]$, a contradiction to $a \neq b \implies [a] \neq [b]$.

For (b) $\implies$ (c), assume (b) holds, but (c) does not. Then there is some x and some $a, b \in E_x$ such that $a \neq b$, but $[a] = [b]$. This would mean that $a - b \in V$. Given the structure of E_x, we may write $a = x + \alpha e_i$ and $b = x + \beta e_j$. We see that $\alpha = 0$ if and only if $a = x$, and $\beta = 0$ if and only if $b = x$. Since $a \neq b$, we cannot have $a = x = b$, and thus at most one of α, β may be 0. This means that $a - b = (x + \alpha e_i) - (x + \beta e_j) = \alpha e_i - \beta e_j$. We began with $a - b \in V$ and have just shown that $a - b \in \langle e_i, e_j\rangle$. This means that $a - b \in V \cap \langle e_i, e_j\rangle$, but $a - b \neq 0$, so $V \cap \langle e_i, e_j\rangle \neq \{0\}$, a contradiction.

Finally, we show that (b) $\Leftrightarrow$ (a). Certainly, as V is a vector subspace of $\mathbb{F}_q^n$, V may be interpreted as a code over $\mathbb{F}_q^n$. We prove (b) $\Leftrightarrow$ (a) by contrapositives. $V \cap \langle e_i, e_j\rangle = \{0\} \,\forall\, i, j \in [\mu]$ is false if and only if $\exists i, j \in [\mu]$ such that $V \cap \langle e_i, e_j\rangle \neq \{0\}$. Since $\{0\} \subseteq V$ and $\{0\} \subseteq \langle e_i, e_j\rangle$, this condition is satisfied if and only if there is some $x = \alpha e_i + \beta e_j \in V$, for $\alpha, \beta \in \mathbb{F}_q$ not both zero. But since $w(x) \leq 2$, this occurs if and only if the minimum distance of V is $d < 3$. By proving the contrapositives, we have now shown that

$$V \cap \langle e_i, e_j\rangle = \{0\} \,\forall\, i, j \in [\mu] \Leftrightarrow d \geq 3.$$

This completes the equivalences. □

These equivalent conditions lead to some important necessary conditions.

Corollary 2.57. *Any bucket configuration based on the quotient of $\mathbb{F}_q^\mu$ by a subspace $V \subset \mathbb{F}_q^\mu$ must satisfy $V \cap \langle e_i, e_j\rangle = \{0\} \,\forall\, i, j \in [\mu]$ to provide a valid batch code.*

Corollary 2.58. *If a bucket configuration is based on a subspace $V \subset \mathbb{F}_q^\mu$, then the minimum distance of V as a code over $\mathbb{F}_q$ must be $d \geq 3$.*

It is important to note that Corollary 2.58 gives a necessary, but not sufficient, condition. That it is necessary follows from the need to satisfy the query $Q = (x, \ldots, x)$, for which R_x represents all recovery sets. That it is not sufficient follows from the need to satisfy other queries, which this condition does not guarantee. Next, we specify a way that a batch code over $\mathbb{F}_q^{\mu-1}$ may be expanded

to a batch code over $\mathbb{F}_q^\mu$.

Lemma 2.59. *Let $\phi : \mathbb{F}_q^n \to \mathbb{F}_q^{\mu-1}$ be defined by $\phi((a_1, \ldots, a_n)) = (a_1, \ldots, a_{\mu-1})$. If $\overline{\mathcal{C}} \subset \mathbb{F}_q^{\mu-1}$ is a batch code that can satisfy any query of $t' = \mu$ elements using a bucket construction based on the subspace $\overline{V} \subseteq \mathbb{F}_q^{\mu-1}$, then for any vector subspace $V \subseteq \mathbb{F}_q^\mu$ such that $\phi(V) = \overline{V}$ satisfying $V \cap \langle e_i, e_j \rangle = \{0\} \, \forall \, i, j \in [\mu]$, the code $\mathcal{C} \subseteq \mathbb{F}_q^\mu$ using subspace V can satisfy any query of $t = \mu + 1$ elements.*

Proof. Suppose V is a subspace of $\mathbb{F}_q^\mu$ such that $\phi(V) = \overline{V}$. We claim that for any $a \in \mathbb{F}_q^\mu$, it holds that $\phi([a]) \subseteq [\phi(a)]$. Certainly, for any $b \in [a]$, we have that $a - b \in V$. This means that $\phi(a - b) \in \phi(V) = \overline{V}$, and by linearity of ϕ, we have $\phi(a) - \phi(b) \in \overline{V}$. This in turn means $\phi(b) \in [\phi(a)]$. Since this is true for any $\phi(b) \in \phi([a])$, we have that $\phi([a]) \subseteq [\phi(a)]$. From this, we see that $a \sim b \Rightarrow [b] = [a] \Rightarrow \phi([b]) \subset [\phi(a)]$. Since $\phi(b) \in \phi([b])$, and $\phi(b) \in [\phi(a)] \Rightarrow [\phi(a)] = [\phi(b)]$, we have in particular that $a \sim b \Rightarrow \phi(a) \sim \phi(b)$.

Now consider any query $Q = (x_1, \ldots, x_n, x_{\mu+1})$ of points in $\mathbb{F}_q^\mu$. The multiset $Q' = (\phi(x_1), \ldots, \phi(x_\mu))$ is a query of μ elements in $\mathbb{F}_q^{\mu-1}$. For any $a \in \mathbb{F}_q^\mu$, let $\overline{a} = \phi(a) \in \mathbb{F}_q^{\mu-1}$. Then we may write $Q' = (\overline{x_1}, \ldots, \overline{x_\mu})$, and since $\overline{\mathcal{C}}$ is a batch code that can satisfy any query of size $t' = \mu$, there exists some query recovery set $\overline{R} = \{R_{\overline{x_1}, i_1}, \ldots, R_{\overline{x_\mu}, i_\mu}\}$ such that

1. $\left| \left(\bigcup_{s=1}^{n} R_{\overline{x_s}, i_\ell} \right) \cap [b] \right| \leq 1 \; \forall \, [c] \in \mathbb{F}_q^{n-1} / \overline{V}$
2. $R_{\overline{x_r}, i_r} \cap R_{\overline{x_s}, i_s} = \emptyset \; \forall \; r, s \in [\mu]$ where $r \neq s$.

Now let $E = \cup_{s=1}^n R_{x_s, i_s}$, and if $\exists z \in E$ such that $z \sim x_{\mu+1}$, then let $i_{\mu+1} = \mu$. Otherwise, let $i_{\mu+1} = 0$. We claim that

$$R = \{R_{x_1, i_1}, \ldots, R_{x_\mu, i_\mu}, R_{x_{\mu+1}, i_{\mu+1}}\}$$

is a solution set for the query $Q = (x_1, \ldots, x_{\mu+1})$. By construction, we have that $\phi(R_{x_s, i_s}) = R_{\overline{x_s}, i_s}$ for $s \in [\mu]$.

First, we check Condition (1) for R. We will assume for a contradiction that it is not satisfied. That is, suppose there is some $[y] \in \mathbb{F}_q^\mu / V$ such that $\left| \left(\bigcup_{\ell=1}^{\mu+1} R_{x_s, i_s} \right) \cap [c] \right| > 1$. This would

mean that there are $a, b \in \bigcup_{\ell=1}^{\mu+1} R_{x_s,i_s}$ such that $a \neq b$, but $a, b \in [y]$, so $a \sim b$. As seen before, this means that $\phi(a) \sim \phi(b)$. If $\phi(a) = \phi(b)$, then by definition of ϕ, we see that $a - b \in \langle e_n \rangle$. This means that $a, b \in E_a$. Since $V \cap \langle e_i, e_j \rangle = \{0\} \,\forall\, i, j \in [\mu]$, part c of Theorem 2.56 implies that $[a] \neq [b]$, a contradiction. Thus, instead, we must have $\phi(a) \neq \phi(b)$, which we write as $\overline{b} \neq \overline{b}$. This leads to a few possibilities.

If $a, b \in E = \bigcup_{s=1}^{\mu} R_{x_s,i_s}$, then $\overline{a}, \overline{b} \in \bigcup_{s=1}^{\mu} R_{\overline{x_s},i_s}$. This would lead to a contradiction to Condition (1) for $\overline{R}$, as $\overline{a}$ and $\overline{b}$ are in the same bucket. By construction, $\phi(R_{x_{\mu+1},i_{\mu+1}}) = \{\phi(x_{\mu+1})\}$, so $a, b \in R_{x_{\mu+1},i_{\mu+1}}$ implies $\overline{a}, \overline{b} \in \phi(R_{x_{\mu+1},i_{\mu+1}}) = \{\phi(x_{\mu+1})\}$, or $\overline{a} = \overline{b} = \phi(x_{\mu+1})$, and we would have a contradiction to $\overline{a} \neq \overline{b}$.

Thus, we suppose without loss of generality that $a \in E$, and $b \in R_{x_{\mu+1},i_{\mu+1}}$. There are two possibilities. If $i_{\mu+1} = 0$, then by our selection of $i_{\mu+1}$, there is no $z \in E$ such that $x_{\mu+1} \sim z$, but $b \in R_{x_{\mu+1},0} = \{x_{\mu+1}\}$, so $b = x_{\mu+1}$, which means $a \sim b$ is a contradiction. If instead $i_{\mu+1} = \mu$, this means $\exists z \in E$ such that $z \sim x_{\mu+1}$. This means there is some $r \in [\mu]$ such that $z \in R_{x_r,i_r}$, and since $a \in E$, we also have some $s \in [\mu]$ such that $a \in R_{x_s,i_s}$.

This again leads to two possibilities. Suppose $s = r$. We see that since $b \in R_{x_{\mu+1},i_{\mu+1}}$, we have $b \in E_{x_{\mu+1}}$ and also $b \neq x_{\mu+1}$. We also have $x_{\mu+1} \in E_{x_{\mu+1}}$, so $b \neq x_{\mu+1} \implies b \not\sim x_{\mu+1}$ by Theorem 2.56. Since $z \sim x_{\mu+1}$, we must have $b \not\sim z$, or transitivity of $\sim$ would break down. Since $a \sim b$, this also means that $a \not\sim z$, so certainly $z \neq a$. Since $s = r$, we have $a, z \in R_{x_r,i_r} = R_{x_s,i_s}$, so $a - z = \alpha e_{i_r}$ for some $\alpha \in \mathbb{F}_q \setminus \{0\}$. We also have $b = x_{\mu+1} + \beta e_\mu$ for some $\beta \in \mathbb{F}_q \setminus \{0\}$, and we consider that $a \sim b$ and $z \sim x_{\mu+1}$. We may combine these as $a - z \sim b - x_{\mu+1}$, or $\alpha e_{i_r} \sim \beta e_\mu$. But this means that $\alpha e_{i_r} - \beta e_\mu \in V$, which is a contradiction to $V \cap \langle e_i, e_j \rangle = \{0\} \,\forall\, i, j \in [\mu]$ unless $\alpha e_{i_r} = \beta e_\mu$, but that would mean $i_r = \mu$, which is impossible given our construction.

Thus, we consider $s \neq r$. If $\overline{a} = \overline{z}$, then $\overline{a} \in R_{\overline{x_s},i_s}$, and $\overline{a} = \overline{z} \in R_{\overline{x_r},i_r}$, so $\overline{a} \in R_{\overline{x_r},i_r} \cap R_{\overline{x_s},i_s}$. This is a contradiction to Condition (2) for $\overline{R}$. If instead $\overline{a} \neq \overline{z}$, note that $\overline{b} = \overline{x_{\mu+1}} \sim \overline{z}$. This means that $\overline{a} \sim \overline{b} \sim \overline{z}$, so $\overline{a} \sim \overline{z}$, and this is a contradiction to Condition (1) for $\overline{R}$. Thus, we have exhausted our possibilities and must have that Condition (1) holds for R.

Next, we show that Condition (2) holds for R. Again, consider the possibilities for a contradiction. If $R_{x_r,i_r} \cap R_{x_s,i_s} \neq \emptyset$ for some $r, s \in [\mu + 1]$ such that $r \neq s$, then there are a couple of

possibilities. If $r, s \in [\mu]$, then this would mean $\exists x \in \mathbb{F}_q^\mu$ such that $x \in R_{x_r,i_r} \cap R_{x_s,i_s}$. But then $\overline{x} \in R_{\overline{x_r},j_r}$ and $\overline{x} \in R_{\overline{x_s},i_s}$, a contradiction to property (2) for $\overline{R}$. Thus, without loss of generality, we have $r \in [\mu]$ and $s = \mu + 1$. If $i_{\mu+1} = 0$, this means $R_{x_s,i_s} = R_{x_{\mu+1},0} = \{x_{\mu+1}\}$, and so the intersection must be $\{x_{\mu+1}\}$. This would mean $x_{\mu+1} \in R_{x_r,i_r}$ and so $x_{\mu+1} \in E$. Since $x_{\mu+1} \sim x_{\mu+1}$ by the reflexive property of $\sim$, it is a $z \in E$ such that $z \sim x_{\mu+1}$, a contradiction to $i_{\mu+1} = 0$.

If instead $i_{\mu+1} = \mu$, we have some $z \in E$ such that $z \sim x_{\mu+1}$, so $z \in R_{x_k,i_k}$ for some $k \in [\mu]$. Since $R_{x_r,i_r} \cap R_{x_{\mu+1},i_{\mu+1}} \neq \emptyset$, we also have some $a \in R_{x_{\mu+1},\mu}$ such that $a \in R_{x_r,i_r}$. As before, if $k = r$, we have $a - z = \alpha e_k$ for some $\alpha \in \mathbb{F}_q \setminus \{0\}$, and we have $a - x_{\mu+1} = \beta e_\mu$ for some $\beta \in \mathbb{F}_q \setminus \{0\}$. But then $a - z \sim a - x_{\mu+1}$, so $\alpha e_k \sim \beta e_\mu$, which leads to a contradiction as before.

This leaves $k \neq r$. Again, if $\overline{a} = \overline{z}$, this leads to a contradiction to (2) for $\overline{R}$, and if $\overline{a} \neq \overline{z}$, we still have $a \in R_{x_{\mu+1},\mu}$, so $\overline{a} \in \phi(R_{x_{\mu+1},\mu}) = \{\overline{x_{\mu+1}}\}$, which means $\overline{a} = \overline{x_{\mu+1}} \sim \overline{z}$, and so this contradicts Condition (1) for $\overline{R}$. Again, we have contradicted all alternatives, so Condition (2) must be satisfied for R. Because R satisfies both conditions, R is a valid query recovery set. Since this may be done for any query $Q = (x_1, \ldots, x_{\mu+1})$ of $\mu + 1$ elements in $\mathbb{F}_q^\mu$, $\mathcal{C}$ is a batch code that can satisfy $t = \mu + 1$ requests. ☐

2.5.3 Diagonal Subspace

Using the subspace $V = \langle(1, \ldots, 1)\rangle \subset \mathbb{F}_q^\mu$, we are able to generate a valid bucket configuration as long as $\mu \geq 3$:

Theorem 2.60. *If $\mu \geq 3$, then let $V = \langle(1, \ldots, 1)\rangle \subset \mathbb{F}_q^\mu$, and use the subspace construction $\mathbb{F}_q^\mu/V$ for the buckets. Then $C_Y(\rho)$ is a batch code with properties $m = q^{\mu-1}$, $\tau = 1$, and $t = \mu + 1$.*

Proof. With this construction, since $\dim(V) = 1$, we have $m = q^{\mu-1}$. We begin with the base case $\mu = 3$, where $V = \langle(1, 1, 1)\rangle$, and $t = 4$. Since $V \cap \langle e_i, e_j\rangle = \{0\}\, \forall\, i, j \in [\mu]$, recovering four copies of any one point is possible, and by Corollary 2.53, so is recovering any four points in the same bucket. Recovering four points in different buckets is trivial using all direct access. This leaves the cases where there are either 2 or 3 distinct buckets. In other words, by rearranging the points in the query and applying Corollary 2.53, we see that the only cases we need to address are, without loss

of generality, $Q = (a, a, a, b)$, $Q = (a, a, b, c)$, or $Q = (a, a, b, b)$, where $a, b, c \in \mathbb{F}_q^\mu$ such that $[a], [b]$, and $[c]$ are all distinct. We handle each of these separately:

1. Consider $Q = (a, a, a, b)$. If $[b] \notin [E_a]$, then any three recovery sets may be used for a, and b may be directly accessed. Otherwise, there is one value $i \in [\mu]$ such that $[b] \in [R_{a,i}]$, and we can satisfy the request using $R = \{R_{b,0}, R_{a,0}, R_{a,j_1}, R_{a,j_2}\}$ such that $j_1, j_2 \in [\mu] \setminus i$.

2. Recovering $Q = (a, a, b, c)$ is similar to the first case, using $R_{b,0}$ and $R_{c,0}$. Since these eliminate at most 2 recovery sets of a through intersection with $[E_a]$, there will be at least one remaining recovery set of a besides the direct access.

3. Consider $Q = (a, a, b, b)$. Utilize $R_{a,0}$ and $R_{b,0}$. If $[b] \in [R_{a,i}]$ for some $i \in [\mu]$, let $j \in [\mu] \setminus i$. Otherwise let j be any $j \in [\mu]$. This means that $[b] \notin [R_{a,j}]$ by construction. To see that we can use both $R_{a,j}$ and $R_{b,j}$, assume by way of contradiction that there exists some $[x] \in [R_{a,j}] \cap [R_{b,j}]$. Then

$$[x] = [a + \alpha e_j] = [b + \beta e_j],$$

where $\alpha, \beta \in \mathbb{F}_q \setminus \{0\}$. This means $(a + \alpha e_j) - (b + \beta e_j) = a + (\alpha - \beta)e_j - b \in V$, which in turn means $[a + (\alpha - \beta)e_j] = [b]$. Either $\alpha = \beta$, so $[a] = [b]$, a contradiction, or $[b] \in [R_{a,j}]$, also a contradiction. Therefore $B(R_{a,j}) \cap B(R_{b,j}) = \emptyset$. Thus we may use $R = \{R_{a,0}, R_{b,0}, R_{a,j}, R_{b,j}\}$.

Now, assume that for some $\mu > 3$, $\overline{V} = \langle(1, \ldots, 1)\rangle \subset \mathbb{F}_q^{\mu-1}$ generates a valid batch code with $t = \mu$. Then by Lemma 2.59, since $V = \langle(1, \ldots, 1)\rangle \in \mathbb{F}_q^\mu$ satisfies $V \cap \langle e_i, e_j\rangle = \{0\} \, \forall \, i, j \in [\mu]$, we can expand the code with buckets generated by $\mathbb{F}_q^{\mu-1}/\overline{V}$ into a code with buckets generated by $\mathbb{F}_q^\mu/V$ that can satisfy any query of $t = \mu + 1$ elements. By induction, we then have that for any $\mu \geq 3$, $V = \langle(1, \ldots, 1)\rangle \subset \mathbb{F}_q^\mu$ generates buckets for a batch code with $t = \mu + 1$. □

2.5.4 Cartesian Codes

Finally, we want to apply the techniques we have developed so far to general Cartesian codes. This results in the following theorem:

Theorem 2.61. *A Cartesian code $C_X(\rho)$ with $X = A_1 \times \cdots \times A_\mu$ of degree ρ is a batch code capable of satisfying any $\mu + 1$ requests if $\mu \geq 3$ and $\rho + 1 < |A_i| \, \forall \, i \in [\mu]$.*

Proof. Since each set A_i is a subset of $\mathbb{F}_q$, we consider any query $Q = (x_1, \ldots, x_{\mu+1})$ of points in X for which we wish to recover $f(x_1), \ldots, f(x_{\mu+1})$. Certainly, this is a query $Q' = (x_1, \ldots, x_{\mu+1})$ of points in $\mathbb{F}_q^\mu$. As long as $\mu \geq 3$, then by Theorem 2.60 we may use $V = \langle(1, \ldots, 1)\rangle$ to define buckets for $\mathbb{F}_q^\mu$, and the query Q' may be satisfied with the solution set

$$\{R'_{x_1,i_1}, \ldots, R'_{x_{\mu+1},i_{\mu+1}}\},$$

with R'_{x_s,i_s} as in Definition 2.47.

For all $s \in [\mu+1]$, we let $R_{x_s,i_s} = R'_{x_s,i_s} \cap X$ and see that R_{x_s,i_s} matches the definition in Lemma 2.46. Since $\rho + 1 < |A_i|$ for all $i \in [\mu]$, by that lemma, the values of $f(R_{x_i})$ are enough to recover $f(a)$, for any $f \in S_{\leq\rho}$ and any $i \in [\mu]$. This means that

$$\{R_{x_1,i_1}, \ldots, R_{x_{\mu+1},i_{\mu+1}}\}$$

is a query recovery set for Q in X. We need only to show that the two conditions are still satisfied. First, we need to define the bucket structure for $C_X(\rho)$.

For any $c \in X$, let $B_c = [c] \cap X$, with the equivalence relation $\sim$ as before ($a \sim b$ if and only if $a - b \in V$). Just as $\sim$ partitions $\mathbb{F}_q^\mu$ into buckets, this equivalence relation partitions X into buckets. If $\exists c \in X$ such that $\left|\left(\bigcup_{s=1}^{\mu+1} R_{x_s,i_s}\right) \cap B_b\right| > 1$, then there must be $a, b \in \bigcup_{s=1}^{\mu+1} R_{x_s,i_s}$ such that $a \sim b$, but $a \neq b$. This means

$$\begin{aligned} a, b \in & \bigcup_{s=1}^{\mu+1} R_{x_s,i_s} \\ &= \bigcup_{s=1}^{\mu+1} \left(R'_{x_s,i_s} \cap X\right) \\ &= \left(\bigcup_{s=1}^{\mu+1} R'_{x_s,i_s}\right) \cap X. \end{aligned}$$

But then we would have $a \sim b \in \bigcup_{s=1}^{\mu+1} R'_{x_s,i_s}$, a contradiction to Condition (1) for R'. Similarly, if there are some $r, s \in [\mu+1]$ such that $R_{x_r,i_r} \cap R_{x_s,i_s} \neq \emptyset$, then we have some $a \in R_{x_r,i_r} \cap R_{x_s,i_s}$, but then necessarily $a \in R'_{x_r,i_r}$ and $a \in R'_{x_s,i_s}$, a contradiction to Condition (2) of R'. Thus, both conditions are satisfied. Since this may be done for any query Q of $n+1$ points in $C_X(\rho)$, $C_X(\rho)$ is

a batch code capable of recovering any query of $t = \mu + 1$ entries. $\square$

2.6 Conclusions

The work in this chapter focused on batch properties of binary Hamming and Reed-Muller codes, as well as general Cartesian codes.

The high locality of binary Hamming codes implies their availability to be at most 1. Binary Hamming codes can be viewed as linear batch codes retrieving queries of at most 2 indices, the trivial case. Nonetheless, we proved that for $t = 2$, binary Hamming codes are actually optimal $[2^{s-1}, 2^s - 1 - s, 2, m, \tau]$ batch codes for $m, \tau \in \mathbb{N}$ such that $m\tau = 2^{s-1}$.

We turned to binary Reed-Muller codes for optimal batch codes that allow larger queries, meaning t-tuples with $t > 2$. This research direction was motivated by the large availability of first-order Reed-Muller codes as seen previously. We proved the optimality of first-order binary Reed-Muller codes for $t = 4$. We then generalized our study to Reed-Muller codes $\mathcal{RM}(\rho, \mu)$ which have order less than half their length by proving that they have batch properties $(2^\mu, k, 4, m, \tau)$ such that $m\tau = 10 \cdot 2^{2\rho-2}$ for $\mathcal{RM}(\rho', \mu)$ where $\mu \in \{2\rho + 2, 2\rho + 3\}$ and $\rho' \leq \rho$.

Finally, for Cartesian codes, we defined a bucket construction where each bucket is determined by a translation of the elements of V for some subspace $V \subset \mathbb{F}_q^\mu$. We proved that for a valid construction the minimum distance of V must be ≥ 3. With $V = \langle(1, \ldots, 1)\rangle$, we showed that a Cartesian code over $\mathbb{F}_q^\mu$ can satisfy queries of length $t = \mu + 1$ for any $\mu \geq 3$. We also generalized this result for any Cartesian code, provided that the degree ρ of polynomials considered is sufficiently small.

Chapter 3

De Bruijn Sequences

3.1 Introduction

De Bruijn sequences are cyclic sequences over an alphabet of size k which contain each word of length n exactly once per cycle. They are named after Nicolaas Govert de Bruijn, who proved the conjecture that there are $2^{2^{n-1}-n}$ such binary sequences of order n [de Bruijn, 1946]. As later acknowledged by de Bruijn [de Bruijn, 1975], they were first characterized (in the binary case) by C. Flye Sainte-Marie in 1894. The count in the k-ary case was proven to be $(k!)^{k^{n-1}}k^{-n}$ by van Aardenne-Ehrenfest and de Bruijn in 1951 [van Aardenne-Ehrenfest and de Bruijn, 1951].

These sequences correspond to maximum length cycles generated by Feedback Shift Registers (FSRs)[Lempel, 1970]. Depending upon the feedback function used in an FSR, it may be implemented using gates in circuitry. A maximum-length cycle in binary may then be used to implement a stream cipher [Babbage and Dodd, 2008, Canniere and Preneel, 2006, Hell et al., 2008, Hell et al., 2007].

In the case of Linear FSRs (LFSRs), the maximum length is $2^n - 1$. Each such cycle is missing only the word of all 0s, and so may be converted to a de Bruijn sequence by the addition of a 0 in the proper place (adding one nonlinear term). LFSRs that generate maximum-length sequences (m-sequences) may be characterized by a primitive polynomial in $\mathbb{F}_2^n[x]$. There are $\frac{\phi(2^n-1)}{n}$ primitive

polynomials over $\mathbb{F}_2$, and so this method of extending an LFSR sequence generates a subset of all de Bruijn sequences of that size [Lidl and Niederreiter, 1983]. In the general case of Nonlinear Feedback Shift Registers (NLFSRs) which generate maximum length sequences, no such characterization is known.

Several algorithms for generating these sequences are surveyed in [Fredricksen, 1982], among others [Ford, 1957, Eldert et al., 1958, Leach, 1960]. One method views such sequences as paths on the de Bruijn graph of order n, and seeks to join smaller cycles of the graph into a larger cycle [Yoeli, 1963]. Two constructions of this form appear in [Etzion and Lempel, 1984]. In addition, Abraham Lempel introduced the D-morphism, a homomorphism from the graph of order n to the graph of order $n-1$ [Lempel, 1970]. Various properties of this homomorphism are proven in [Akinwande, 2010]. Under it, the preimages of a de Bruijn cycle of order $n-1$ are two disjoint cycles which may be joined to create a de Bruijn sequence of order n. The complexity of this method is examined in [Mandal and Gong, 2016], and an efficient implementation is given in [Yang et al., 2017]. Applying this method over k orders is examined in [Mandal and Gong, 2013]. This method has also been generalized to the k-ary case in [Alhakim and Akinwande, 2011], which also demonstrates how exponentially many binary de Bruijn sequences may be generated.

In this chapter, we show how to efficiently determine whether a companion pair lies on two separate cycles given knowledge of the recursive structure. This begins with Theorem 3.47 providing a way to recursively locate a given word. As part of this calculation, we need Theorem 3.60 to determine sums of subsequences. By careful accounting of expanded terms, we show in Lemma 3.67 that the time complexity of this operation is $\mathcal{O}\left(n^4 \log(n)\right)$, and so as a corollary we have that the index function has complexity $\mathcal{O}\left(n^5 \log(n)\right)$. Finally, Theorem 3.69 gives the total complexity of the construction step when a valid toggle is chosen randomly. This is followed by results on the proportion of valid toggles, which may lead to $\mathcal{O}\left(n^6 \log(n)\right)$ total complexity.

Section 3.2 introduces the background and notation necessary to build up recursive sums. Some preliminary results achieved in the examination of homomorphic preimage constructions are introduced in Section 3.3. Then, in Section 3.4, we build up the notion of the indexing function. This is followed by Section 3.5, which works up to a formula for recursively expanding terms of iterated sums. The complexities of these operations are analyzed in Section 3.6, followed by a summary of

the results so far in Section 3.7.

3.2 Background and Notation

In the major results of this chapter, we examine binary sequences with a special property and how to construct them. That is, we build sequences from the binary group $B = \mathbb{Z}/2\mathbb{Z} = \{0, 1\}$. In the more general form, we may instead consider sequences over an alphabet of size q. That is, we use the group of integers modulo q, $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z} = \{0, 1, \dots, q-1\}$. From this, we may define the de Bruijn graph of order n.

Definition 3.1. *The* de Bruijn graph of order n *(over an alphabet of size q) is the directed graph $G_{n,q}$ with vertices $\mathbb{Z}_q^n$, and an edge (x, y) for $x = (x_1, \dots, x_n), y = (y_1, \dots, y_n) \in \mathbb{Z}_q^n$ if and only if*

$$(x_2, \dots, x_{n-1}, x_n) = (y_1, y_2 \dots, y_{n-1}). \tag{3.1}$$

In the commonly considered binary case, we shall omit the 2 and write G_n. Using this definition, it is easy to then build to the definition of a de Bruijn sequence.

Definition 3.2. *A* de Bruijn sequence of order n *(over an alphabet of size q) is a vertex-disjoint cycle of $G_{n,q}$ with length q^n.*

That is, a de Bruijn sequence must contain every word of length n over the alphabet $\mathbb{Z}_q$ exactly once. As noted in [Lempel, 1970], if $(x^{(1)}, x^{(2)}, \dots, x^{(2^n)})$ is a de Bruijn sequence of order n, we may represent it by sequence $c_1 c_2 \dots c_{2^n}$, where $c_i = x_1^{(i)}$ for $1 \leq i \leq q^n$. This follows because an edge exists only if (3.1) is satisfied. It is this more compact notation that we use throughout the paper for ease of understanding. In fact, for a general sequence of length ℓ, we use the notation $c_1 \dots c_\ell$.

Using the group structure of B, we define the following:

Definition 3.3. *The* Boolean complement *of $\alpha \in B$ is $\overline{\alpha} = \alpha + 1$.*

That is, we have $\overline{0} = 0 + 1 = 1$, and $\overline{1} = 1 + 1 = 0$. Using this complement, for any

$x = (x_1, \ldots, x_n) \in B^n$, we may define three important related words:

Definition 3.4. *The* complement *of x is*

$$\overline{x} = (\overline{x_1}, \ldots, \overline{x_n}).$$

Similarly, we note that for a cycle $C = c_1c_2 \ldots c_\ell$, we have $\overline{C} = \overline{c_1 c_2} \ldots \overline{c_\ell}$, which we call the dual *of C. The* conjugate *of x is*

$$\hat{x} = (\overline{x_1}, x_2, \ldots, , x_{n-1}, x_n),$$

and the companion *of x is*

$$x' = (x_1, x_2, \ldots, , x_{n-1}, \overline{x_n}).$$

For structural reasons, we primarily examine the companions of words rather than conjugates in this paper. Traditionally, the definition of *adjacent* sequences is in terms of conjugates.

Definition 3.5. *Two disjoint cycles C_1 and C_2 in the de Bruijn graph G_n are said to be* adjacent *if there exists some $x \in B^n$ such that x is in C_1 and $\hat{x}$ is in C_2.*

If (x, y) is the edge from x in C_1, then $(\hat{x}, y')$ must be an edge in C_2, and thus it is equivalent to define two disjoint cycles as adjacent if there exists $y \in B^n$ such that $y \in C_1$ and $y' \in C_2$. From [Yoeli, 1963], we get the following lemma:

Lemma 3.6. *If C_1 and C_2 are adjacent, with x and y as above, then by replacing the edges (x, y) and $(\hat{x}, y')$ with the edges (x, y') and $(\hat{x}, y)$, the resulting subgraph of G_n is one cycle.*

Again, it can be seen that all that matters is the existence of those edges. This motivates our definition of a toggle point:

Definition 3.7. *A* toggle point *t of order n is a word*

$$t = (t_1, \ldots, t_{n-1}) \in B^{n-1}.$$

In particular, we are interested in when such a toggle point allows us to join two disjoint cycles.

Definition 3.8. *A toggle point $t = (t_1, \dots, t_{n-1})$ is* valid *for cycles C_1 and C_2 if they are adjacent with y in C_1 and y' in C_2, where $y = (t_1, \dots, t_{n-1}, y_n)$ for some $y_n \in B$.*

Thus, if a valid toggle point t for C_1 and C_2 exists, we may join the two cycles by "toggling" the edges containing t. In the following sections, we are concerned with identifying toggles in the situation where C_1 and C_2 are complements that together cover every vertex of G_n.

3.2.1 The D-morphism

To find such cycles, we first must introduce the D-morphism from [Lempel, 1970]:

Definition 3.9. *The* Lempel D-morphism *is a function $D : B^n \to B^{n-1}$ such that, for $x = (x_1, x_2, \dots, x_n) \in B^n$,*

$$D(x) = (x_1 + x_2, x_2 + x_3, \dots, x_{n-1} + x_n) \in B^{n-1}. \tag{3.2}$$

As Lempel showed, this is actually a homomorphism from G_n to G_{n-1}. To proceed further, we need the notion of the weight of a cycle, and of primitive cycles.

Definition 3.10. *The* weight *of a cycle $C = c_1 c_2 \dots c_\ell$ is*

$$w(C) = \sum_{i=1}^{\ell} c_i.$$

Definition 3.11. *A cycle C is* primitive *if it is vertex disjoint from its dual $\overline{C}$.*

From [Lempel, 1970], we have a central theorem which allows us to construct de Bruijn sequences.

Theorem 3.12. *An ℓ-cycle Γ in G_{n-1} is the D-morphic image of a primitive ℓ-cycle C in G_{n-1} if and only if $w(\Gamma) = 0$*

That is, if a cycle Γ of G_{n-1} has an even number of ones, then it has a preimage of the same length in G_n. Moreover, it is easily verified that if $D(C) = \Gamma$, then $D(\overline{C}) = \Gamma$ as well.

At this point, we introduce notation for these preimages.

Definition 3.13. *If C is a cycle of G_n such that $w(C) = 0$, then ${}^0C$ and ${}^1C$ are the sequences beginning with* 0 *and* 1 *respectively such that $D({}^0C) = D({}^1C) = C$.*

Since ${}^0C$ and ${}^1C$ are primitive, they cannot share a vertex by definition, so they are disjoint. With the easily observed fact that $w(r) = 0$ for any de Bruijn sequence r in G_{n-1}, we see that the D-morphic preimages ${}^0r$ and ${}^1r$ are each primitive and of length 2^{n-1}. Since they are also disjoint, this means that together they cover all of the vertices of G_n. Thus, we need only find a valid toggle point for ${}^0r$ and ${}^1r$, and the resulting joined cycle is of length 2^n, making it a de Bruijn sequence.

In fact, we prove this in an alternate manner, but first we must introduce two important functions:

Definition 3.14. *For a de Bruijn sequence s represented as $(s^{(1)}, s^{(2)}, \ldots, s^{(2^n)})$ in G_n and any word $c \in B^n$, the* index function *I_s is defined by $I_s(c) = \ell$ such that $s^{(\ell)} = c$.*

That is, the function gives the location in the de Bruijn sequence of a given word. This is well-defined because a de Bruijn sequence s must contain every word $c \in B^n$ exactly once. We now give a function that is almost the inverse of I, in a sense. However, instead of giving the full word at a given index, it only gives us the initial bit.

Definition 3.15. *For a de Bruijn sequence s represented in the compact notation as $s_1 s_2 \ldots s_{2^n}$, and any $1 \le \ell \le 2^n$, the* symbol *function $S^{(0)}(s)$ is defined by*

$$S_\ell^{(0)}(s) = s_\ell.$$

This function instead gives us the value of a bit in the more compact representation of the de Bruijn sequence s. From this, we introduce the recursive function $S_\ell^{(m)}(s)$ which we shall use to evaluate $S^{(0)}$ efficiently.

Definition 3.16. *The* recursive sum *function $S_\ell^{(m)}(s)$ for $m \ge 1$ is defined by*

$$S_\ell^{(m)}(s) = \sum_{i=1}^{\ell} S_i^{(m-1)}(s).$$

We also need some notation to for subsequences and some sequences which appear frequently.

Definition 3.17. *A* subsequence s_a^b *for* $a \leq b \in \mathbb{N}$ *is defined as*

$$s_a^b = s_a s_{a+1} \dots s_{b-1} s_b.$$

In the case where $a = b$, we have a single $s_a^b = s_a \in B$, and in the case where $b < a$, we let s_a^b represent the empty sequence. To avoid confusion, when concatenating sequences using this notation (and especially using the notation ${}^0r$ and ${}^1r$), we separate the subsequences with "|".

Definition 3.18. *The* concatenation *of two subsequences* s_a^b *and* r_c^d *is denoted*

$$s_a^b | r_c^d = s_a s_{a+1} \dots s_{b-1} s_b r_c r_{c+1} \dots r_{d-1} r_d.$$

We also frequently make use of subsequences of the following two infinite sequences:

Definition 3.19. *The* initial one *sequence* $\underline{1}$ *is defined by* $\underline{1}_1 = 1$ *and* $\underline{1}_i = 0$ *for* $i > 1$*. The* all ones *sequence* 1 *is defined by* $1_i = 1$ *for* $i \geq 1$*.*

That is, we have

$$\underline{1} = 100\dots$$

$$1 = 111\dots.$$

With the necessary notation in place, we now cover some preliminary results relating to de Bruijn graphs

3.3 Preliminary Results

In this section, we cover two related results achieved along the way to determining the validity of toggle points. These are the lift structure given by D-morphisms and a matrix representation of cycles on de Bruijn graphs.

3.3.1 D-Homomorphisms are covering maps

First, we aim to show that the homomorphisms used in [Alhakim and Nouiehed, 2017] make $G_{n+k,q}$ a lift, or covering graph of $G_{n,q}$.

We start with a result for homomorphisms in general from [Akinwande, 2010].

Lemma 3.20. *Any graph homomorphism H from $G_{n+k,q}$ to $G_{n,q}$ is of the form*

$$H(x_1, \ldots, x_{n+k}) = (h(x_1, \ldots, x_{k+1}), \ldots, h(x_n, \ldots, x_{n+k})),$$

for some function h from $\mathbb{Z}_q^{k+1}$ to $\mathbb{Z}_q$.

We note that the D-morphism defined before may be written in this form, with the function $h(x_1, x_2) = x_1 + x_2$. At this point, we introduce some notation to make explanation of desired properties of h (and thus H) easier.

Definition 3.21. *For a function $h : \mathbb{Z}_q^{k+1} \to \mathbb{Z}_q$, and for any $(a_1, \ldots, a_k) \in \mathbb{Z}_q^k$, let the* last function *and the* first function *of h (at $(a_1, \ldots, a_k)$) be respectively defined as*

$$h(x)_{(a_1,\ldots,a_k)} = h(x, a_1, \ldots, a_k)$$
$$h_{(a_1,\ldots,a_k)}(x) = h(a_1, \ldots, a_k, x)$$

That is, these are the functions where all but the first (respectively, last) elements are fixed, with only the first (respectively, last) varying. With this notation in place, we give the definition of Property (D) from [Alhakim and Nouiehed, 2017]

Definition 3.22. *A homomorphism H is said to have* Property (D) *if each vertex disjoint path in $G_{n,k}$ is the image of q^k non-overlapping vertex disjoint paths in $G_{n+k,q}$.*

We have seen that the D-morphism has Property (D). The proof in [Alhakim and Nouiehed, 2017] relies on the following from [Akinwande, 2010], which is rephrased in our new notation.

Lemma 3.23. *A homomorphism H has Property (D) if and only if for any $(a_1, \ldots, a_k) \in \mathbb{Z}_q^k$, the functions $h(x)_{(a_1,\ldots,a_k)}$ and $h_{(a_1,\ldots,a_k)}(x)$ are bijections.*

This is a more formal way of saying that the function h from which H is constructed must be one-to-one when all but the first or last variables are considered fixed. For instance, the $h(x_1, x_2) = x_1 + x_2$ for the D-morphism is a one-to-one function of x_1 and x_2 when the other variable is considered to be fixed. We next introduce the definition of a lift for a digraph from [Makelov, 2015]. This is built up from the definition of a homomorphism for a digraph.

Definition 3.24. *A* digraph homomorphism $f : G \to H$ *(where G has vertices V_G and edges E_G and H has vertices V_H and edges E_H) is a pair of maps $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ such that if the edge $e = (u, v) \in E_G$, then the edge $f_E(e) = (f_V(u), f_V(v)) \in E_H$.*

With this definition in place, we may now more succinctly define a covering map or lift:

Definition 3.25. *A* covering map $p : G \to H$ *of* digraphs *G and H as above is an edge-surjective homomorphism such that for every $v \in V_G$, the set of edges with tail (respectively, head) v is mapped bijectively to the set of edges with tail (respectively, head) $p_V(v)$. That is, the restrictions*

$$p_E : \{(v, x) \in E_G\} \to \{(p_V(v), p_V(x)) \in E_H\}$$
$$p_E : \{(x, v) \in E_G\} \to \{(p_V(x), p_V(v)) \in E_H\}$$

are both bijections. We say that G is a lift *of H.*

We use these definitions to prove the connection between homomorphisms with Property (D) and lifts of de Bruijn digraphs. First, we start with an important lemma which allows us to see that such homomorphisms are edge-surjective.

Lemma 3.26. *For a homomorphism $H : G_{n+k,q} \to G_{n,q}$ with Property (D), any $(a_1, \ldots, a_k) \in \mathbb{Z}_q^k$, and $(b_1, \ldots, b_n) \in \mathbb{Z}_q^n$, there exists $(a_1, \ldots, a_{n+k})$ such that $H(a_1, \ldots, a_{n+k}) = (b_1, \ldots, b_n)$.*

Proof. Consider any homomorphism $H : G_{n+k,q} \to G_{n,q}$ with Property (D). This means that

$$\begin{aligned} H(a_1, \ldots, a_{n+k}) &= (h(a_1, \ldots, a_{k+1}), \ldots, h(a_n, \ldots, a_{n+k})) \\ &= (h_{(a_1,\ldots,a_k)}(a_{k+1}), \ldots, h_{(a_n,\ldots,a_{n+k-1})}(a_{n+k})). \end{aligned}$$

Since $h_{(a_1,\ldots,a_k)}(a_{k+1})$ is a bijection by Lemma 3.23, there exists some a_{k+1} such that the function

evaluates to b_1. We may then move to the next entry, $h_{(a_2,\ldots,a_{k+1})}(a_{k+2})$, and find that there exists some a_{k+1} such that this evaluates to b_2. This may be repeated until a_{n+k} is reached and the last entry evaluates to b_n. Thus, we have $H(a_1,\ldots,a_{n+k}) = (b_1,\ldots,b_n)$. $\square$

Note that this is a generalization of Definition 3.13.

Corollary 3.27. *Any homomorphism $H : G_{n+k,q} \to G_{n,q}$ is vertex-surjective.*

This can be seen by simply considering $(a_1,\ldots,a_k) = (0,\ldots,0)$, or in fact any beginning words of length k.

Theorem 3.28. *$G_{n+k,q}$ is a lift of $G_{n,q}$, with the covering map being any homomorphism $H : G_{n+k,q} \to G_{n,q}$ with Property (D).*

Proof. We have just seen that any such H is vertex-surjective. To confirm that H is edge-surjective, consider any edge

$$(b_1,\ldots,b_n) \to (b_2,\ldots,b_{n+1}).$$

We already know that there is some $(a_1,\ldots,a_{n+k})$ such that $H(a_1,\ldots,a_{n+k}) = (b_1,\ldots,b_n)$, but we also know that, given $a_{n+1},\ldots,a_{n+k}$, we may find a_{n+k+1} such that $h(a_{n+1},\ldots,a_{n+k+1}) = b_{n+1}$, and so we will have that $H(a_2,\ldots,a_{n+k+1}) = (b_2,\ldots,b_{n+1})$, which means that the edge

$$(a_1,\ldots,a_{n+k}) \to (a_2,\ldots,a_{n+k+1})$$

is mapped to

$$(b_1,\ldots,b_n) \to (b_2,\ldots,b_{n+1}),$$

as desired. This can be done for any edge in $G_{n,q}$, and so H is edge-surjective.

Finally, for every $v = (a_1,\ldots,a_{n+k}) \in V_{G_{n+k,q}}$, we have that the edges with tail v are

$$(a_1,\ldots,a_{n+k}) \to (a_2,\ldots,a_{n+k},c),$$

for any $c \in \mathbb{Z}_q$, while vertex $H(v)$ is the tail of edges

$$(b_1, \ldots, b_{n+k}) \to (b_2, \ldots, b_{n+k}, d),$$

for any $d \in \mathbb{Z}_q$. We note that

$$\begin{aligned} H(a_2, \ldots, a_{n+k}, c) &= (h(a_2, \ldots, a_{k+2}, \ldots, h(a_n, \ldots, a_{n+k}), h(a_{n+1}, \ldots, a_{n+k}, c)) \\ &= (b_2, \ldots, b_n, h(a_{n+1}, \ldots, a_{n+k}, c)). \end{aligned}$$

Since $h_{(a_{n+1}, \ldots, a_{n+k}}(c)$ is a bijection, this last entry will bijectively cover $(b_2, \ldots, b_{n+k}, d)$, and so there is a bijection between edges with tail v and edges with tail $H(v)$. A similar process based on a flipped version of Lemma 3.26 can be done to see that there is a bijection between edges with head v and edges with head $H(v)$, and so we find that H is a covering map, making $G_{n+k,q}$ a lift of $G_{n,q}$. □

3.3.2 Counting of Multi-level Homomorphisms

We now briefly consider the number of possible homomorphisms with Property (D).

Lemma 3.29. *The number of distinct homomorphisms with property (D) which make $G_{n+k,q}$ a lift of $G_{n,q}$ is*

$$L_q^{q^{k-1}},$$

where L_q is the number of $q \times q$ distinct Latin squares.

Proof. By Lemma 3.20, any graph homomorphism H is built from a function h, and by Lemma 3.23, H has Property (D) if and only if h is one-to-one with respect to the first and last variable when all other variables are held constant. For $H : G_{n+k,q} \to G_{k,q}$, h is a function of $1+k$ variables. We may consider a $q \times q$ table of outputs when the middle $k-1$ variables are held constant. For any given values of the middle $k-1$ variables, the entry in row i and column j indicates the output when the first variable has value i and the last has value j. It is clear that for this function to be one-to-one in the described manner, each row and column of this table must have exactly one copy of each of

the q symbols. This precisely matches the definition of a Latin square, and so the number of such possible tables is L_q, the number of $q \times q$ Latin squares.

Since a different such square may be selected for each of the q^{k-1} different selections of middle variables, there are $L_q^{q^{k-1}}$ distinct possible functions h. We claim these all result in distinct homomorphisms H. If two functions h and h' differ in any given entry, then the corresponding inputs necessarily result in different outputs, and so a tuple that contains the given input sequence will result in two different homomorphic images. Thus, the number of distinct homomorphisms $H : G_{n+k,q} \to G_{n,q}$ with Property (D) is $L_q^{q^{k-1}}$. $\square$

There is a formula for L_q given in [Shao and Wei, 1992]. At this juncture, however, it is only important to note that there is a lower bound [Ryser, 1963]. This leads us to one more conclusion about the number of homomorphisms.

Corollary 3.30. *There exist homomorphisms $H : G_{n+k,q} \to G_{n,q}$ with property (D) that are not the composition of k homomorphisms between the graphs of intermediate order.*

Proof. If we consider only compositions of k functions of two variables to map from sequences of $k+1$ symbols down to 1, then for each of the k functions, there are L_q choices, which results in at most L_q^k choices. Without knowing how to compute L_q, it is trivial to see that so long as L_q is positive, there exists some k such that $L_q^k < L_q^{q^{k-1}}$. Thus, there exist homomorphisms between graphs of order that differ by k that cannot be formed by k compositions of homomorphisms between graphs which differ in order by 1. $\square$

3.3.3 Matrix Representation

We now turn to a form of representation that was devised to help study locations of potential edges to toggle. We start with the definition of an adjacency matrix which we shall use in this section.

Definition 3.31. *For a digraph G with vertices V_G and edges E_G, the adjacency matrix A_G is a $|V_G| \times |V_G|$ matrix where $A_{i,j} = 1$ if $(v_i, v_j) \in E_G$ and $A_{i,j} = 0$ otherwise.*

We note that this relies on there being an order of the vertices, and so we define an order

for the vertices $V_{G_{n,q}}$ depending on homomorphisms with Property (D).

Definition 3.32. *Let $H_1, \ldots, H_{n-1}$, where $H_i : G_{i+1,q} \to G_{i,q}$ be homomorphisms with Property (D). Then let $C_1 : \mathbb{Z}_q \to \{0, 1, \ldots, q-1\} (\subset \mathbb{Z})$ be defined by $C_1(a) = a$ (note the change of domain from the group $\mathbb{Z}_q$ to a subset of the group $\mathbb{Z}$). Then for each $i > 1$, let the function $C_i : \mathbb{Z}_q^i \to \{0, 1, \ldots, q^i - 1\} (\subset \mathbb{Z})$ be defined by*

$$C_i(a_1, \ldots, a_i) = C_{i-1}(H_{i-1}(a_1, \ldots, a_i)) \cdot q + C_1(a_1).$$

We now claim that these functions are bijective, so that they may be used to define an ordering:

Lemma 3.33. *For $1 \leq i \leq n$, the function $C_i : \mathbb{Z}_q^i \to \{0, 1, \ldots, q^i - 1\}$ as defined above is bijective.*

Proof. We will prove this inductively. We begin with the base case $i = 1$. The function C_1 is injective since $C_1(a) = C_1(a')$ implies $a = a'$. It is also surjective since for $a \in \{0, \ldots, q-1\}]$, $C_1(a) = a$.

For our induction hypothesis, assume that C_i is a bijection for some $1 \leq i \leq n-1$. Because the domain $\mathbb{Z}_q^{i+1}$ and the codomain $\{0, 1, \ldots, q^{i+1} - 1\}$ have the same cardinality, C_{i+1} is a bijection if and only if it is surjective. Consider any $d \in \{0, 1, \ldots, q^{i+1} - 1\}$, and let $b, a_1 \in \mathbb{Z}$ be the quotient and remainder of d divided by q. That is, $d = bq + a_1$. We certainly have that $b \in \{0, 1, \ldots, q^i - 1\}$, and since C_i is a bijection, it is surjective, so there is some $(b_1, \ldots, b_i) \in \mathbb{Z}^q$ such that $C_i(b_1, \ldots, b_i) = b$. We then note that by Lemma 3.26, we may start with any a_1 and construct an $(a_1, \ldots, a_{i+1})$ such that $H_i(a_1, \ldots, a_{i+1}) = (b_1, \ldots, b_i)$. We then write

$$\begin{aligned} C_{i+1}(a_1, \ldots, a_{i+1}) =& C_i(H_i(a_1, \ldots, a_{i+1})) \cdot q + C_1(a_1) \\ =& C_i(b_1, \ldots, b_i) \cdot q + a_1 \\ =& bq + a_1 \\ =& d, \end{aligned}$$

and since this may be done for any $d \in \{0, \ldots, q^{i+1} - 1\}$, C_{i+1} is surjective, and therefore a bijection. By induction, this holds for all $1 \leq i \leq n$. □

From this, we simply define $v_i \in V_{Gn,q}$ to be the unique vertex $v \in \mathbb{Z}_q^n$ such that $C_n(v) = i$. Note that in this case $i \in \{0, \ldots, q^n\}$. For this convenient indexing, we will have to start the indices of all matrix entries at 0.

We now introduce the Dirac notation, widely used in quantum mechanics to represent quantum states, which will be useful to compactly specify this representation:

Definition 3.34. *For any $0 \le c \le q^n - 1$, let*

$$\langle c| = e_c = \begin{bmatrix} a_0 & a_1 & \ldots & a_{q^n-1} \end{bmatrix}$$

$$|c\rangle = e_c^T = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{q^n-1} \end{bmatrix},$$

where $a_c = 1$ and $a_j = 0$ for all $j \neq c$.

Again note that we index starting at 0 so that $\langle C_n(0,0,\ldots,0)| = e_0 = [10\ldots0]$, as desired, rather than having to add 1 to the index. This means that $\langle C(v_i)| = e_i$, and similarly, $|C(v_i)\rangle = e_i^T$. We now construct a matrix to represent the behavior of h_i.

Definition 3.35. *For $1 \le i \le n-1$, let*

$$H_{i,1} = \sum_{b \in \mathbb{Z}_q} \left[|C_1(b)\rangle \langle C_1(b)| \otimes \left(\sum_{a \in \mathbb{Z}_q}^{k} |C_1(a)\rangle \left\langle C_1(h_{i\,(a)}^{-1}(b)) \right| \right) \right].$$

This works because for each i, $h_i(a_1, a_2)$ must be a bijection in terms of the last variable when the first is held fixed. That is, $h_{i(a_1)}(a_2)$ is a bijection. If $h_{i(a)}(a') = b$, then $h_{i(a)}^{-1}(b) = a'$. Intuitively, the structure of this matrix is such that in the submatrix corresponding to b along the diagonal, the row corresponding to a has a 1 in the entry for a' such that $h_i(a, a') = b$. Because $h_{i(a)}(a')$ is a bijection, this makes $H_{i,1}$ a permutation matrix. We formalize this in the following:

Lemma 3.36. *For any $0 \le c \le q^2 - 1$, if c_1c_2 is the q-ary representation of c, then $\langle c| H_{i,1} = \left\langle c_1 h_{i\,(c_2)}^{-1}(c_1) \right|$.*

Proof. We note that

$$
\begin{aligned}
\langle c| H_{i,1} &= \langle c_1c_2| H_{i,1} = \langle c_1| \otimes \langle c_2| H_{i,1} \\
&= \langle c_1| \otimes \langle c_2| \sum_{b\in\mathbb{Z}_q} \left[|C_1(b)\rangle \langle C_1(b)| \otimes \left(\sum_{a\in\mathbb{Z}_q} |C_1(a)\rangle \left\langle C_1(h^{-1}_{i\,(a)}(b)) \right| \right) \right] \\
&= \langle c_1| \otimes \langle c_2| \sum_{b\in\{0,\ldots,q-1\}} \left[|b\rangle \langle b| \otimes \left(\sum_{a\in\{0,\ldots,q-1\}}^{k} |a\rangle \left\langle h^{-1}_{i\,(a)}(b) \right| \right) \right] \\
&= \sum_{b=0}^{q-1} \left[\langle c_1| \, |b\rangle \langle b| \otimes \left(\langle c_2| \sum_{a=0}^{q-1} |a\rangle \left\langle h^{-1}_{i\,(a)}(b) \right| \right) \right] \\
&= \langle c_1| \otimes \left(\sum_{a=0}^{q-1} \langle c_2| \, |a\rangle \left\langle h^{-1}_{i\,(a)}(c_1) \right| \right) = \langle c_1| \otimes \left\langle h^{-1}_{i\,(c_2)}(c_1) \right| \\
&= \left\langle c_1 h^{-1}_{i\,(c_2)}(c_1) \right|,
\end{aligned}
$$

where $c_1 h^{-1}_{i\,(c_2)}(c_1)$ is understood to be a q-ary number. That is, $c_1 h^{-1}_{i\,(c_2)}(c_1) = c_1 q + h^{-1}_{i\,(c_2)}(c_1)$. This follows because $\langle c_1| \, |b\rangle = 1$ when $b = c_1$ and 0 otherwise, and likewise for a. ☐

Thus, when selecting row c_1c_2 of a product $H_{i,1}A$ (left multiplication by $\langle c|$), this corresponds to selecting row $c_1 h^{-1}_{i\,(c_2)}(c_1)$ of A. Since $h^{-1}_{i\,(a)}(b)$ is a bijection, this mapping is a bijection, and so $H_{i,1}$ is a permutation matrix.

In the binary case, we always use the same homomorphism H, letting $h(0,0) = 0$, $h(0,1) = 1$, $h(1,0) = 1$, and $h(1,1) = 0$. This makes the matrix

$$
H_{1,1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.
$$

Next, we build from this to a matrix representing the action of H_i.

Definition 3.37. *For $1 < j \leq i$, let*

$$
H_{i,j} = I_q \otimes H_{i,j-1}.
$$

Note that, since $H_{i,1}$ is a block diagonal matrix with blocks of size q, by construction, each $H_{i,j}$ is a block diagonal matrix with blocks of size q. Furthermore, because each of those blocks is individually a permutation matrix, the entire matrix $H_{i,j}$ is a permutation matrix. We make this more concrete in the following.

Lemma 3.38. *For any $1 \le i \le n-1$, any $1 \le j \le i$, and any $0 \le c \le q^{j+1}-1$, with $c = c_1 \dots c_j c_{j+1}$,*

$$\langle c_1 \dots c_j c_{j+1} | \, H_{i,j} = \left\langle c_1 \dots c_j h^{-1}_{i(c_{j+1})}(c_j) \right| .$$

Proof. We prove this inductively. For any $1 \le i \le n-1$, Lemma 3.36 provides the base case where $j = 1$. Then, suppose this holds for some $1 \le j < i$. We write

$$\begin{aligned}\langle c_1 \dots c_{j+2} | \, H_{i,j+1} &= \langle c_1 \dots c_{j+1} c_{j+2} | \, H_{i,j+1} = (\langle c_1 | \otimes \langle c_2 \dots c_{j+1} c_{j+2} |) \, (I_q \otimes H_{i,j}) \\ &= (\langle c_1 | \, I_q) \otimes (\langle c_2 \dots c_{j+1} c_{j+2} | \, H_{i,j}) = \langle c_1 | \otimes \left\langle c_2 \dots c_{j+1} h^{-1}_{i(c_{j+2})}(c_{j+1}) \right| \\ &= \left\langle c_1 c_2 \dots c_{j+1} h^{-1}_{i(c_{j+2})}(c_{j+1}) \right| .\end{aligned}$$

Thus, this holds for $j+1$, and so by induction, it holds for all $1 \le j \le i$. ☐

That is, selecting row $c_1 \dots c_j c_{j+1}$ of the product $H_{i,j}A$ is equivalent to selecting row $c_1 \dots c_j h_{i(c_{j+1})}(c_j)$ of A. With these tools in place, if we have that A_i is the adjacency matrix of a subgraph of $G_{i,q}$, we make the following claim.

Theorem 3.39. *If L_i is a subgraph of $G_{i,q}$ with adjacency matrix A_i, then $A_{i+1} = H_{i,i}(A_i \otimes I_q)$ is the adjacency matrix of a subgraph L_{i+1} of $G_{i+1,q}$ which is the lift of L_i under the homomorphism H_i.*

Proof. We know that H_i maps vertices of $G_{i+1,q}$ to vertices of $G_{i,q}$ by definition. Since $V_{L_{i+1}} = V_{G_{i+1,q}}$ and $V_{L_i} = V_{G_{i,q}}$, there is no concern about H_i as a map from the vertices of L_{i+1} to those of L_i. We must show, however, that when H_i is restricted to the edges $E_{L_{i+1}}$ represented by A_{i+1}, the homomorphic images are all contained in E_{L_i}, which is represented by A_i.

Suppose the $E_{L_{i+1}}$ contains an edge (u, v). We must show that E_{L_i} contains the edge $(H_i(u), H_i(v))$. We have $u, v \in \mathbb{Z}_q^{i+1}$, and write $u = (u_1, \dots, u_{i+1}), v = (v_1, \dots, v_{i+1})$. Then we

know that A_{i+1} has a 1 in row $C_{i+1}(u_1, \ldots, u_{i+1})$ and column $C_{i+1}(v_1, \ldots, v_{i+1})$. Let $c_1 \ldots c_{i+1}$ be the q-ary representation of $C_{i+1}(u_1, \ldots, u_{i+1})$ and $c'_1 \ldots c'_{i+1}$ be the q-ary representation of $C_{i+1}(v_1, \ldots, v_{i+1})$. By the definition of C_{i+1}, we have that

$$\begin{aligned} c_1 \ldots c_{i+1} &= C_i(H_i(u_1, \ldots, u_{i+1}))C_1(u_1) \\ &= c_1 \ldots c_i u_1, \\ c'_1 \ldots c'_{i+1} &= C_i(H_i(v_1, \ldots, v_{i+1}))C_1(v_1) \\ &= c'_1 \ldots c'_i v_1. \end{aligned}$$

Now consider row $c_1 \ldots c_i u_1$ of $A_{i+1} = H_{i,i}(A_i \otimes I_q)$. This is equivalent to multiplication on the left by $\langle c_1 \ldots c_i u_1|$, and using Lemma 3.38, we see that this is

$$\begin{aligned} \langle c_1 \ldots c_i u_1| \, H_{i,i}(A_i \otimes I_q) &= \left\langle c_1 \ldots c_i h_{i(u_1)}(c_i)\right| (A_i \otimes I_q) \\ &= \left(\langle c_1 \ldots c_i| \otimes \left\langle h^{-1}_{i(u_1)}(c_i)\right|\right) (A_i \otimes I_q) \\ &= (\langle c_1 \ldots c_i| \, A_i) \otimes (\left\langle h^{-1}_{i(u_1)}(c_i)\right| I_q) \\ &= (\langle c_1 \ldots c_i| \, A_i) \otimes \left\langle h^{-1}_{i(u_1)}(c_i)\right| . \end{aligned}$$

By the definition of $\otimes$, this has a 1 in column $c'_1 \ldots c'_i v_1$ if and only if there is a 1 in column $c'_1 \ldots c'_i$ of $\langle c_1 \ldots c_i| \, A_i$ and a 1 in column v_1 of $\left\langle h_{i(u_1)}(c_i)\right|$. But this is in turn equivalent to a 1 in row $c_1 \ldots c_i$, column $c'_1 \ldots c'_i$ of A_i and that $v_1 = h_{i(u_1)}(c_i)$. Since $C_i(H_i(u_1, \ldots, u_{i+1})) = c_1 \ldots c_i$, by the definition of C_i, we must have that c_i is the first entry of $H_i(u_1, \ldots, u_{i+1})$, which is $h_i(u_1, u_2)$. That is, $h_i(u_1, u_2) = c_i$, and so $h_{i(u_1)}(u_2) = c_i$, and thus $v_1 = h^{-1}_{i(u_1)}(c_i) = u_2$. Thus, we have that $v_1 = u_2$, and by the definition of A_i, there is an edge from vertex $c_1 \ldots c_i = C_i(H_i(u_1, \ldots, u_{i+1}))$ of L_i to vertex $c'_1 \ldots c'_i = C_i(H_i(v_1, \ldots, v_{i+1}))$ of L_i. This means precisely that there is an edge from $H_i(u_1, \ldots, u_{i+1})$ to $H_i(v_1, \ldots, v_{i+1})$, or that $(H_i(u), H_i(v)) \in E_{L_i}$.

Furthermore, since $(H_i(u), H_i(v)) \in E_{L_i} \subset E_{G_{i,q}}$, we have that the last $i - 1$ entries of $H_i(u)$ are the same as the first $i - 1$ entries of $H_i(v)$. That is, $h_i(u_{j+1}, u_{j+2}) = h_i(v_j, v_{j+1})$ for $1 \leq j \leq i - 1$. We now inductively show that $u_{j+1} = v_j$ for $1 \leq j \leq i - 1$. For the base case, we have already seen that $u_2 = v_1$. Assuming that $u_{j+1} = v_j$, we then use that $h_i(u_{j+1}, u_{j+2}) =$

$h_i(v_j, v_{j+1}) = h_i(u_{j+1}, v_{j+1})$. Alternatively, we may write this as $h_{i(u_{j+1}}(u_{j+2}) = h_{i(u_{j+1}}(v_{j+1})$, and because $h_{i(u_{j+1})}(b)$ is a bijection, we must have that $u_{j+2} = v_{j+1}$. Thus, $u_{(j+1)+1} = v_{j+1}$, and the condition holds for $j + 1$. By induction this holds for all $1 \leq j \leq i - 1$. This means, however, that (u, v) is in fact an edge in $E_{G_{i+1,q}}$, and so L_{i+1} is in fact a subgraph of $G_{i+1,q}$ as claimed.

Now note that any edge with tail u must correspond to a 1 in row $c_1 \dots c_i u_1$ of A_{i+1}. As before, this is $(\langle c_1 \dots c_i | \, A_i) \otimes \left\langle h^{-1}_{i(u_1)}(c_i) \right|$, and there is a 1 precisely when there is an edge $(H_i(u), H_i(v))$ for some $v \in V_{L_i}$, and $v_1 = u_2$. If $H_i(v) = H_i(v')$ and $v_1 = u_2 = v'_1$, then we see that $v = v'$, and so the restriction

$$H'_{iE} : \{(u, v) \in E_{L_{i+1}}\} \to \{(H_i(u), H_i(v)) \in E_{L_i}\}$$

is injective. We claim it is also surjective. Let there be some $b \in \mathbb{Z}_q^i$ for which there is an edge $(H_i(u), b)$ in L_i. Then let $c'_1 \dots c'_i$ be the q-ary representation of b. We see from the above that $c'_1 \dots c'_i u_1$ is the representation for a $v \in L_{i+1}$ such that $H_i(v) = b$. Thus, this restriction is a bijection. A similar process verifies that the restriction to edges with head u is also a bijection, and so H_i is a lift from L_i to L_{i+1}. □

With these representative results out of the way, we move on to build up the results dealing with the indexing and symbol-finding of de Bruijn sequences.

3.4 Indexing

In this section, we build up to a function that allows us to compute the location in a constructed sequence of any word. With the proper notation and language in place, we may now define what we mean by an edge toggling construction.

Definition 3.40. *We say that a sequence s is constructed from an order $n - 1$ de Bruijn sequence r with toggle b if there is a valid toggle point for ${}^0r$ and ${}^1r$.*

With this definition, we can now examine what such a sequence looks like in terms of the concatenation of subsequences of ${}^0r$ and ${}^1r$.

Lemma 3.41. *If there is a valid toggle point b for ${}^0r$ and ${}^1r$ as above, then there is $\gamma \in B$ such that $b\gamma$ appears in ${}^0r$ and $b\overline{\gamma}$ appears in ${}^1r$. Let s be the sequence constructed by that toggle point. Then we have*

$$s_1^{2^n} = {}^0r_1^{j-1}|{}^1r_{j'}^{2^{n-1}}|{}^1r_1^{j'-1}|{}^0r_j^{2^{n-1}},$$

where $j = I_{{}^0r}(b\gamma)$ and $j' = I_{{}^1r}(b\overline{\gamma})$.

Proof. This is verified by noting that ${}^0r_1^{j-1}$ ends precisely with the x such that the edge (x, y) contains b, and then ${}^1r_{j'}$ picks up with the y' such that the edge $(\overline{x}, y')$ contains b. The other concatenations are proved similarly. □

Note that for $b\gamma$ to appear in ${}^0r$ more than once would require $D(b\gamma)$ to appear in r more than once, a contradiction to r being a de Bruijn sequence. The same applies to $b\overline{\gamma}$ appearing twice in ${}^1r$. We now introduce a notion from [Lempel, 1970], but in our notation and setting.

Lemma 3.42. *For any sequence s of length l, and any $j \leq l$, $s_j = s_1 + S_{j-1}^{(1)}(D(s))$.*

Proof. We see quite readily that

$$\begin{aligned} s_1 + S_{j-1}^{(1)}(D(s)) &= s_1 + \sum_{i=1}^{j-1} D(s)_i = s_1 + \sum_{i=1}^{j-1}(s_i + s_{i+1}) \\ &= s_1 + \sum_{i=1}^{j-1} s_i + \sum_{i=1}^{j-1} s_{i+1} = s_1 + \sum_{i=1}^{j-1} s_i + \sum_{i=2}^{j} s_i \\ &= s_j, \end{aligned}$$

where the last line follows because addition in B is the same as subtraction, so all terms but s_j cancel. We note that this works for $j = 1$ because a sum from $i = 1$ to 0 is an empty sum and thus has a value of 0. □

Corollary 3.43. *It holds that $({}^0r)_j = S_{j-1}^{(1)}(r)$ and*

$$({}^1r)_i = 1 + S_{j-1}^{(1)}(r) = 1 + ({}^0r)_i. \tag{3.3}$$

Proof. This follows directly from applying Lemma 3.42 with $({}^0r)_1 = 0$, $({}^1r)_1 = 1$ and $D({}^0r) =$

$D({}^1r) = r$ by Definition 3.13. □

We now address some fundamental facts about sequences constructed by edge toggling:

Lemma 3.44. *If a sequence s is constructed from a $B(2, n-1)$ de Bruijn sequence r with toggle b, with $j = I_{{}^0r}(b\gamma)$ and $j' = I_{{}^1r}(b\overline{\gamma})$, then*

$$D(s)_1^{2^n} = r_1^{j-1}|r_{j'}^{2^{n-1}}|r_1^{j'-1}|r_j^{2^{n-1}}.$$

Proof. Certainly we have that $D({}^0r)_1^{j-1} = r_1^{j-2}$ by construction. Since j' is the index in ${}^1r$ of $b\overline{\gamma}$, ${}^1r_{j'} = b_1 = {}^0 r_j$, and so $D({}^0r_{j-1}|{}^1r_{j'}) = D({}^0r_{j-1}|{}^0r_j) = r_{j-1}$. Thus, the sequence $D(s)$ begins with r_1^{j-1}. Similarly, $D({}^1r_{j'}^{2^{n-1}}) = r_{j'}^{2^{n-1}-1}$, and $D({}^1r_{2^{n-1}}|{}^1r_1) = D({}^1r_{2^{n-1}}|{}^1r_{2^{n-1}+1}) = r_{2^{n-1}}$, so the next subsequence in $H(s)$ is $r_{j'}^{2^{n-1}}$. The same method may be used to prove that the next two subsequences are $r_1^{j'-1}$ and $r_j^{2^{n-1}}$. □

Lemma 3.45. *For any word $d \in B^n$, if $s_i = d_1$ and $D(s)_i^{i+n-2} = D(d)$, then $s_i^{i+n-1} = d$.*

Proof. This follows from the application of Lemma 3.42 to each entry of both s_i^{i+n-1} and d, since the part of the D-morphic image used to compute the values is equal. □

With these tools built up, we may now address the central theorem which allows us to build the index function:

Theorem 3.46. *Suppose that s is a sequence constructed from an order $n-1$ de Bruijn sequence r with a toggle b. For any $d \in B^n$, let $\ell = I_r(D(d))$, and $\sigma = S_{\ell-1}^{(1)}(r)$. Then let*

$$m = \begin{cases} \ell & \text{if } \sigma = d_1 \text{ and } \ell < j \\ \ell + 2^{n-1} & \text{if } \sigma = d_1 \text{ and } \ell \geq j \\ \ell + j - j' & \text{if } \sigma = \overline{d_1} \text{ and } \ell \geq j' \\ \ell + j - j' + 2^{n-1} & \text{if } \sigma = \overline{d_1} \text{ and } \ell < j' \end{cases},$$

where $b\gamma$ appears in ${}^0r$ and $b\overline{\gamma}$ appears in ${}^1r$, with $j = I_{{}^0r}(b\gamma)$ and $j' = I_{{}^1r}(b\overline{\gamma})$. Then $s_m^{m+n-1} = d$.

Proof. By assumption, r is an order $n-1$ de Bruijn sequence, and since $D(d)$ is a word of length $n-1$, it must appear precisely once in r, and so $\ell = I_r(D(d))$ is well defined. We shall handle each of the cases separately.

If $\sigma = d_1$ and $\ell < j$, then $s_\ell = 0 + S^{(1)}_{\ell-1}(D(s))$, so by Lemma 3.44, $s_\ell = 0 + S^{(1)}_{\ell-1}(r) = 0 + \sigma = d_1$. We claim that $D(s)_\ell^{\ell+n-2} = D(d)$. Certainly if $\ell + n - 2 \leq j - 1$, this is the case by the structure of r and the definition of ℓ. Note that since $\ell < j$, we have $\ell + n - 2 < j + n - 2$. Since j' is the index of $b\overline{\gamma}$ in ${}^1r$, we have ${}^1r_{j'}^{j'+n-2} = b$. Since j is the index of $b\gamma$ in ${}^0r$, we also have ${}^0r_j^{j+n-2} = b$, and thus $r_{j'}^{j'+n-3} = D(b) = r_j^{j+n-3}$. Since $\ell + n - 2 \leq j + n - 3$, we know that all of the remaining bits of $D(d)$ are present. Thus, we have $s_\ell = d_1$ and $D(s)_\ell^{\ell+n-2} = D(d)$, and so $s_\ell^{\ell+n-1} = d$ by Lemma 3.45.

This method relies on the first $n-2$ bits of the subsequence after the concatenation point matching the $n-2$ bits that would have followed the subsequence before the concatenation point. This same method may be used for each of the remaining sections by virtue of the toggle being b of length $n-1$. This necessitates that the homomorphic image $D(b)$ of length $n-2$ appears at the join points. For the middle concatenation, it is simply that r is a de Bruijn sequence, and so it is a cycle of length 2^{n-1}.

Thus, if $\sigma = d_1$ and $\ell \geq j$, we need only note that

$$\begin{aligned}
s_{\ell+2^{n-1}} =& 0 + S^{(1)}_{\ell+2^{n-1}-1}(D(s)) = 0 + \sum_{i=1}^{\ell+2^{n-1}-1}(D(s)) \\
=& \sum_{i=1}^{j-1} r_i + \sum_{i=j'}^{2^{n-1}} r_i + \sum_{i=1}^{j'-1} r_i + \sum_{i=j}^{\ell} r_i = \sum_{i=1}^{j'-1} r_i + \sum_{i=j'}^{2^{n-1}} r_i + \sum_{i=1}^{j-1} r_i + \sum_{i=j}^{\ell+2^{n-1}} r_i \\
=& \sum_{i=1}^{2^{n-1}} r_i + \sum_{i=1}^{\ell} r_i = 0 + S^{(1)}_\ell(r) = 0 + \sigma \\
=& d_1
\end{aligned}$$

and $D(s)_{\ell+2^{n-1}}^{\ell+2^{n-1}+n-2} = r_\ell^{\ell+n-2} = D(d)$ to see that $s_{\ell+2^{n-1}}^{\ell+2^{n-1}+n-1} = d$.

Likewise, for the final two cases, $D(d)$ appears at the appropriate location in $D(s)$ because

it appears in r. If $\sigma = \overline{d_1}$ and $\ell \geq j'$, then

$$\begin{aligned}
s_{\ell+j-j'} =& 0 + S^{(1)}_{\ell+j-j'}(r) = 0 + \sum_{i=1}^{\ell+j-j'-1} H(s)_i = \sum_{i=1}^{j-1} r_i + \sum_{i=j'}^{\ell-1} r_i \\
=& \sum_{i=1}^{j-1} r_i - \sum_{i=1}^{j'-1} r_i + \sum_{i=1}^{j'-1} r_i + \sum_{i=j'}^{\ell-1} r_i = \sum_{i=1}^{j-1} r_i - \sum_{i=1}^{j'-1} r_i + \sum_{i=1}^{\ell-1} r_i \\
=& \sum_{i=1}^{j-1} r_i - \sum_{i=1}^{j'-1} r_i + S^{(1)}_{\ell-1}(r) = S^{(1)}_{j-1}(r) + 1 - \left(1 + S^{(1)}_{j'-1}(r)\right) + S^{(1)}_{\ell-1}(r) \\
=& {}^0r_j + 1 - {}^1 r_{j'} + \sigma = b_1 + 1 - b_1 + \overline{d_1}1 + \overline{d_1} \\
=& d_1.
\end{aligned}$$

Finally, if $\sigma = \overline{d_1}$ and $\ell < j'$, then

$$\begin{aligned}
s_{\ell+j-j'+2^{n-1}} =& 0 + S^{(1)}_{\ell+j-j'+2^{n-1}-1}(D(s)) = \sum_{i=1}^{j-1} r_i + \sum_{i=j'}^{2^{n-1}} r_i + \sum_{i=1}^{\ell-1} r_i \\
=& \sum_{i=1}^{j-1} r_i + \sum_{i=j'}^{2^{n-1}} r_i + S^{(1)}_{\ell-1}(r) = \sum_{i=1}^{j-1} r_i - \sum_{i=1}^{j'-1} r_i + \sum_{i=1}^{j'-1} r_i + \sum_{i=j'}^{2^{n-1}} r_i + \sigma \\
=& \sum_{i=1}^{j-1} r_i - \sum_{i=1}^{j'-1} r_i + \sum_{i=1}^{2^{n-1}} r_i + \sigma = 1 + w(r) + \sigma = 1 + \overline{d_1} \\
=& d_1.
\end{aligned}$$

This, combined with the proper position of $D(d)$, ensures that $s_m^{m+n-1} = d$. □

Theorem 3.47. *A sequence s constructed from an order $n-1$ de Bruijn sequence r with toggle b is itself an order n de Bruijn sequence, and for any $d \in B^n$, $I_s(d) = m$ as defined above.*

Proof. By Theorem 3.46, we know that any word $d \in B^n$ appears somewhere in s. Since there are 2^n such words and exactly 2^n vertices when s is viewed as a path in G_n, each word must appear exactly once. This makes s an order n de Bruijn sequence. Since m (as computed above) gives the location of d in s, by definition, $I_s(d) = m$. □

Note that this construction provides a way to quickly calculate the index of a given word in

s if there is a quick way to calculate $S^{(1)}_{\ell-1}(r)$. Furthermore, if there exists a quick way to calculate such sums, then computing $\sum_{i=1}^{I_s(d)-1} H(s)$ is itself a quick task, which provides a quick way to check whether two words lie on separate levels ${}^0s$ and ${}^1s$. This is the inspiration for the following section.

3.5 Recursive Sums

As noted in the previous section, determining valid toggle points can be broken down into a process of summing terms in the homomorphic image of the sequence. In turn, each term in that sequence can be viewed as a sum of terms in the next order down, and so on until we reach the lowest-order sequence used in the construction.

Naively, splitting each term for the current order into multiple terms for the next order down would result in exponentially many terms to compute. This would be computationally infeasible for large enough n, and depending upon the complexity of the terms, not an improvement over generating the terms of the lower order sequence by brute force and then summing. However, as we shall see, by careful accounting of like terms, the number of terms to calculate is bounded by $3k^2 + 4k + 1$, where k is the number of orders down which we have recursed. Furthermore, each term in the end may be rewritten as a sum of binomial terms, for each of which the value may be calculated in $\log n$ time. Since we want to be able to split, rearrange, and rejoin these sequences to make like terms that can be gathered, we need to start with basic operations and build up from there. First, we acknowledge a rudimentary fact about recursive sums of sequences.

Lemma 3.48. *If $s'_i = s_i$ for $1 \leq i \leq \ell$, then*

$$S^{(m)}_{\ell'}(s') = S^{(m)}_{\ell'}(s) \tag{3.4}$$

for all $m \geq 1$ and $\ell' \leq \ell$.

Corollary 3.49. *For any sequence s, let s' be the sequence defined by $s'_1 = 0$ and $s'_i = s_{i-1}$ for all $i \geq 2$. Then for any $\ell \geq 0$,*

$$S^{(m)}_{\ell}(s') = S^{(m)}_{\ell-1}(s)$$

for all $m \geq 1$.

This is a natural consequence of considering that the leading 0 may be removed and all following elements re-indexed.

Corollary 3.50. *For any $h \geq 0$ and sequence s, let s' be the sequence defined by $s'_i = 0$ for $1 \leq i \leq h$ and $s'_i = s_{i-h}$ for all $i > h$. Then for any $\ell \geq 0$,*

$$S_\ell^{(m)}(s') = S_{\ell-h}^{(m)}(s) \tag{3.5}$$

for all $m \geq 1$.

This can be seen by repeatedly applying Corollary 3.49.

Corollary 3.51. *If $s' = 0_1^j|s$, and $\ell \geq j$, then*

$$S_\ell^{(m)}(s') = S_{\ell-j}^{(m)}(s).$$

Proof. To see this, simply construct $s'' = s_1^{\ell-j}|0_{j+1}^\ell$, and apply Corollary 3.50 to see that

$$S_\ell^{(m)}(s') = S_{\ell-j}^{(m)}(s'') = S_{\ell-j}^{(m)}(s).$$

□

We now build some more tools that are useful in proving relationships between recursive sums of sequences:

Lemma 3.52. *For any sequences s and s' and any $\ell \geq 0$,*

$$S_\ell^{(m)}(s+s') = S_\ell^{(m)}(s) + S_\ell^{(m)}(s')$$

for all $m \geq 1$.

Proof. We prove this by induction. First, for the base case $m = 1$, for any $\ell \geq 0$ we have

$$S_{\ell}^{(1)}(s+s') = \sum_{i=1}^{\ell}(s+s')_i = \sum_{i=1}^{\ell}(s_i+s'_i) = \sum_{i=1}^{\ell} s_i + \sum_{i=1}^{\ell} s'_i$$
$$= S_{\ell}^{(1)}(s) + S_{\ell}^{(1)}(s').$$

Then, for the induction step, we assume

$$S_{\ell}^{(m)}(s+s') = S_{\ell}^{(m)}(s) + S_{\ell}^{(m)}(s')$$

for some $m \geq 1$. We see that

$$S_{\ell}^{(m+1)}(s+s') = \sum_{i=1}^{\ell} S_i^{(m)}(s+s') = \sum_{i=1}^{\ell}(S_i^{(m)}(s) + S_i^{(m)}(s'))$$
$$= \sum_{i=1}^{\ell} S_i^{(m)}(s) + \sum_{i=1}^{\ell} S_i^{(m)}(s') = S_{\ell}^{(m+1)}(s) + S_{\ell}^{(m+1)}(s').$$

Thus, by induction, we have

$$S_{\ell}^{(m)}(s+s') = S_{\ell}^{(m)}(s) + S_{\ell}^{(m)}(s')$$

for all $m \geq 1$. □

Lemma 3.53. *For any $h \geq 1$ and sequences $s^1, \ldots, s^h$, for any $\ell \geq 0$,*

$$S_{\ell}^{(m)}\left(\sum_{i=1}^{h} s^i\right) = \sum_{i=1}^{h} S_{\ell}^{(m)}\left(s^i\right) \tag{3.6}$$

for all $m \geq 1$.

This is a consequence of repeatedly applying Lemma 3.52 to any sum with finite terms.

The following is a key operation that allows for dealing with the case where the sequence ends in a zero and we wish to consider the sequence without this element:

Lemma 3.54. *For any sequence s, let s' be the sequence defined by $s'_i = s_i$ for $1 \leq i \leq \ell$, and*

$s'_{\ell+1} = 0$. *Then* $\forall m \geq 1$,

$$S_{\ell+1}^{(m)}(s') = \sum_{i=1}^{m} S_{\ell}^{(i)}(s).$$

Proof. We again prove by induction: For the base case $m = 1$, we have

$$\begin{aligned} S_{\ell+1}^{(1)}(s') &= \sum_{i=1}^{\ell+1} s'_i = s'_{\ell+1} + \sum_{i=1}^{\ell} s'_i = 0 + \sum_{i=1}^{\ell} s_i = S_{\ell}^{(1)}(s) \\ &= \sum_{i=1}^{1} S_{\ell}^{(1)}(s). \end{aligned}$$

Then, for our induction step, we assume

$$S_{\ell+1}^{(m)}(s') = \sum_{i=1}^{m} S_{\ell}^{(i)}(s),$$

for some $m \geq 1$ and we show that

$$\begin{aligned} S_{\ell+1}^{(m+1)}(s') &= \sum_{i=1}^{\ell+1} S_i^{(m)}(s') = S_{\ell+1}^{(m)}(s') + \sum_{i=1}^{\ell} S_i^{(m)}(s') \\ &= \sum_{i=1}^{m} S_{\ell}^{(i)}(s) + \sum_{i=1}^{\ell} S_i^{(m)}(s) = \sum_{i=1}^{m} S_{\ell}^{(i)}(s) + S_{\ell}^{(m+1)}(s) \\ &= \sum_{i=1}^{m+1} S_{\ell}^{(i)}(s). \end{aligned}$$

Thus, by induction, we have

$$S_{\ell+1}^{(m)}(s') = \sum_{i=1}^{m} S_{\ell}^{(i)}(s)$$

for all $m \geq 1$. □

Next, we extend this last, most complicated building block so that we may work with more general sequences. To do this, we first make note of an intermediate result.

Lemma 3.55. *For all* $h \geq 1$,

$$S_1^{(h)}(\underline{1}) = 1.$$

Proof. First, we note that $\sum_{i=1}^{1} 1 = 1$, and so $S_1^{(1)}(\underline{1}) = 1$. We assume for induction that $S_1^{(h)}(\underline{1}) = 1$

for some $h \geq 1$. Then

$$S_1^{(h+1)}(\underline{1}) = \sum_{i=1}^{1} S_1^{(h)}(\underline{1}) = S_1^{(h)}(\underline{1})$$
$$=1,$$

and so by induction we have that $S_1^{(h)}(\underline{1}) = 1$ for all $h \geq 1$. ☐

Lemma 3.56. *For any $h > 0$, any $\ell \geq 0$, and any sequence s, let s' be the sequence defined by $s'_i = s_i$ for $1 \leq i \leq \ell$, and $s'_{\ell+i} = 0$ for $1 \leq i \leq h$. Then $\forall m \geq 1$,*

$$S_{\ell+h}^{(m)}(s') = \sum_{i=1}^{m} S_{m-i+1}^{(h)}(\underline{1})\, S_\ell^{(i)}(s) \tag{3.7}$$

for all $h \geq 1$.

Proof. Using Lemma 3.55, we see that Lemma 3.54 is the base case ($h = 1$). For the induction step, assume that

$$S_{\ell+h}^{(m)}(s') = \sum_{i=1}^{m} S_{m-i+1}^{(h)}(\underline{1})\, S_\ell^{(i)}(s)$$

for some $h \geq 1$. Then we have that

$$\begin{aligned} S_{\ell+(h+1)}^{(m)}(s'') &= \sum_{i=1}^{m} S_{\ell+h}^{(i)}(s') = \sum_{i=1}^{m} \sum_{j=1}^{i} S_{i-j+1}^{(h)}(\underline{1})\, S_\ell^{(j)}(s) = \sum_{j=1}^{m} \sum_{i=j}^{m} S_{i-j+1}^{(h)}(\underline{1})\, S_\ell^{(j)}(s) \\ &= \sum_{j=1}^{m} \left[\sum_{i=1}^{m-j+1} S_i^{(h)}(\underline{1}) \right] S_\ell^{(j)}(s) = \sum_{j=1}^{m} S_{m-j+1}^{(h+1)}(\underline{1})\, S_\ell^{(j)}(s) \\ &= \sum_{i=1}^{m} S_{m-i+1}^{(h+1)}(\underline{1})\, S_\ell^{(i)}(s). \end{aligned}$$

Thus, by induction, we find

$$S_{\ell+h}^{(m)}(s') = \sum_{i=1}^{m} S_{m-i+1}^{(h)}(\underline{1})\, S_\ell^{(i)}(s)$$

for all $h \geq 1$ (and any $\ell \geq 0$, $m \geq 1$). ☐

We also want to rewrite everything in terms of the lower-order sequence, for which we develop the following lemma:

Lemma 3.57. *For all $\ell \geq 1$ and $m \geq 1$,*

$$S_{\ell}^{(m)}\left({}^{0}r\right) = S_{\ell-1}^{(m+1)}(r)\,. \tag{3.8}$$

Proof. We start with the base case $m = 1$. There, we have

$$\begin{aligned} S_{\ell}^{(m)}\left({}^{0}r\right) =& \sum_{i=1}^{\ell} ({}^{0}r)_i = \sum_{i=1}^{\ell}\sum_{j=1}^{i-1} r_i = \sum_{i=1}^{\ell} S_{i-1}^{(1)}(r) \\ =& S_0^{(1)}(r) + \sum_{i=2}^{\ell} S_{i-1}^{(1)}(r) = 0 + \sum_{i=1}^{\ell-1} S_i^{(1)}(r) \\ =& S_{\ell-1}^{(2)}(r)\,. \end{aligned}$$

Then, we assume that for some $m \geq 1$ and for all $\ell \geq 1$, we have $S_{\ell}^{(m)}\left({}^{0}r\right) = S_{\ell-1}^{(m+1)}(r)$. This means that

$$\begin{aligned} S_{\ell}^{(m+1)}\left({}^{0}r\right) =& \sum_{i=1}^{\ell} S_i^{(m)}\left({}^{0}r\right) = \sum_{i=1}^{\ell} S_{i-1}^{(m+1)}(r) = S_0^{(m+1)}(r) + \sum_{i=2}^{\ell} S_{i-1}^{(m+1)}(r) \\ =& 0 + \sum_{i=1}^{\ell-1} S_i^{(m+1)}(r) = S_i^{(m+2)}(r) \\ =& S_i^{((m+1)+1)}(r)\,, \end{aligned}$$

and so

$$S_{\ell}^{(m)}\left({}^{0}r\right) = S_{\ell-1}^{(m+1)}(r)$$

for all $\ell \geq 1$ and $m \geq 1$. □

Because the initial one sequence $\underline{1}$ and the all ones sequence 1 appear frequently, we have the following result which allows us to compute values for those recursive sums. This is related to an identity sometimes known as the Hockey Stick identity or Christmas Stocking Theorem [Ross, 1997]. We write the proof using our notation for clarity.

Lemma 3.58. *For any $m \geq 1$ and $\ell \geq 1$,*

$$S_{\ell}^{(m)}(\underline{1}) = \binom{\ell + m - 2}{m - 1}. \tag{3.9}$$

Proof. For the base case $\ell = 1$, we have seen that $S_1^{(m)}(\underline{1}) = 1$ for all $m \geq 1$. We then note that $1 = \binom{m-1}{m-1} = \binom{\ell+m-2}{m-1}$, and so the claim holds for $\ell = 1$. For induction, assume that for some $\ell \geq 1$, we have that $S_\ell^{(m)}(\underline{1}) = \binom{\ell+m-2}{m-1}$.

Then we may calculate

$$\begin{aligned}
S_{\ell+1}^{(m)}(\underline{1}) &= \sum_{i=1}^{\ell+1} S_i^{(m-1)}(\underline{1}) = S_{\ell+1}^{(m-1)}(\underline{1}) + \sum_{i=1}^{\ell} S_i^{(m-1)}(\underline{1}) = \binom{\ell+m-2}{m-2} + S_\ell^{(m)}(\underline{1}) \\
&= \binom{\ell+m-2}{m-2} + \binom{\ell+m-2}{m-1} = \binom{\ell+m-2}{\ell} + \binom{\ell+m-2}{\ell-1} = \binom{\ell+m-1}{\ell} \\
&= \binom{(\ell+1)+m-2}{m-1},
\end{aligned}$$

and so the result holds for $\ell+1$, and thus by induction for all $\ell \geq 1$, and for all $m \geq 1$ □

Corollary 3.59. *It holds that*

$$S_\ell^{(m)}(1) = \binom{\ell+m-1}{m}.$$

Proof. This follows because ${}^0\underline{1}_1^\ell = 0|1_2^\ell$, and so we may write

$$\begin{aligned}
S_\ell^{(m)}(1) &\overset{(3.5)}{=} S_{\ell+1}^{(m)}\left(0|1_2^{\ell+1}\right) \overset{(3.5)}{=} S_{\ell+1}^{(m)}\left({}^0\underline{1}_1^{\ell+1}\right) \overset{(3.8)}{=} S_\ell^{(m+1)}(\underline{1}) \\
&\overset{(3.9)}{=} \binom{\ell+m-1}{m}.
\end{aligned}$$

□

Now, we put these tools to use. Essentially, we want to be able to rewrite an iterated sum of a sequence as several terms, each of which is an iterated sum over the lower-order sequence from which it was constructed. This depends upon where in the constructed sequence the index falls.

Theorem 3.60. *If the sequence s is constructed as*

$$s_1^{2^n} = {}^0r_1^{j-1}|{}^1r_{j'}^{2^{n-1}}|{}^1r_1^{j'-1}|{}^0r_j^{2^{n-1}},$$

then we may break up $S_\ell^{(m)}(s)$ *as*

$$S_\ell^{(m)}(s) = \begin{cases} S_{\ell-1}^{(m+1)}(r) \\ \qquad\qquad \text{if } 0 \le \ell < j \\ S_{\ell-j+j'-1}^{(m+1)}(r) + \sum_{i=1}^{m} \binom{\ell-j+m-i}{m-i} \left[S_{j-2}^{(i+1)}(r) - S_{j'-2}^{(i+1)}(r) \right] \\ + \binom{\ell-j+m}{m} \\ \qquad\qquad \text{if } j \le \ell < j - j' + 2^{n-1} + 1 \\ S_{\ell-j+j'-2^{n-1}-1}^{(m+1)}(r) + \sum_{i=1}^{m} \binom{\ell-j+m-i}{m-i} \left[S_{j-2}^{(i+1)}(r) - S_{j'-2}^{(i+1)}(r) \right] \\ + \sum_{i=1}^{m} \binom{\ell-j+j'-2^{n-1}+m-i-1}{m-i} S_{2^{n-1}-1}^{(i+1)}(r) + \binom{\ell-j+m}{m} \\ \qquad\qquad \text{if } j - j' + 2^{n-1} < \ell < j + 2^{n-1} \\ S_{\ell-2^{n-1}-1}^{(m+1)}(r) + \sum_{i=1}^{m} \binom{\ell-j+j'-2^{n-1}+m-i-1}{m-i} S_{2^{n-1}-1}^{(i+1)}(r) \\ + \sum_{i=1}^{m} \left[\binom{\ell-j+m-i}{m-i} - \binom{\ell-j-2^{n-1}+m-i}{m-i} \right] \left[S_{j-2}^{(i+1)}(r) - S_{j'-2}^{(i+1)}(r) \right] \\ + \sum_{i=1}^{m} \binom{\ell-j-2^{n-1}+m-i}{m-i} \binom{2^{n-1}+i-1}{i} \\ \qquad\qquad \text{if } j + 2^{n-1} \le \ell < 2^n. \end{cases}$$

Proof. We start with

$$s_1^{2^n} = {}^0r_1^{j-1}|^1r_{j'}^{2^{n-1}}|^1r_1^{j'-1}|^0r_j^{2^{n-1}}$$

and break this up into several pieces which from which we can compose s_1^ℓ. These are

$$\begin{aligned} s^{(1)} &= {}^0r_1^{j-1} \\ s^{(2)} &= 0_1^{j-1}|^1r_{j'}^{2^{n-1}} \\ s^{(3)} &= 0_1^{j-j'+2^{n-1}}|^1r_1^{j'-1} \\ s^{(4)} &= 0_1^{j-1+2^{n-1}}|^0r_j^{2^{n-1}}, \end{aligned}$$

where each of these sequences is followed by zeros. Depending on the value of ℓ, the sequence s_1^ℓ

may contain several of these. We thus determine what they break down to in terms of the sequence r.

If $\ell \geq j$, then

$$\begin{aligned} S_\ell^{(m)}\left(s^{(1)}\right) &= S_\ell^{(m)}\left({}^0r_1^{j-1}|0_j^\ell\right) \\ &\overset{(3.7),(3.4)}{=} \sum_{i=1}^m S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right) S_{j-1}^{(i)}\left({}^0r\right). \end{aligned}$$

If $\ell \geq j - j' + 2^{n-1} + 1$, then

$$\begin{aligned} S_\ell^{(m)}\left(s^{(2)}\right) &= S_\ell^{(m)}\left(0_1^{j-1}|{}^1r_{j'}^{2^{n-1}}|0_{j-j'+2^{n-1}+1}^\ell\right) \\ &\overset{(3.3),(3.6)}{=} S_\ell^{(m)}\left(0_1^{j-1}|{}^0r_{j'}^{2^{n-1}}|0_{j-j'+2^{n-1}+1}^\ell\right) + S_\ell^{(m)}\left(0_1^{j-1}|1_{j'}^{2^{n-1}}|0_{j-j'+2^{n-1}+1}^\ell\right) \\ &\overset{(3.5)}{=} S_{\ell-j+j'}^{(m)}\left(0_1^{j'-1}|{}^0r_{j'}^{2^{n-1}}|0_{2^{n-1}+1}^{\ell-j+j'}\right) + S_{\ell-j+1}^{(m)}\left(1_1^{2^{n-1}-j'+1}|0_{2^{n-1}-j'+2}^{\ell-j+1}\right) \\ &\overset{(3.6)}{=} S_{\ell-j+j'}^{(m)}\left({}^0r_1^{2^{n-1}}|0_{2^{n-1}+1}^{\ell-j+j'}\right) - S_{\ell-j+j'}^{(m)}\left({}^0r_1^{j'-1}|0_{j'}^{\ell-j+j'}\right) \\ &\quad + S_{\ell-j+1}^{(m)}\left(1_1^{2^{n-1}-j'+1}|0_{2^{n-1}-j'+2}^{\ell-j+1}\right) \\ &\overset{(3.7),(3.4)}{=} \sum_{i=1}^m S_{m-i+1}^{(\ell-j+j'-2^{n-1})}\left(\underline{1}\right) S_{2^{n-1}}^{(i)}\left({}^0r\right) - \sum_{i=1}^m S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right) S_{j'-1}^{(i)}\left({}^0r\right) \\ &\quad + S_{\ell-j+1}^{(m)}\left(1_1^{2^{n-1}-j'+1}|0_{2^{n-1}-j'+2}^{\ell-j+1}\right). \end{aligned}$$

If $\ell \geq j + 2^{n-1}$, then

$$\begin{aligned} S_\ell^{(m)}\left(s^{(3)}\right) &= S_\ell^{(m)}\left(0_1^{j-j'+2^{n-1}}|{}^1r_1^{j'-1}|0_{j+2^{n-1}}^\ell\right) \\ &\overset{(3.3),(3.6)}{=} S_\ell^{(m)}\left(0_1^{j-j'+2^{n-1}}|{}^0r_1^{j'-1}|0_{j+2^{n-1}}^\ell\right) + S_\ell^{(m)}\left(0_1^{j-j'+2^{n-1}}|1_1^{j'-1}|0_{j+2^{n-1}}^\ell\right) \\ &\overset{(3.5)}{=} S_{\ell-j+j'-2^{n-1}}^{(m)}\left({}^0r_1^{j'-1}|0_{j'}^{\ell-j+j'-2^{n-1}}\right) \\ &\quad + S_{\ell-j+1}^{(m)}\left(0_1^{2^{n-1}-j'+1}|1_{2^{n-1}-j'+2}^{2^{n-1}}|0_{2^{n-1}+1}^{\ell-j+1}\right) \\ &\overset{(3.7),(3.4)}{=} \sum_{i=1}^m S_{m-i+1}^{(\ell-j-2^{n-1}+1)}\left(\underline{1}\right) S_{j'-1}^{(i)}\left({}^0r\right) \\ &\quad + S_{\ell-j+1}^{(m)}\left(0_1^{2^{n-1}-j'+1}|1_{2^{n-1}-j'+2}^{2^{n-1}}|0_{2^{n-1}+1}^{\ell-j+1}\right). \end{aligned}$$

In addition to the previous components which may appear in s_1^ℓ, we must also deal with the last, truncated component. Finally, for each case, we break off the components previously laid out and combine like terms if possible.

If $1 \leq \ell \leq j-1$, then

$$\begin{aligned} S_\ell^{(m)}(s) &= S_\ell^{(m)}\left({}^0r_1^\ell\right) \\ &\overset{(3.8),(3.4)}{=} S_{\ell-1}^{(m+1)}(r), \end{aligned}$$

and we are done.

If $j \leq \ell \leq j - j' + 2^{n-1}$, then we may write

$$\begin{aligned} S_\ell^{(m)}\left(0_1^{j-1}|{}^1r_{j'}^{\ell-j+j'}\right) &\overset{(3.6)}{=} S_\ell^{(m)}\left(0_1^{j-1}|{}^0r_{j'}^{\ell-j+j'}\right) + S_\ell^{(m)}\left(0_1^{j-1}|1_{j'}^{\ell-j+j'}\right) \\ &\overset{(3.5)}{=} S_{\ell-j+j'}^{(m)}\left(0_1^{j'-1}|{}^0r_{j'}^{\ell-j+j'}\right) + S_{\ell-j+1}^{(m)}\left(1_1^{\ell-j+1}\right) \\ &\overset{(3.6)}{=} S_{\ell-j+j'}^{(m)}\left({}^0r_1^{\ell-j+j'}\right) - S_{\ell-j+j'}^{(m)}\left({}^0r_1^{j'-1}|0_{j'}^{\ell-j+j'}\right) \\ &\quad + S_{\ell-j+1}^{(m)}\left(1_1^{\ell-j+1}\right) \\ &\overset{(3.7),(3.4)}{=} S_{\ell-j+j'}^{(m)}\left({}^0r\right) - \sum_{i=1}^m S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right) S_{j'-1}^{(i)}\left({}^0r\right) + S_{\ell-j+1}^{(m)}(1). \end{aligned}$$

From this, we may write

$$\begin{aligned} S_\ell^{(m)}(s) &\overset{(3.4)}{=} S_\ell^{(m)}\left(s^{(1)} + 0_1^{j-1}|{}^1r_{j'}^{\ell-j+j'}\right) \\ &\overset{(3.6)}{=} \sum_{i=1}^m S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right) S_{j-1}^{(i)}\left({}^0r\right) + S_{\ell-j+j'}^{(m)}\left({}^0r\right) - \sum_{i=1}^m S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right) S_{j'-1}^{(i)}\left({}^0r\right) \\ &\quad + S_{\ell-j+1}^{(m)}(1) \\ &= S_{\ell-j+j'}^{(m)}\left({}^0r\right) + \sum_{i=1}^m S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right)\left[S_{j-1}^{(i)}\left({}^0r\right) - S_{j'-1}^{(i)}\left({}^0r\right)\right] + S_{\ell-j+1}^{(m)}(1) \\ &\overset{(3.8),(3.9)}{=} S_{\ell-j+j'-1}^{(m+1)}(r) + \sum_{i=1}^m \binom{\ell-j+m-i}{m-i}\left[S_{j-2}^{(i+1)}(r) - S_{j'-2}^{(i+1)}(r)\right] + \binom{\ell-j+m}{m}. \end{aligned}$$

If $j - j' + 2^{n-1} + 1 \le \ell \le j + 2^{n-1}$, then we may write

$$\begin{aligned}
S_\ell^{(m)}\left(0_1^{j-j'+2^{n-1}}|^1 r_1^{\ell-j+j'-2^{n-1}}\right) &\overset{(3.6)}{=} S_\ell^{(m)}\left(0_1^{j-j'+2^{n-1}}|^0 r_1^{\ell-j+j'-2^{n-1}}\right) \\
&\quad + S_\ell^{(m)}\left(0_1^{j-j'+2^{n-1}}|1_1^{\ell-j+j'-2^{n-1}}\right) \\
&\overset{(3.5),(3.4)}{=} S_{\ell-j+j'-2^{n-1}}^{(m)}\left({}^0 r\right) \\
&\quad + S_{\ell-j+1}^{(m)}\left(0_1^{2^{n-1}-j'+1}|1_{2^{n-1}-j'+2}^{\ell-j+1}\right).
\end{aligned}$$

From this, we may write

$$\begin{aligned}
S_\ell^{(m)}(s) &\overset{(3.4)}{=} S_\ell^{(m)}\left(s^{(1)} + s^{(2)} + 0_1^{j-j'+2^{n-1}}|^1 r_1^{\ell-j+j'-2^{n-1}}\right) \\
&\overset{(3.6)}{=} \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right) S_{j-1}^{(i)}\left({}^0 r\right) + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+j'-2^{n-1})}\left(\underline{1}\right) S_{2^{n-1}}^{(i)}\left({}^0 r\right) \\
&\quad - \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right) S_{j'-1}^{(i)}\left({}^0 r\right) + S_{\ell-j+1}^{(m)}\left(1_1^{2^{n-1}-j'+1}|0_{2^{n-1}-j'+2}^{\ell-j+1}\right) \\
&\quad + S_{\ell-j+j'-2^{n-1}}^{(m)}\left({}^0 r\right) + S_{\ell-j+1}^{(m)}\left(0_1^{2^{n-1}-j'+1}|1_{2^{n-1}-j'+2}^{\ell-j+1}\right) \\
&\overset{(3.6)}{=} S_{\ell-j+j'-2^{n-1}}^{(m)}\left({}^0 r\right) + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+1)}\left(\underline{1}\right)\left[S_{j-1}^{(i)}\left({}^0 r\right) - S_{j'-1}^{(i)}\left({}^0 r\right)\right] \\
&\quad + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+j'-2^{n-1})}\left(\underline{1}\right) S_{2^{n-1}}^{(i)}\left({}^0 r\right) + S_{\ell-j+1}^{(m)}\left(1_1^{\ell-j+1}\right) \\
&\overset{(3.8),(3.9)}{=} S_{\ell-j+j'-2^{n-1}-1}^{(m+1)}(r) + \sum_{i=1}^{m} \binom{\ell-j+m-i}{m-i}\left[S_{j-2}^{(i+1)}(r) - S_{j'-2}^{(i+1)}(r)\right] \\
&\quad + \sum_{i=1}^{m} \binom{\ell-j+j'-2^{n-1}+m-i-1}{m-i} S_{2^{n-1}-1}^{(i+1)}(r) + \binom{\ell-j+m}{m}.
\end{aligned}$$

If $j + 2^{n-1} \le \ell < 2^n$, then we may write

$$\begin{aligned}
S_\ell^{(m)}\left(0_1^{j+2^{n-1}-1}|^0 r_j^{\ell-2^{n-1}}\right) &\overset{(3.5)}{=} S_{\ell-2^{n-1}}^{(m)}\left(0_1^{j-1}|^0 r_j^{\ell-2^{n-1}}\right) \\
&\overset{(3.6)}{=} S_{\ell-2^{n-1}}^{(m)}\left({}^0 r_1^{\ell-2^{n-1}}\right) - S_{\ell-2^{n-1}}^{(m)}\left({}^0 r_1^{j-1}|0_j^{\ell-2^{n-1}}\right) \\
&\overset{(3.7),(3.4)}{=} S_{\ell-2^{n-1}}^{(m)}\left({}^0 r\right) - \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j-2^{n-1}+1)}\left(\underline{1}\right) S_{j-1}^{(i)}\left({}^0 r\right).
\end{aligned}$$

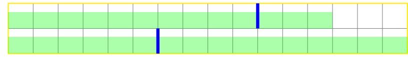
This then allows us to write

$$
\begin{aligned}
S_\ell^{(m)}(s) &\overset{(3.4)}{=} S_\ell^{(m)}\left(s^{(1)} + s^{(2)} + s^{(3)} + 0_1^{j+2^{n-1}-1}|{}^0r_j^{\ell-2^{n-1}}\right) \\
&\overset{(3.6)}{=} \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+1)}(\underline{1})\, S_{j-1}^{(i)}\left({}^0r\right) + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+j'-2^{n-1})}(\underline{1})\, S_{2^{n-1}}^{(i)}\left({}^0r\right) \\
&\quad - \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+1)}(\underline{1})\, S_{j'-1}^{(i)}\left({}^0r\right) + S_{\ell-j+1}^{(m)}\left(1_1^{2^{n-1}-j'+1}|0_{2^{n-1}-j'+2}^{\ell-j+1}\right) \\
&\quad + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j-2^{n-1}+1)}(\underline{1})\, S_{j'-1}^{(i)}\left({}^0r\right) \\
&\quad + S_{\ell-j+1}^{(m)}\left(0_1^{2^{n-1}-j'+1}|1_{2^{n-1}-j'+2}^{2^{n-1}}|0_{2^{n-1}+1}^{\ell-j+1}\right) + S_{\ell-2^{n-1}}^{(m)}\left({}^0r\right) \\
&\quad - \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j-2^{n-1}+1)}(\underline{1})\, S_{j-1}^{(i)}\left({}^0r\right) \\
&\overset{(3.6)}{=} S_{\ell-2^{n-1}}^{(m)}\left({}^0r\right) + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+1)}(\underline{1})\left[S_{j-1}^{(i)}\left({}^0r\right) - S_{j'-1}^{(i)}\left({}^0r\right)\right] \\
&\quad + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+j'-2^{n-1})}(\underline{1})\, S_{2^{n-1}}^{(i)}\left({}^0r\right) \\
&\quad + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j-2^{n-1}+1)}(\underline{1})\left[S_{j'-1}^{(i)}\left({}^0r\right) - S_{j-1}^{(i)}\left({}^0r\right)\right] \\
&\quad + S_{\ell-j+1}^{(m)}\left(1_1^{2^{n-1}}|0_{2^{n-1}+1}^{\ell-j+1}\right) \\
&\overset{(3.7),(3.4)}{=} S_{\ell-2^{n-1}}^{(m)}\left({}^0r\right) \\
&\quad + \sum_{i=1}^{m}\left[S_{m-i+1}^{(\ell-j+1)}(\underline{1}) - S_{m-i+1}^{(\ell-j-2^{n-1}+1)}(\underline{1})\right]\left[S_{j-1}^{(i)}\left({}^0r\right) - S_{j'-1}^{(i)}\left({}^0r\right)\right] \\
&\quad + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j+j'-2^{n-1})}(\underline{1})\, S_{2^{n-1}}^{(i)}\left({}^0r\right) + \sum_{i=1}^{m} S_{m-i+1}^{(\ell-j-2^{n-1}+1)}(\underline{1})\, S_{2^{n-1}}^{(i)}(1) \\
&\overset{(3.8),(3.9)}{=} S_{\ell-2^{n-1}-1}^{(m+1)}(r) + \sum_{i=1}^{m}\binom{\ell-j+j'-2^{n-1}+m-i-1}{m-i} S_{2^{n-1}-1}^{(i+1)}(r) \\
&\quad + \sum_{i=1}^{m}\left[\binom{\ell-j+m-i}{m-i} - \binom{\ell-j-2^{n-1}+m-i}{m-i}\right]\left[S_{j-2}^{(i+1)}(r) - S_{j'-2}^{(i+1)}(r)\right] \\
&\quad + \sum_{i=1}^{m}\binom{\ell-j-2^{n-1}+m-i}{m-i}\binom{2^{n-1}+i-1}{i}.
\end{aligned}
$$

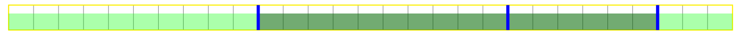
□

We can visualize the proof in an alternative manner as follows. With the two separate levels of the lift of r represented by the two rows, we see that the sequence s in the most complicated case

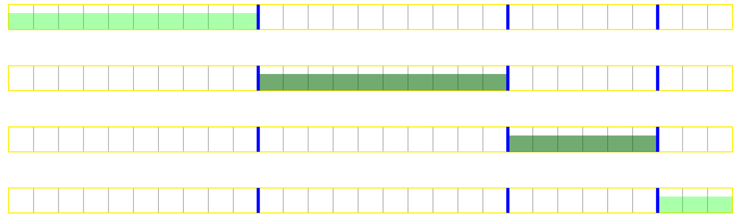
makes up the following:



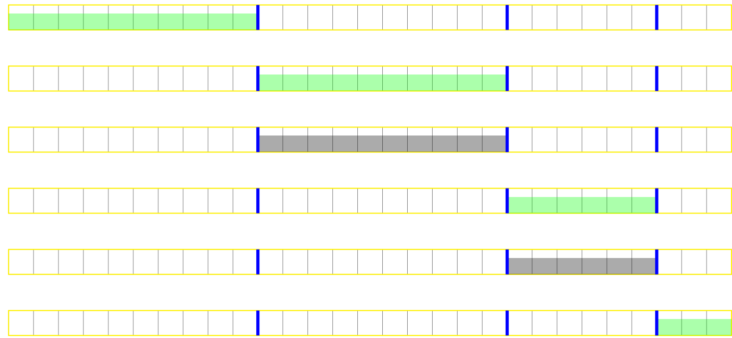
where the summation proceeds along the first row to the blue line representing the beginning of the toggle point in ${}^{0}r$, then proceeds from the blue line in the second row representing the same in ${}^{1}r$, reaches the end of ${}^{1}r$, loops back around to the beginning and proceeds to the toggle point, before jumping back up to the first row to finish. This can be rearranged linearly as



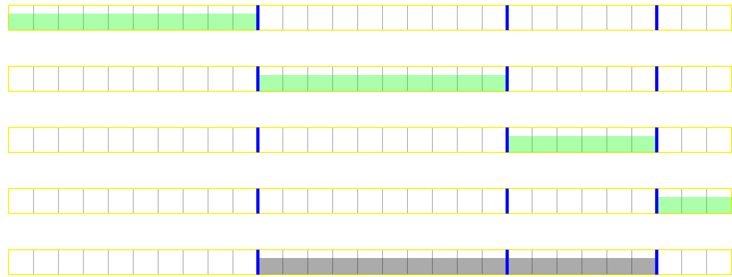
where the darker entries represent the complement of the corresponding entries in ${}^{0}r$. We then apply (3.6) to break it up into pieces:



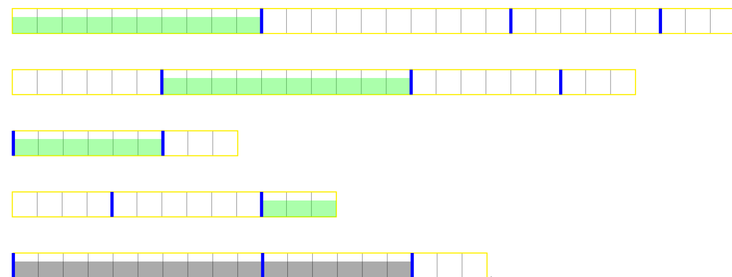
Here empty boxes represent entries that we know are 0, but are part of the sum (remember that trailing zeros cannot simply be ignored). We use (3.3) and (3.6) to separate out the ones (grey):



We can use (3.6) again to gather the ones together:

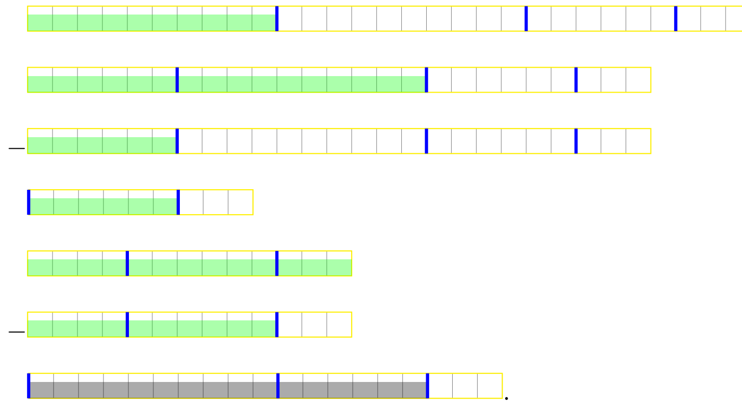


Now we shift some of these using (3.5) to make the subsequences match up with their positions in ${}^0r$:



Here, we can see that the structure mentioned at the beginning where we start on ${}^{0}r$, reach the toggle point, jump down and finish ${}^{1}r$, then wrap around from the beginning of ${}^{1}r$ to the toggle point, and finally finish up the sum in ${}^{0}r$, with the last line being the 1s that have been pulled out.

Since eliminating trailing zeros turns one term into multiple terms (here represented by "$\leftarrow$", followed by the number of zeros removed), we want to do this step last, and only once for each sequence. Since we want all terms in the end in terms of r, we must first deal with the leading zeros in some terms. We do that by once again using (3.6):



Now we may use (3.7) to eliminate the trailing zeros, keeping track of how many were eliminated:

$\leftarrow (\ell - j)$

$\leftarrow (\ell - j + j' - 2^{n-1})$

$- \leftarrow (\ell + 2^{n-1})$

$\leftarrow (\ell - j + 2^{n-1})$

$- \leftarrow (\ell - j - 2^{n-1})$

$\leftarrow (\ell - j - 2^{n-1})$.

Next we may gather like terms and rearrange to get the following:

$\leftarrow (\ell - j) - \leftarrow (\ell + 2^{n-1} - j)$

$\leftarrow (\ell + 2^{n-1} - j) - \leftarrow (\ell - j)$

$\leftarrow (\ell - j + j' - 2^{n-1})$

$\leftarrow (\ell - j - 2^{n-1})$.

The remaining steps are simply combining the two lines with the same shifts (but opposite signs), and representing everything in terms of the sequence r instead of ${}^{0}r$, which changes the index and order of summation by 1 each. We must also actually turn the shifts into sums in terms of binomials.

The other, less complex summations consist of a subset of these individual terms and a few others to deal with s ending in different quadrants. They may be graphically verified in a similar manner by removing and changing some terms.

3.6 Complexity

Now, we discuss the complexity of computing such recursive sums via Theorem 3.60. To do this we must first give several bounds. Since computations are being done in $B = \mathbb{Z}/2\mathbb{Z}$, we only need the parity of results. For this, we use a theorem of Kummer [Kummer, 1852]:

Theorem 3.61 (Kummer's Theorem (1852))**.** *Let p be a prime number and $m, n \in \mathbb{Z}$ $(m, n \geq 0)$. If k is the number of carries when $n - m$ and m are added together in base p, then $p^k \,\Big|\, \binom{n}{m}$, but $p^{k+1} \nmid \binom{n}{m}$.*

In particular, this means that $2 \,\Big|\, \binom{n}{m}$ if and only if there is at least one carry when $n - m$ and m are added together modulo 2. This leads to the following:

Corollary 3.62. *The value of $\binom{n}{m}$ in $\mathbb{Z}/2\mathbb{Z}$ can be computed in $\mathcal{O}(\log(a))$ steps, where a is the greater of $n - m$ and m.*

Proof. The number $n - m$ takes $\log(n - m)$ bits to store, and the number m takes $\log(m)$ bits. To determine the value of $\binom{n}{m}$, we only need to compare the bits one by one to see if there is a place where a carry would take place. If there is, then the value is 0. If not, the value is 1. Since there are at most $\log(a)$ bits to compare, the number of operations is $\mathcal{O}(\log(a))$. $\square$

Next, we look at the cost in terms of operations and space for expanding a single coefficient symbolically:

Lemma 3.63. *Breaking down $S_\ell^{(m)}(s)$ symbolically as in Theorem 3.60 takes $\mathcal{O}(m \log(n))$ operations and $\mathcal{O}(m)$ space.*

Proof. We must make a comparison to determine which case we are in. Depending on the case, we can determine the number of operations and storage it takes.

In the first case, we need only to compute 2 new inputs for S, and we only need to store this one coefficient.

In the second case, there are $4 + 2m$ new inputs. In addition, m coefficients need to be computed, each of which can happen in $\mathcal{O}(\log(n))$ time. Thus, the complexity is $\mathcal{O}(m \log(n))$, and each of these coefficients must be stored, so $\mathcal{O}(m)$ storage is required.

In the third case, there are $5 + 3m$ inputs to compute and $2m + 1$ coefficients. Thus, again, the runtime is $\mathcal{O}(m \log n)$ and storage complexity is $\mathcal{O}(m)$.

Finally, in the fourth case, there are $5 + 3m$ inputs and $5m$ coefficients. Again, the runtime is $\mathcal{O}(m \log n)$ and the storage complexity is $\mathcal{O}(m)$. $\square$

The real strength of this method is that many like terms are combined. That is, when shifting from recursive sums of s to recursive sums of r, there are a limited number of inputs that may appear in S.

Lemma 3.64. *Given a de Bruijn sequence s of order n constructed recursively as in Theorem 3.60, the number of unique indices that appear after recursively rewriting $S_\ell^{(1)}(s)$ in terms of the sequence of order $n - k$ is at most $1 + 3k$.*

Proof. We prove this recursively. To begin with, when $k = 0$, we only have $S_\ell^{(1)}(s)$, and so there is precisely 1 index. Next, suppose that for some $k \geq 0$, there are at most $1 + 3k$ unique indices in the expansion. Then in the expansion of each of these terms with index h, as in Theorem 3.60, there is the first term that has an index in terms of h. This results in at most $1 + 3k$ indices, with the potential for some of these indices to be the same. The remaining terms have indices that are either $j - 2$, $j' - 2$, or $2^{n-k} - 1$, where j and j' are the indices of the toggle point that created the order $n - k$ sequence from one of order $n - (k + 1)$. We note that this introduces at most 3 new indices, and so there are at most $1 + 3k + 3 = 1 + 3(k + 1)$ indices in the terms using the sequence of order $n - (k + 1)$. Thus, by induction, this result holds for any $k \geq 0$ such that the order n sequence is built recursively from an order $n - k$ sequence. $\square$

Lemma 3.65. *Given a de Bruijn sequence s of order n constructed recursively as in Theorem 3.60, the highest order of summation that appears after recursively rewriting $S_\ell^{(1)}(s)$ in terms of the sequence of order $n - k$ is at most $k + 1$.*

Proof. Again, we use induction. In the base case $k = 0$, we have $S_\ell^{(1)}(s)$, and so the only order of

summation is 1. Next, assume that for some $k \geq 0$, we know that the highest order appearing in the terms is at most $k+1$. By Theorem 3.60, we know that each of the terms of order k or less expands to terms of order at most $k+1$, and any terms of order $k+1$ may expand to terms with order $k+2 = (k+1)+1$. Thus, by induction, as long as we may recursively expand, we find that the highest order summation when $S_\ell^{(1)}(s)$ is written in terms of the order $n-k$ de Bruijn sequence is $k+1$. □

Corollary 3.66. *The total number of terms appearing in the expansion of $S_\ell^{(1)}(s)$ in terms of the sequence of order $n-k$ is at most $a(k+1)$, where $a = \max(1+3k, 2^{n-k})$.*

Proof. First, note that all indices must be between 1 and 2^{n-k}, which means there are at most $\max(1+3k, 2^{n-k})$ indices. Since $m = 1$ in $S_\ell^{(1)}(s)$, and the possible orders that appear in the expansion are between 1 and $k+1$, we have at most $a(k+1)$ terms, where $a = \max(1+3k, 2^{n-k})$. □

Now with these bounds in place, we may consider the time and space complexity of computing $S_\ell^{(1)}(s)$.

Lemma 3.67. *If a de Bruijn sequence s of order n is recursively constructed from a de Bruijn sequence of order 2 by edge toggling, then calculating $S_\ell^{(1)}(s)$ for any $1 \leq \ell \leq 2^n$ takes $\mathcal{O}\left(n^4 \log(n)\right)$ time and $\mathcal{O}\left(n^2\right)$ space.*

Proof. Since the sequence is of order n and is constructed recursively from one of order 2, we break it down into sequences of order $n-k$ for $1 \leq k \leq n-2$. For each k, we need to have stored the previous coefficients for $k-1$. This is at most $(1+3(k-1))(k-1+1) = 3k^2 - 2k$ terms. We also need storage for as many as $(1+3k)(k+1) = 3k^2+4k+1$ new coefficients. By Lemma 3.63 we see that each term takes at worst $\mathcal{O}(k\log(n))$ time to compute the coefficients. This means that the bound on the time for step k is $(3k^2-2k)\cdot\mathcal{O}(k\log(n)) = \mathcal{O}\left(k^3\log(n)\right)$. The storage allocated for all terms suffices, and so is not cumulative.

As k goes from 1 to $n-2$, the storage needed increases, but once the terms from $k-1$ have all been expanded, we no longer need to store them. This means that at any given step, we only require storage for at most $3k^2+4k+1+3k^2-2k = 6k^2+2k+1$ coefficients. Since we have at most $k = n-2$, this results in $\mathcal{O}\left(n^2\right)$ as a storage requirement. For the time complexity, we note

that at each step we may have to do $\mathcal{O}\left(k^3 \log(n)\right)$ operations, resulting in a total complexity of $\mathcal{O}\left(n^4 \log(n)\right)$. ☐

As a result of this, we are able to consider the complexity of calculating $I_s(d)$ for a word $d \in B^n$:

Corollary 3.68. *If s is a sequence as above, then calculating $I_s(d)$ for any $d \in B^n$ takes $\mathcal{O}\left(n^5 \log(k)\right)$ time and $\mathcal{O}\left(n^2\right)$ space.*

Proof. We note that Theorem 3.47 gives us the index constructed from Theorem 3.46. In that theorem, we must calculate $S^{(1)}_{\ell-1}(r)$, where $\ell = I_r(D(d))$. At the base of the recursion, these values may simply be stored in a lookup table, requiring constant time. However, each call of $S^{(1)}_{\ell-1}(r)$, when r is of order k, takes $\mathcal{O}\left(k^4 \log(k)\right)$ time and $\mathcal{O}\left(k^2\right)$ space. Again, once we have calculated the lower-order values, we may discard the space used, and so the total space requirement is $\mathcal{O}\left(n^2\right)$. The time is cumulative, however, and so it is $\mathcal{O}\left(n^5 \log(n)\right)$. ☐

Finally, this lets us build up to the central theorem of this paper, which is the complexity of specifying a de Bruijn sequence of arbitrary length from scratch.

Theorem 3.69. *One may determine an order n de Bruijn sequence s_n recursively by starting with $s_2 = 0011$ and selecting a toggle point between the D-morphic preimages ${}^0s_{k-1}$ and ${}^1s_{k-1}$ to join the two cycles into the de Bruijn sequence s_k for each order k from 3 to n. Each successful step of verifying that a given toggle is valid takes $\mathcal{O}\left(k^5 \log(k)\right)$ time and $\mathcal{O}\left(k^2\right)$ space.*

Proof. This follows swiftly from Lemma 3.67 and Corollary 3.68. For each k, we start with the sequence s_{k-1} (where $s_2 = 0011$ is the base sequence) and select a random $d \in B^{k-1}$. We must determine if d followed by 0 and d followed by 1 ($d0$ and $d1$, respectively) appear on different cycles. Thus we calculate $j = I_{s_{k-1}}(D(d0))$ and $j' = I_{s_{k-1}}(D(d1))$, which takes $\mathcal{O}\left(k^5 \log(k)\right)$ time and $\mathcal{O}\left(k^2\right)$ space. Next, we check that $S^{(1)}_j(s_{k-1}) \neq S^{(1)}_{j'}(s_{k-1})$ so that j and j' are the indices in ${}^0s_{k-1}$ and ${}^1s_{k-1}$ of $d0$ and $d1$ (though not necessarily in that order). This step takes $\mathcal{O}\left(k^4 \log(k)\right)$ time and $\mathcal{O}\left(k^2\right)$ space, and so together these two steps take $\mathcal{O}\left(k^5 \log(k)\right)$ time and $\mathcal{O}\left(k^2\right)$ space. If the check fails, we simply select another $d \in B^{k-1}$. ☐

3.7 Conclusions

In this paper, we showed how to efficiently determine whether a companion pair (equivalently, a conjugate pair) lies on two separate cycles given knowledge of the recursive structure. Theorem 3.47 provided a way to recursively locate a given word, relying on the value of a sum of a particular subsequence. Theorem 3.60 allowed us to determine such sums recursively. In Lemma 3.67, it was shown that by combining like terms, the time complexity of this operation is $\mathcal{O}\left(n^4 \log(n)\right)$, which meant the index function has complexity $\mathcal{O}\left(n^5 \log(n)\right)$. Finally, Theorem 3.69 gave the total complexity of the construction step when a valid toggle is chosen randomly.

The total complexity of this construction depends upon the proportion of toggles that are valid, and thus how many times each step must be run before a valid construction is found. As a result, we are interested in how often a randomly chosen toggle works. This is complicated by the fact that the proportion of toggles that work depends upon the previous choices of toggles for lower orders. However, we have verified the following data:

n	$C(n)$	2^n	min. children	$\frac{C(n+1)}{C(n)2^n}$
2	1	4	2	0.5
3	2	8	6	0.75
4	12	16	6	0.5417
5	104	32	12	0.4952
6	1648	64	20	0.4860
7	51264	128	44	0.4955,

where $C(n)$ is the total number of de Bruijn sequences of order n we were able to generate using this method, 2^n is the number of potential toggles tried for *each* sequence of order n to attempt to build a sequence of order $n+1$, minimum children is the smallest number of those that were found to work for a given sequence, and $\frac{C(n+1)}{C(n)2^n}$ gives the average proportion of toggles that were valid. This is because it is the ratio of the number of sequences of order $n+1$ (which required a valid toggle) to the total number of attempts (sequences of order n times the number of potential toggles).

As can be seen from the table, this proportion is about $\frac{1}{2}$. This leads us to theorize that an average of 2 attempts will be made at each step, which would make the total time com-

plexity $\mathcal{O}\left(n^6 \log n\right)$. It can also be seen that the minimum number of children was never less than $\frac{1}{4}$ of the number of potential toggle points. That is, the number of valid toggles was always at least 2^{n-2}, which leads us to theorize that this method can be used to generate at least $\prod_{k=3}^{n} 2^{k-3} = 2^{\frac{(n-3)(n-2)}{2}}$ sequences. This would be an improvement on the 2^{n-2} sequences generated by [Alhakim and Akinwande, 2011] in the binary case. In the context of devices using de Bruijn sequences for stream ciphers, this increased number of sequences could mean an improvement in the security for the same sequence length.

Appendices

Appendix A De Bruijn Seqeunce generator code

```
/*
 * DBSeq.cpp
 *
 *  Created on: Nov 19, 2019
 *      Author: Travis Alan Baumbaugh
 */

#include "DBSeq.h"
#include "RSums.h"

SeNode::SeNode(int wLen){
	n = wLen;
	symbols = new int[n];
	epos = n-1;
}

SeNode::SeNode(int* syms, int wLen) : SeNode(wLen){
	// Copy the symbols into the new array representing the node
	for (int i=0; i<n; ++i){
		symbols[i]=syms[i];
	}
}

SeNode* SeNode::image(Hom* hom){
	// Create a blank image of the node
	SeNode* newNode = new SeNode(n-1);
	// Loop through the current node
	int cur;
	int next = (epos + 1) % n;	// Initialize second input to beginning
	for (int i=0; i<n-1; ++i){
		cur = next;				// Use the next input
		if (++next == n) next=0;// Increment next input modulo n
		newNode->symbols[i]=hom->h(symbols[cur],symbols[next]);
	}
	return newNode;
}

int SeNode::toInt(int k){
	int val = 0;
	int pos = epos;
	for (int i=0; i<n; ++i){
		if (++pos == n) pos = 0;
		val *= k;
		val += symbols[pos];
	}
	return val;
}

int SeNode::last(){
	return symbols[epos];
}

int SeNode::first(){
	int pos = epos;
	if (++pos == n) pos = 0;
	return symbols[pos];
}

void SeNode::append(int symb){
	// Update the position of the end of the node to effectively shift
	if (++epos == n) epos=0;
```

```
62          // Insert the new symbol;
63          symbols[epos] = symb;
64  }
65
66  bool SeNode::compareEnd(int* potEnd){
67          // Check each entry one by one
68          int pos = epos;
69          if (++pos == n) pos = 0;
70          for (int i=0; i<n-1; ++i){
71                  if (++pos == n) pos = 0;
72                  if (potEnd[i] != symbols[pos]){
73                          return false;
74                  }
75          }
76          return true;
77  }
78
79  bool SeNode::operator==(const SeNode& right) const {
80          // Check that the two nodes have the same size
81          if (n != right.n){
82                  return false;
83          }
84          // Loop through all of the symbols
85          int lpos = epos;
86          int rpos = right.epos;
87          for (int i=0; i<n; i++){
88                  // Increment the positions
89                  if (++lpos==n) lpos = 0;
90                  if (++rpos==n) rpos = 0;
91                  // Compare the current symbols
92                  if (symbols[lpos] != right.symbols[rpos]){
93                          return false;
94                  }
95          }
96          return true;
97  }
98
99  bool SeNode::operator!=(const SeNode& right) const {
100         return !(*this == right);
101 }
102
103 SeNode* Hom::H(SeNode* orgNode){
104         return orgNode->image(this);
105 }
106
107 Hom::Hom(int numSymb){
108         k = numSymb;
109 }
110
111 // Basic versions of h and h inverse that do not involve a scalar
112 int Hom::h(int first, int second){
113         return (first + k -second) % k;
114 }
115
116 int Hom::hinv(int first, int output){
117         return (first + output) % k;
118 }
119
120 DBSeq::DBSeq(int numSymb, int wLen){
121         k = numSymb;
122         n = wLen;
123         // Initialize the beginning and ending symbols
124         begSyms = new int[n];
125         endSyms = new int[n];
```

```
126         for (int i=0; i<n; ++i){
127                 begSyms[i] = 0;
128                 endSyms[i] = 0;
129         }
130 }
131
132 DBSeq::~DBSeq(){
133         delete[] begSyms;
134         delete[] endSyms;
135 }
136
137 DBSeqb::DBSeqb(int numSymb, int wLen, int* seqForm) : DBSeq(numSymb, wLen){
138         k = numSymb;
139         // Calculate the size of the lookup table
140         int ktn = 1;
141         for (int i=0; i<n; ++i){
142                 ktn *= k;
143         }
144         lookup = new int[ktn];
145         for (int i=0; i<ktn; ++i){
146                 lookup[i] = 0;
147         }
148         // Fill in a current node
149         SeNode* curNode = new SeNode(seqForm,n);
150         // Fill in the lookup table
151         int pos = n-1;
152         int symb = 0;
153         for (int i=0; i<ktn; ++i){
154                 if (++pos == ktn) pos = 0;
155                 symb = seqForm[pos];
156                 lookup[curNode->toInt(k)] = symb;
157                 curNode->append(symb);
158         }
159         // Modify the end symbol
160         endSyms[0] =1;
161 }
162
163 DBSeqb::~DBSeqb(){
164         delete lookup;
165 }
166
167 DBSeqr::DBSeqr(Hom* homomorphism, int wLen, DBSeq* recSeq, int* togPoint,
168 int zeroToggle, int oneToggle, int levShift) : DBSeq(homomorphism->k, wLen){
169         hom = homomorphism;
170         subSeq = recSeq;
171         // Copy in the toggle
172         toggle = new int[n-1];
173         for (int i=0; i<n-1; ++i){
174                 toggle[i]=togPoint[i];
175         }
176         jo = zeroToggle;
177         jp = oneToggle;
178         // Calculate the value of en
179         // Compute the value of en
180         en = 1;
181         for (int i=0; i<n-1; ++i){
182                 en *= k;
183         }
184         shift = levShift;
185         // Modify the end symbol
186         //endSyms[0] = levShift;
187 }
188
189 DBSeqr::~DBSeqr(){
```

```
190         delete toggle;
191 }
192
193 DBIter* DBSeq::begin(){
194         // Create an empty integer array
195         return genIter(new SeNode(begSyms, n));
196 }
197
198 DBIter* DBSeq::end(){
199         return genIter(new SeNode(endSyms, n));
200 }
201
202 // Blank iterator to prevent DBSeq from being abstract
203 DBIter* DBSeq::genIter(SeNode* begNode){
204         return new DBIter();
205 }
206
207 DBIter* DBSeqb::genIter(SeNode* begNode){
208         return new DBIterb(*this, begNode);
209 }
210
211 DBIter* DBSeqr::genIter(SeNode* begNode){
212         return new DBIterr(*this, begNode);
213 }
214
215 // Blank rank function to prevent DBSeq from being abstract
216 int DBSeq::rank(SeNode& node){
217         return 0;
218 }
219
220 int DBSeqb::rank(SeNode& node){
221         // Find the rank by brute force
222         int i = 1;
223         // Get an iterator for this sequence
224         DBIter& it = *(this->begin());
225         while (*it != node){
226                 ++it;
227                 ++i;
228         }
229         return i;
230 }
231
232 int DBSeqr::rank(SeNode& node){
233         // Find the rank recursively
234         int ellp = subSeq->rank(*(hom->H(&node)));
235         // Create a coefficient array for the lower order sequence
236         CoeffArr lowerCoeffs(k, n-1, jo, jp, 1);
237         lowerCoeffs.addCoeff(ellp-1,1);
238         int d = subSeq->eval(&lowerCoeffs);
239         // Use whether the value is d1 to determine level
240         if (d == node.first()){
241                 // The node will appear on the 0 level
242                 if (ellp < jo){
243                         // The node is in the first "quadrant"
244                         return ellp;
245                 } else {
246                         // The node is in the last "quadrant"
247                         return ellp + en;
248                 }
249         } else {
250                 // The node will appear on the 1 level
251                 if (ellp < jp){
252                         // The node is in the third "quadrant"
253                         return ellp + jo - jp + en;
```

```
254                 } else {
255                         // The node is in the second "quadrant"
256                         return ellp + jo - jp;
257                 }
258         }
259 }
260
261 // Blank eval function to prevent DBSeq from being abstract
262 int DBSeq::eval(CoeffArr* curArray){
263         return 0;
264 }
265
266 int DBSeqb::eval(CoeffArr* curArray){
267         return curArray->evaluate(this);
268 }
269
270 int DBSeqr::eval(CoeffArr* curArray){
271         // We need to go (at least) one level lower with the coefficients
272         CoeffArr lowerCoeffs(k, n-1, jo, jp, curArray->maxM+1);
273         curArray->lowerOrder(&lowerCoeffs);
274         return subSeq->eval(&lowerCoeffs);
275 }
276
277 DBIter::~DBIter(){
278         // Empty destructor
279 }
280
281 DBIterb::DBIterb(DBSeqb & seq, SeNode* newNode) : parSeq( seq ){
282         curNode = newNode;
283 }
284
285 DBIterb::DBIterb(DBIterb& oldIter) : parSeq( oldIter.parSeq ) {
286         curNode = oldIter.curNode;
287 }
288
289 DBIterb::~DBIterb(){
290         delete curNode;
291 }
292
293 DBIterr::DBIterr(DBSeqr & seq, SeNode* newNode) : parSeq( seq ),
294 subIter( parSeq.subSeq->genIter(parSeq.hom->H(newNode)) ){
295         curNode = newNode;
296 }
297
298 DBIterr::~DBIterr(){
299         delete curNode;
300         delete subIter;
301 }
302
303 // Default behaviour is undefined
304 DBIter& DBIter::operator++(){
305         return *this;
306 }
307
308 DBIter& DBIterb::operator++(){
309         // Look up the new symbol and add it
310         curNode->append(parSeq.lookup[curNode->toInt(parSeq.k)]);
311         return *this;
312 }
313
314 DBIter& DBIterr::operator++(){
315         // Move through the lower-order iterator
316         ++(*subIter);
317         // Find the preimage of the new node
```

```
318         int newSymb = parSeq.hom->hinv(curNode->last(),(*(*subIter)).last());
319         // Test if this node will be a toggle
320         if (curNode->compareEnd(parSeq.toggle)){
321                 // Add on the shift to implement the toggle
322                 newSymb=parSeq.hom->hinv(newSymb,parSeq.shift);
323                 curNode->append(newSymb);
324                 // Modify the sub-iterator to handle the change in image
325                 subIter = parSeq.subSeq->genIter(parSeq.hom->H(curNode));
326         } else {
327                 curNode->append(newSymb);       // Just add the new symbol
328         }
329         return *this;
330 }
331
332 SeNode& DBIter::operator*(){
333         // Get a reference to the current node
334         return *curNode;
335 }
336
337 void DBIterb::setNode(SeNode* newNode){
338         curNode = newNode;
339 }
340
341 void DBIterr::setNode(SeNode* newNode){
342         curNode = newNode;
343 }
344
345 // Blank function to create necessary object code
346 void DBIter::setNode(SeNode* newNode){
347         return;
348 }
```

Listing 1: de Bruijn and Interator Code

Appendix B Toggle Checking Code

```
/*
 * RSums.cpp
 *
 *  Created on: Dec 3, 2019
 *      Author: Travis Alan Baumbaugh
 */

#include "RSums.h"

void CoeffArr::expand(int ell, int m){
	// Only Works in BINARY
	if (ell<jo){
		this->addCoeff(ell-1, m+1);
	} else if (ell <= jo-jp+en){
		this->addCoeff(ell-jo+jp-1,m+1);
		// Loop through possible orders
		for (int i=1; i<=m; ++i){
			//Calculate the coefficient
			if (this->binMod(ell-jo,m-i) == 1){
				this->addCoeff(jo-2, i+1);
				this->addCoeff(jp-2, i+1);
			}
		}
		// Add to the number
		this->addVal(this->binMod(ell-jo, m));
	} else if (ell < jo+en){
		this->addCoeff(ell-jo+jp-en-1, m+1);
		// Loop through possible orders
		for (int i=1; i<=m; ++i){
			//Calculate the coefficient
			if (this->binMod(ell-jo,m-i) == 1){
				this->addCoeff(jo-2, i+1);
				this->addCoeff(jp-2, i+1);
			}
		}
		// Loop through possible orders
		for (int i=1; i<=m; ++i){
			//Calculate the coefficient
			if (this->binMod(ell-jo+jp-en-1,m-i) == 1){
				this->addCoeff(en-1, i+1);
			}
		}
		// Add to the number
		this->addVal(this->binMod(ell-jo, m));
	} else {
		this->addCoeff(ell-en-1, m+1);
		// Loop through possible orders
		for (int i=1; i<=m; ++i){
			//Calculate the coefficient
			if ((this->binMod(ell-jo,m-i)^this->binMod(ell-jo-en,m-i)) == 1){
				this->addCoeff(jo-2, i+1);
				this->addCoeff(jp-2, i+1);
			}
		}
		// Loop through possible orders
		for (int i=1; i<=m; ++i){
			//Calculate the coefficient
			if (this->binMod(ell-jo+jp-en-1,m-i) == 1){
				this->addCoeff(en-1, i+1);
			}
		}
```

```
62                  // Add to the number
63                  for (int i=1; i<=m; ++i){
64                          this->addVal(this->binMod(ell-jo-en, m-i)*this->binMod(en-1, i));
65                  }
66          }
67  }
68
69  void CoeffArr::addCoeff(int index, int order){
70          // Handle the case where the index is not positive
71          if (index <= 0){
72                  return;
73          }
74          std::unordered_map<int,int*>::iterator loc = coeffs.find(index);
75          if(loc == coeffs.end()){
76                  // There have been no coefficients for this index yet, create the array
77                  int* newArray = new int[maxM];
78                  for (int i=0; i<maxM; ++i){
79                          newArray[i]=0;
80                  }
81                  coeffs.insert(std::make_pair(index, newArray));
82                  loc = coeffs.find(index);
83          }
84          // Increment this coefficient mod k
85          if (++(loc->second[order-1]) == k) loc->second[order-1] = 0;
86  }
87
88  int CoeffArr::binMod(int nMinusM, int m){
89          // Determines whether (n choose m) is divisible by 2
90          // and returns the remainder
91          if ((nMinusM&m) > 0){
92                  return 0;
93          }
94          return 1;
95  }
96
97  void CoeffArr::addVal(int val){
98          scalar += val;
99          scalar %= k;
100 }
101
102 void CoeffArr::lowerOrder(CoeffArr* r){
103         // Loop through these coefficients and add them
104         // to the lower-order coefficient array
105         std::unordered_map<int,int*>::iterator it = coeffs.begin();
106         while (it != coeffs.end()){
107                 // Get the index
108                 int ell = it->first;
109                 // Loop over all of the possible indices
110                 for (int i=0; i<maxM; ++i){
111                         if(it->second[i]){
112                                 r->expand(ell, i+1);
113                         }
114                 }
115                 ++it;
116         }
117         // Don't forget to transfer the scalar!
118         r->addVal(scalar);
119 }
120
121 int CoeffArr::evaluate(DBSeq* seq){
122         // First, determine how much of the sequence we must generate
123         int maxEll = 0;
124         std::unordered_map<int,int*>::iterator it = coeffs.begin();
125         while (it != coeffs.end()){
```

```
126                 int ell = it->first;
127                 // Determine if this index is larger than ones encountered
128                 if (ell > maxEll){
129                         maxEll = ell;
130                 }
131                 ++it;
132         }
133         // Create an array to store the part of the sequence we need
134         int* symSeq = new int[maxEll];
135         // Go through and generate the necessary part of the sequence
136         int i=0;
137         // First write 0s
138         while ((i<n) && (i<maxEll)){
139                 symSeq[i]=0;
140                 ++i;
141         }
142         // Get an iterator to the sequence
143         DBIter& node = *(seq->begin());
144         // Use it to fill the remainder of the sequence
145         while (i<maxEll){
146                 symSeq[i]=(*(++node)).last();
147                 i++;
148         }
149         // Start keeping track of the value
150         int val = 0;
151         // Go through all of the coefficients by index
152         it = coeffs.begin();
153         while (it != coeffs.end()){
154                 int ell = it->first;
155                 // Loop over all of the possible indices
156                 for (int i=0; i<maxM; ++i){
157                         if(it->second[i]){
158                                 // We have rS{ell}{i+1}{seq}
159                                 // Loop over the values up to ell
160                                 for (int j=0; j<ell; ++j){
161                                         if (symSeq[j]){
162                                                 // We have 0_{1}^{j}|1_{j+1}^{j+1}|0_{j+2}^{ell}
163                                                 // summed up to ell,
164                                                 // Equivalent to 1_{1}^{1}|0{2}^{ell-j}
165                                                 // summed up to ell-j.
166                                                 // This is \rS{ell-j}{i+1}{_1}
167                                                 // Which is \binom{ell-j+i-1}{i}
168                                                 // Which is binMod(ell-j-1,i)
169                                                 val += this->binMod(ell-j-1,i);
170                                                 val %= k;
171                                         }
172                                 }
173                         }
174                 }
175                 ++it;
176         }
177         delete[] symSeq;
178         val += scalar;
179         val %= k;
180         return val;
181 }
182
183 void CoeffArr::clear(){
184         // Loop through the indices and remove arrays
185         std::unordered_map<int,int*>::iterator it = coeffs.begin();
186         while (it != coeffs.end()){
187                 delete[] it->second;
188                 ++it;
189         }
```

```
190         coeffs.clear();
191         scalar = 0;
192 }
193
194 CoeffArr::~CoeffArr(){
195         this->clear();
196 }
197
198 CoeffArr::CoeffArr(int numSyms, int wLen, int zeroTogPoint, int oneTogPoint, int mLimit){
199         k = numSyms;
200         // Throw an error if k isn't 2 (non-binary isn't implemented yet)
201         if (k != 2){
202                 throw std::invalid_argument( "Only binary (k=2) works for now" );
203         }
204         n = wLen;
205         jo = zeroTogPoint;
206         jp = oneTogPoint;
207         // Compute the value of en
208         en = 1;
209         for (int i=0; i<n; ++i){
210                 en *= k;
211         }
212         maxM = mLimit;
213         scalar = 0;
214 }
```

Listing 2: Toggle Checking Code

Bibliography

[Akinwande, 2010] Akinwande, M. B. O. (2010). *Homomorphisms of nonbinary de Bruijn graphs with applications*. ProQuest LLC, Ann Arbor, MI. Thesis (Ph.D.)–Clarkson University.

[Alhakim and Akinwande, 2011] Alhakim, A. and Akinwande, M. (2011). A recursive construction of nonbinary de Bruijn sequences. *Des. Codes Cryptogr.*, 60(2):155–169.

[Alhakim and Nouiehed, 2017] Alhakim, A. and Nouiehed, M. (2017). Stretching de Bruijn sequences. *Des. Codes Cryptogr.*, 85(2):381–394.

[Assmus and Key, 1992] Assmus, Jr., E. F. and Key, J. D. (1992). *Designs and their codes*, volume 103 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge.

[Babbage and Dodd, 2008] Babbage, S. and Dodd, M. (2008). *The MICKEY Stream Ciphers*, pages 191–209. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Baumbaugh et al., 2018] Baumbaugh, T., Diaz, Y., Friesenhahn, S., Manganiello, F., and Vetter, A. (2018). Batch codes from Hamming and Reed-Muller codes. *J. Algebra Comb. Discrete Struct. Appl.*, 5(3):153–165.

[Beelen and Datta, 2018] Beelen, P. and Datta, M. (2018). Generalized Hamming weights of affine Cartesian codes. *Finite Fields Appl.*, 51:130–145.

[Bhattacharya et al., 2012] Bhattacharya, S., Ruj, S., and Roy, B. (2012). Combinatorial batch codes: a lower bound and optimal constructions. *Adv. Math. Commun.*, 6(2):165–174.

[Blahut, 2003] Blahut, R. E. (2003). *Algebraic codes for data transmission*. Cambridge university press.

[Boone and Peterson, 2000] Boone, J. V. and Peterson, R. R. (2000). *The Start of the Digital Revolution: SIGSALY - Secure Digital Voice Communications in WWII*. Center for Cryptologic History, National Security Agency, Fort George G. Meade, Maryland.

[Bujtás and Tuza, 2011] Bujtás, C. and Tuza, Z. (2011). Optimal combinatorial batch codes derived from dual systems. *Miskolc Math. Notes*, 12(1):11–23.

[Cadambe and Mazumdar, 2015] Cadambe, V. R. and Mazumdar, A. (2015). Bounds on the size of locally recoverable codes. *IEEE Trans. Inform. Theory*, 61(11):5787–5794.

[Canniere and Preneel, 2006] Canniere, C. D. and Preneel, B. (2006). Trivium specifications. *eSTREAM, ECRYPT Stream Cipher Project.*

[Chor et al., 1998] Chor, B., Goldreich, O., Kushilevitz, E., and Sudan, M. (1998). Private information retrieval. *J. ACM*, 45(6):965–982.

[de Bruijn, 1975] de Bruijn, N. (1975). *Acknowledgement of priority to C. Flye Sainte-Marie on the counting of circular arrangements of 2^n zeros and ones that show each n-letter word exactly once.* EUT report. WSK, Dept. of Mathematics and Computing Science. Technische Hogeschool Eindhoven.

[de Bruijn, 1946] de Bruijn, N. G. (1946). A combinatorial problem. *Nederl. Akad. Wetensch., Proc.*, 49:758–764 = Indagationes Math. 8, 461–467 (1946).

[Dimakis et al., 2011] Dimakis, A. G., Ramchandran, K., Wu, Y., and Suh, C. (2011). A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489.

[Dimsdale and Weinberg, 1960] Dimsdale, B. and Weinberg, G. M. (1960). Programmed error correction in Project Mercury. *Comm. ACM*, 3:649–652.

[Eldert et al., 1958] Eldert, C., Gurk, H. M., Gray, H. J., and Rubinoff, M. (1958). Shifting counters. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 77(1):70–74.

[Etzion and Lempel, 1984] Etzion, T. and Lempel, A. (1984). Algorithms for the generation of full-length shift-register sequences. *IEEE Trans. Inform. Theory*, 30(3):480–484.

[Fazeli et al., 2015] Fazeli, A., Vardy, A., and Yaakobi, E. (2015). Codes for distributed pir with low storage overhead. In *2015 IEEE International Symposium on Information Theory (ISIT)*, pages 2852–2856.

[Ford, 1957] Ford, L. R. (1957). *A Cyclic Arrangement of n-tuples.* Rand Corporation.

[Fredricksen, 1982] Fredricksen, H. (1982). A survey of full length nonlinear shift register cycle algorithms. *SIAM Rev.*, 24(2):195–221.

[Hamming, 1950] Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Tech. J.*, 29:147–160.

[Heijnen and Pellikaan, 1998] Heijnen, P. and Pellikaan, R. (1998). Generalized Hamming weights of q-ary Reed-Muller codes. *IEEE Trans. Inform. Theory*, 44(1):181–196.

[Hell et al., 2008] Hell, M., Johansson, T., Maximov, A., and Meier, W. (2008). *The Grain Family of Stream Ciphers*, pages 179–190. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Hell et al., 2007] Hell, M., Johansson, T., and Meier, W. (2007). Grain: A stream cipher for constrained environments. *Int. J. Wire. Mob. Comput.*, 2(1):86–93.

[Huang et al., 2016] Huang, P., Yaakobi, E., Uchikawa, H., and Siegel, P. H. (2016). Binary linear locally repairable codes. *IEEE Trans. Inform. Theory*, 62(11):6268–6283.

[Ishai et al., 2004] Ishai, Y., Kushilevitz, E., Ostrovsky, R., and Sahai, A. (2004). Batch codes and their applications. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 262–271. ACM, New York.

[Kummer, 1852] Kummer, E. E. (1852). Über die Ergänzungssätze zu den allgemeinen Reciprocitätsgesetzen. *J. Reine Angew. Math.*, 44:93–146.

[Kushilevitz and Ostrovsky, 1997] Kushilevitz, E. and Ostrovsky, R. (1997). Replication is not needed: single database, computationally-private information retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 364–373.

[Leach, 1960] Leach, E. B. (1960). Regular sequences and frequency distributions. *Proc. Amer. Math. Soc.*, 11:566–574.

[Lempel, 1970] Lempel, A. (1970). On a homomorphism of the de Bruijn graph and its applications to the design of feedback shift registers. *IEEE Trans. Comput.*, C-19:1204–1209.

[Lidl and Niederreiter, 1983] Lidl, R. and Niederreiter, H. (1983). *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley Publishing Company, Advanced Book Program, Reading, MA. With a foreword by P. M. Cohn.

[Lipmaa and Skachek, 2015] Lipmaa, H. and Skachek, V. (2015). Linear batch codes. In *Coding theory and applications*, volume 3 of *CIM Ser. Math. Sci.*, pages 245–253. Springer, Cham.

[López et al., 2019] López, H. H., Manganiello, F., and Matthews, G. L. (2019). Affine Cartesian codes with complementary duals. *Finite Fields Appl.*, 57:13–28.

[López et al., 2014] López, H. H., Rentería-Márquez, C., and Villarreal, R. H. (2014). Affine Cartesian codes. *Des. Codes Cryptogr.*, 71(1):5–19.

[MacWilliams and Sloane, 1977] MacWilliams, F. J. and Sloane, N. J. A. (1977). *The theory of error-correcting codes. II*. North-Holland Publishing Co., Amsterdam-New York-Oxford. North-Holland Mathematical Library, Vol. 16.

[Makelov, 2015] Makelov, A. A. (2015). Expansion in lifts of graphs.

[Mandal and Gong, 2013] Mandal, K. and Gong, G. (2013). Cryptographically strong de bruijn sequences with large periods. In Knudsen, L. R. and Wu, H., editors, *Selected Areas in Cryptography*, pages 104–118, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Mandal and Gong, 2016] Mandal, K. and Gong, G. (2016). Feedback reconstruction and implementations of pseudorandom number generators from composited de Bruijn sequences. *IEEE Trans. Comput.*, 65(9):2725–2738.

[Miller, 1882] Miller, F. (1882). *Telegraphic code to insure privacy and secrecy in the transmission of telegrams*. CM Cornwell.

[Muller, 1954] Muller, D. E. (1954). Application of boolean algebra to switching circuit design and to error detection. *Transactions of the I.R.E. Professional Group on Electronic Computers*, EC-3(3):6–12.

[Paterson et al., 2009] Paterson, M. B., Stinson, D. R., and Wei, R. (2009). Combinatorial batch codes. *Adv. Math. Commun.*, 3(1):13–27.

[Phillips, 2003] Phillips, G. M. (2003). *Interpolation and approximation by polynomials*, volume 14 of *CMS Books in Mathematics/Ouvrages de Mathématiques de la SMC*. Springer-Verlag, New York.

[Ramakrishnan and Wootters, 2018] Ramakrishnan, P. and Wootters, M. (2018). On taking advantage of multiple requests in error correcting codes. *CoRR*, abs/1802.00875.

[Reed, 1954] Reed, I. S. (1954). A class of multiple-error-correcting codes and the decoding scheme. *Trans. I.R.E.*, PGIT-4:38–49.

[Riet et al., 2018] Riet, A.-E., Skachek, V., and Thomas, E. K. (2018). Asynchronous Batch and PIR Codes from Hypergraphs. *ArXiv e-prints*.

[Ross, 1997] Ross, P. (1997). Generalized hockey stick identities and n-dimensional blockwalking. *The College Mathematics Journal*, 28(4):325.

[Ryser, 1963] Ryser, H. J. (1963). *Combinatorial mathematics*, volume 14. American Mathematical Soc.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Tech. J.*, 27:379–423, 623–656.

[Shannon, 1949] Shannon, C. E. (1949). Communication theory of secrecy systems. *Bell System Tech. J.*, 28:656–715.

[Shao and Wei, 1992] Shao, J. Y. and Wei, W. D. (1992). A formula for the number of Latin squares. *Discrete Math.*, 110(1-3):293–296.

[Silberstein and Gál, 2016] Silberstein, N. and Gál, A. (2016). Optimal combinatorial batch codes based on block designs. *Des. Codes Cryptogr.*, 78(2):409–424.

[Skachek, 2018] Skachek, V. (2018). Batch and PIR codes and their connections to locally repairable codes. In *Network coding and subspace designs*, Signals Commun. Technol., pages 427–442. Springer, Cham.

[Swift and Guertin, 2000] Swift, G. M. and Guertin, S. M. (2000). In-flight observations of multiple-bit upset in drams. *IEEE Transactions on Nuclear Science*, 47(6):2386–2391.

[Thomas and Skachek, 2017] Thomas, E. K. and Skachek, V. (2017). Constructions and bounds for batch codes with small parameters. In *Coding theory and applications*, volume 10495 of *Lecture Notes in Comput. Sci.*, pages 283–295. Springer, Cham.

[van Aardenne-Ehrenfest and de Bruijn, 1951] van Aardenne-Ehrenfest, T. and de Bruijn, N. G. (1951). Circuits and trees in oriented linear graphs. *Simon Stevin*, 28:203–217.

[Yang et al., 2017] Yang, B., Mandal, K., Aagaard, M. D., and Gong, G. (2017). Efficient composited de Bruijn sequence generators. *IEEE Trans. Comput.*, 66(8):1354–1368.

[Yoeli, 1963] Yoeli, M. (1963). Counting with nonlinear binary feedback shift registers. *IEEE Transactions on Electronic Computers*, EC-12(4):357–361.