BEN–GURION UNIVERSITY OF THE NEGEV

FACULTY OF ENGINEERING SCIENCES

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SNAKE-IN-THE-BOX CODES FOR RANK MODULATION

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE M.Sc. DEGREE

By: Yonatan Yehezkeally

June 2016

BEN–GURION UNIVERSITY OF THE NEGEV

FACULTY OF ENGINEERING SCIENCES

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SNAKE-IN-THE-BOX CODES FOR RANK MODULATION

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE M.Sc. DEGREE

By: Yonatan Yehezkeally

Supervised by: Prof. Moshe Schwartz

Author: _____ Date: 15/06/2016

Supervisor: _____ Date: 15/06/2016

Supervisor: _____ Date: _____

Chairman of M.Sc. degree committee: _____ Date: 15/06/2016

June 2016

Abstract

Motivated by the rank-modulation scheme with applications to flash memory, we consider Gray codes capable of detecting a single error, also known as snake-in-the-box codes. We study two error metrics: Kendall's τ-metric, which applies to charge-constrained errors, and the ℓ_∞-metric, which is useful in the case of limited-magnitude errors. In both cases we construct snake-in-the-box codes with rate asymptotically tending to 1. We also provide efficient successor-calculation functions, as well as ranking and unranking functions. Finally, we also study bounds on the parameters of such codes.

Index terms

Snake-in-the-box codes, rank modulation, permutations, flash memory

Acknowledgments

I would like to evince my sincerest gratitude to my supervisor, Prof. Moshe Schwartz, for his guidance, counsel, and helping hand, within academia and without, and also for his unfailing patience, without all of which neither this work in particular nor my graduate studies in general would ever have come to fruition.

Yonatan Yehezkeally

Contents

Notations

- $[n]$ - The set of positive integers less than or equal to n, $\{1, 2, \ldots, n\}$.
- S_n - The symmetric group of order n; the group of all permutations on the set $[n]$. Also denoted as Sym_n.
- $[a_1, a_2, \ldots, a_n]$ (where $\{a_i\}_{i=1}^n = [n]$) - The vector notation of the permutation σ such that $\sigma(i) = a_i$.
- $(a_1, a_2, \ldots, a_k)$ (where $k \leq n$ and $\{a_i\}_{i=1}^k \subset [n]$ are k distinct elements) - The cycle notation of the permutation σ such that $\sigma(a_i) = a_{(i \bmod k)+1}$, $i = 1, 2, \ldots, k$, and $\sigma(b) = b$ for $b \in [n] \setminus \{a_i\}_{i=1}^k$.
- $\sigma\tau$ (where $\sigma, \tau \in S_n$) - The composition of σ, τ, i.e., $(\sigma\tau)(i) = \sigma(\tau(i))$.
- A_n - The Alternating group of order n; the subgroup of even permutations in S_n.
- $R(C)$ (where $C \subset S$ is a code) - The Rate of C, defined $R(C) = \frac{\log|C|}{\log|S|}$.
- $t_i : S_n \to S_n$ - The "push-to-the-top" operation on the ith index: $t_i([a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n]) = [a_i, a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n]$.
- $\underline{t}_i : S_n \to S_n$ - The "push-to-the-bottom" operation on the $(n+1-i)$th index.
- $(n, M, \mathcal{M})$-snake - A Gray code over S_n, of size M, and minimal $\mathcal{M}$-metric distance 2. Sometimes abbreviated $\mathcal{M}$-snake.
- Kendall's τ-metric - the metric on S_n, denoted by $d_{\mathcal{K}}(\alpha, \beta)$, which is defined as the minimal number of adjacent transpositions required to transform α to β.
- ℓ_∞-metric - the metric on S_n defined by $d_\infty(\alpha, \beta) = \max|\alpha(i) - \beta(i)|$.
- $G_n = (V_n, E_n)$ - Kendall's τ adjacency graph; $V_n = S_n$ and $\{\alpha, \beta\} \in E_n$ if $d_{\mathcal{K}}(\alpha, \beta) = 1$.

List of Figures

Chapter 1

Introduction

Flash memory is a non-volatile storage medium which is electrically programmable and erasable. Its current wide use is motivated by its high storage density and relative low cost. Among the chief disadvantages of flash memories is their inherent asymmetry between cell programming (injecting cells with charge) and cell erasure (removing charge from cells). While single cells can be programmed with relative ease, in the current architecture, the process of erasure can only be preformed by completely depleting large blocks of cells of their charge. Moreover, the removal of charge from cells physically damages them over time.

This issue is exacerbated as a result of the ever-present demand for denser memory: smaller cells are more delicate, and are damaged faster during erasure. They also contain less charge and are therefore more prone to error. In addition, flash memories, at present, use multilevel cells, where charge-levels are quantized to simulate a finite alphabet – the more levels, the less safety margins are left, and data integrity is compromised. Consequently, over-programming (increasing a cell's charge-level above the designated mark) is a real problem, requiring a costly and damaging erasure cycle. Hence, in a programming cycle, charge-levels are usually made to gradually approach the desirable value, making for lengthier programming cycles as well (see [1]).

Recent works have proposed to alleviate this concern by jointly storing information in a group of cells, which allows for a trade-off between data capacity and rewriting capability [2]. In generalization of the Write Once Memory model (first presented in [3]), the Write Asymmetrical Memory model was discussed, applicable to flash memory (among

other media). By representing several logical variables in a group of physical multilevel-cells–rather than representing each separately–*floating codes* were presented in which a programming cycle was limited to modifying a single logical variable, and were shown ([4, 5]) to be asymptotically optimal in the number of rewriting cycles they guarantee between erasure operations (dubbed *deficiency*) in the case of constant data alphabet. Codes were also developed for representing recent entries in a data stream, named *buffer codes* (see [2] and references therein). Both schemes have since been generalized in [6], which also broadened the range of cases where asymptotically optimal deficiency can be obtained. [7] also analysed the expected deficiency, as opposed to worst-case analysis, and presents encoding-decoding schemes which improve known results in some cases.

Another effort to counter the effects of write-asymmetry in flash memories was recently made in the introduction of the Rank-Modulation scheme [8]. This scheme represents the data stored in a group of cells in the permutation suggested by their relative charge-levels rather than the levels themselves. That is, if $c_1, c_2, \ldots, c_n \in \mathbb{R}$ represent the charge-levels of $n \in \mathbb{N}$ cells, then that group of cells is said to encode the permutation $\sigma \in S_n$ such that:

$$c_{\sigma(1)} > c_{\sigma(2)} > \ldots > c_{\sigma(n)} > 0.$$

This scheme eliminates the need for discretization of charge-levels. Furthermore, it was suggested in [8] that programming could be restricted to "push-to-the-top" operations, under which constraint one only programs a group of cells by increasing the charge-level of a single cell above that of all others. In this manner, over-programming is no longer an issue.

In addition, storing data using this scheme also improves the memory's robustness against other noise types. Retention–the process of slow charge leakage from cells–tends to affect all cells with a similar trend [1]. Since rank modulation stores information in the differences between charge-levels rather than their absolute values, it offers more resilience against that type of noise. It is also worth noting that the advantages of rank modulation have been experimentally applied to phase-change memory (see [9]).

Gray codes using "push-to-the-top" operations and spanning the entire space of permutations were also studied in [8]. The Gray code [10] was first introduced as a sequence

of distinct binary vectors of fixed length, where every adjacent pair differs in a single coordinate. It has since been generalized to sequences of distinct states $s_1, s_2, \ldots, s_k \in S$ such that for every $i < k$ there exists a function in a predetermined set of transitions $t \in T$ such that $s_{i+1} = t(s_i)$.

Such codes have been found to be applicable to a wide range of problems (see [11] for a survey), including permanent-computation [12], circuit-testing [13], image-processing [14], hashing [15], coding [8,16,17] and data storing/extraction [18]. In particular, the existence of such codes was put in context of the Lovász conjecture, a currently-open problem, whereas their generation is cast into the problem of finding a Hamiltonian path (or cycle) in Cayley digraphs, which is of notable interest in our work.

Specifically, when the states one considers are permutations on $n \in \mathbb{N}$ elements and the allowed transitions are "push-to-the-top" operations, [8] referred to such Gray codes as *n-length Rank-Modulation Gray Codes* (n-RMGC's), and it presented such codes traversing the entire set of permutations. In this fashion, a set of n rank-modulation cells could implement a single logical multilevel cell with $n!$ levels, where an increase of 1 in the logical cell's level corresponds to a single transition in the n-RMGC. This allows for a natural integration of rank modulation with other multilevel approaches discussed above.

We also note that generating permutations using "push-to-the-top" operations is of independent interest, called "nested cycling" in [19] (and references therein), motivated by a fast "push-to-the-top" operation[1] (cycling) available on some computer architectures.

Other recent works have explored error-correcting codes for rank modulation, where different types of errors are addressed by a careful choice of metric. In [20–22], Kendall's τ-metric was considered to model errors caused by charge-constrained noise. In contrast, the ℓ_∞-metric was used in [23, 24], as it models limited-magnitude spike errors.

In this work, we explore Gray codes for rank modulation which detect a single error, under both metrics mentioned above. The study of error-detecting Gray codes, known as *snake-in-the-box codes*, was first proposed by Kautz in [25] in the context of the hypercube with the Hamming metric and with single-bit flips as allowable transitions, due to applications for counters in asynchronous systems and for the coding of digitalized

[1]The operations described in [19] are actually mirror images of "push-to-the-top" . Furthermore, the permutation-generation scheme there is not a Gray code since it repeats some of the previously generated permutations, also making it inefficient.

analog data. These codes, and in particular their maximal achievable lengths, have since been studied extensively (see [26, 27]). They have also seen applications to other areas, such as coding theory (see [28] and references therein).

The work is organized as follows: In Chapter 2 we present basic notation and definitions. In Chapter 3 we review properties of Kendall's τ-metric, present a recursive construction of snake-in-the-box codes over the alternating groups of odd orders with rate asymptotically tending to 1, then present some upper-bounds on the size of such snake-in-the-box codes in general, and conclude by presenting auxiliary functions needed for the use of codes generated by this construction. In Chapter 4 we present a direct construction of snake-in-the-box codes of every order in the ℓ_∞-metric based on results from [8], with rates that asymptotically tend to 1. We conclude in Chapter 5 with some ad-hoc results, as well as some open questions.

Chapter 2

Preliminaries

Given a permutation σ on n elements (i.e., a bijection from and into the set $[n] = \{1, 2, \ldots, n\}$), we shall denote it by $\sigma = [\sigma(1), \sigma(2), \ldots, \sigma(n)]$. This form is called the *vector notation* for permutations. We let S_n be the symmetric group on $[n]$ (that is, the group of all permutations on $[n]$). For $\sigma, \tau \in S_n$, their composition, denoted $\sigma\tau$, is the permutation for which $\sigma\tau(i) = \sigma(\tau(i))$ for all $i \in [n]$. It is well known that $|S_n| = n!$.

Example 1. *One has precisely* 6 *ways of organizing the elements of* $[3]$ *in a row. These are:*

$$[1, 2, 3], [1, 3, 2], [2, 3, 1], [2, 1, 3], [3, 1, 2], [3, 2, 1].$$

These 6 *permutations form the group* S_3*.*

A cycle, denoted $(a_1, a_2, \ldots, a_k)$, is a permutation mapping $a_i \mapsto a_{i+1}$ for all $i \in [k-1]$, as well as $a_k \mapsto a_1$. We shall occasionally use *cycle notation* in which a permutation is described as a composition of (usually disjoint) cycles. We also recall that any permutation may be represented as a composition of cycles of size 2 (known as *transpositions*), and that the parity of the number of transpositions does not depend on the decomposition. Thus we have *even* and *odd* permutations, with positive and negative *signs*, respectively. We let A_n be the subgroup of all even permutations on $[n]$, called the *alternating group of order* n. Again, it is well known that $|A_n| = \frac{1}{2}|S_n|$.

Example 2. *Of the permutations presented in Example 1, only the following are even:*

$$[1, 2, 3], [2, 3, 1], [3, 1, 2].$$

They form the group A_3. Put in cycle notation, they are:

$$\mathrm{id}, (1,2,3), (1,3,2),$$

where id *denotes the identity permutation.*

Definition 1. *Given an* alphabet S *and a set of functions* $T \subseteq S^S = \{f \mid f : S \to S\}$*, referred to as* transitions*, a* Gray code *over* S*, using transitions* T*, of size* $M \in \mathbb{N}$*, is a sequence* $C = (c_0, c_1, \ldots, c_{M-1})$ *of* M *distinct elements of* S*, called* codewords*, such that for all* $j \in [M-1]$ *there exists* $t \in T$ *such that* $c_j = t(c_{j-1})$*.*

Example 3. *In the classic example of a Gray code [10], the codewords one considers are binary vectors* $(\mathbb{Z}/2\mathbb{Z})^n$*, and the allowable transitions are* $t_i(c) = c + e_i$*,* $i = 1, 2, \ldots n$*, where* e_i *is the vector whose* i*th coordinate equals* 1 *and the rest equal* 0*. Thus, permitted transitions are flips of a single coordinate.*
Other examples which have seen use in literature (see [11] and references therein) include generating all permutations on $[n]$ *such that consecutive permutations defer by composition with adjacent transitions (thus* $T = \{t_i : S_n \to S_n \mid i \in [n-1];\ t_i(\sigma) = \sigma(i, i+1)\}$*), generating all subsets* A *of* $[n]$ *satisfying* $|A| = k$ *such that consecutive subsets differ by a single element (by abuse of the definition,* $t_{i,j}(A) = A\Delta\{i,j\}$*, since the set of codewords isn't invariant under* $t_i : \mathcal{P}([n]) \to \mathcal{P}([n])$*), or generating all spanning trees of a graph such that consecutive trees differ by a single edge.*

Alternatively, when the original codeword c_0 is either known or immaterial, we use a slight abuse of notation in referring to the sequence of transitions $(t_{k_1}, \ldots, t_{k_{M-1}})$ generating the code (i.e., $c_j = t_{k_j}(c_{j-1})$) as the code itself.

In the above definition, when $M = |S|$ the Gray code is called *complete*. If there exists $t \in T$ such that $t(c_{M-1}) = c_0$ the Gray code is called *cyclic*, M is called its *period*, and we shall, when listing the code by its sequence of transformations, include $t_{k_M} = t$ at the end of the list. The *rate* of C, denoted $R(C)$, is defined as the ratio of bits of information transmittable using the code to those carried by the entire alphabet, i.e.,

$$R(C) = \frac{\log_2 M}{\log_2 |S|}.$$

In the context of rank modulation for flash memories, the set of transformations T comprises of "push-to-the-top" operations, first used in [8], and later also in [29, 30]. We denote by $t_i : S_n \to S_n$ the "push-to-the-top" operation on index i, i.e.,

$$t_i[a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n] = [a_i, a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n],$$

and we henceforth set $T = \{t_2, t_3, \ldots, t_n\}$. We also note that, in cycle notation,

$$t_i\sigma = \sigma\,(i, i-1, \ldots, 1)\,. \tag{2.1}$$

For ease of presentation only, we also denote by $\underline{t}_i$ the "push-to-the-bottom" operation on index $n+1-i$, i.e.,

$$\underline{t}_i[a_1, a_2, \ldots, a_{n-i}, a_{n+1-i}, a_{n+2-i}, \ldots, a_n] = [a_1, a_2, \ldots, a_{n-i}, a_{n+2-i}, \ldots, a_n, a_{n+1-i}].$$

Restricting the transformations to "push-to-the-top" operations allows fast cell programming, and eliminates overshoots (see [8]). In the context of flash memory, "push-to-the-top" operations have also been used in [30–32].

Let $\mathcal{M}$ be a metric over S defined by $d : S \times S \to \mathbb{N} \cup \{0\}$. Given a transmitted codeword $c \in C$ and its received version $\tilde{c} \in S$, we say a single error occurred if $d(c, \tilde{c}) = 1$. We are interested in Gray codes capable of detecting single errors, which we now define.

Definition 2. *Let $\mathcal{M}$ be a metric over S defined by a distance function d. A* snake-in-the-box code *over $\mathcal{M}$ and S, using transitions T, is a Gray code C over S and using T, in which for every pair of distinct elements $c, c' \in C$, $c \neq c'$, one has $d\,(c, c') \geq 2$.*

Since throughout this work our ambient space is S_n, and the transformations we use are the "push-to-the-top" operations T, we shall abbreviate our notation and call the snake-in-the-box code of size M an $(n, M, \mathcal{M})$-*snake*, or an $\mathcal{M}$-*snake*. We will be considering two metrics in the next chapters: Kendall's τ-metric, $\mathcal{K}$, and the ℓ_∞-metric, with their respective $\mathcal{K}$-snakes and ℓ_∞-snakes.

It is interesting to note that the classical definition of snake-in-the-box codes (see the survey [27]) is slightly weaker in the sense that $d(c, c') \geq 2$ is required for distinct $c, c' \in C$,

unless c and c' are adjacent in C. This, however, is a compromise due to the fact that in the classical codes over binary vectors, the transformations (which flip a single bit) always create adjacent codewords at distance 1 apart. This compromise is unnecessary in our case since, as we shall later see, the "push-to-the-top" operations allow adjacent words at distance 2 or more apart.

Chapter 3

Kendall's τ-Metric and $\mathcal{K}$-Snakes

Kendall's τ-metric [33], denoted $\mathcal{K}$, is induced by the bubble-sort distance which measures the minimal amount of adjacent transpositions required to transform one permutation into the other. For example, the distance between the permutations $[2,1,4,3]$ and $[2,4,3,1]$ is 2, as

$$[2,1,4,3] \to [2,4,1,3] \to [2,4,3,1]$$

is a shortest sequence of adjacent transpositions between the two. More formally, for $\alpha, \beta \in S_n$, as noted in [20],

$$d_{\mathcal{K}}(\alpha,\beta) = \left|\{(i,j) \mid \alpha(i) < \alpha(j) \ \wedge \ \beta(i) > \beta(j)\}\right|.$$

The metric $\mathcal{K}$ was first introduced by Kendall [33] in the study of ranking in statistics. It was observed in [20] that a bounded distance in Kendall's τ-metric models errors caused by bounded changes in charge-levels of cells in the flash memory. Error-correcting codes for this metric were studied in [20, 21, 34].

We let Kendall's τ *adjacency graph* of order $n \in \mathbb{N}$ be the graph $G_n = (S_n, E_n)$ whose vertices are the elements of the symmetric group, and $\{\alpha, \beta\} \in E_n$ whenever $d_K(\alpha,\beta) = 1$. It is well known that Kendall's τ-metric is *graphic* [35], i.e., for every $\alpha, \beta \in S_n$, $d_K(\alpha,\beta)$ equals the length of the shortest path between the two in the adjacency graph, G_n.

3.1 Construction

We begin the construction process by restricting ourselves to Gray codes using only "push-to-the-top" operations on odd indices. The following lemma provides the motivation for this restriction.

Lemma 1. *A Gray code over S_n using only "push-to-the-top" operations on odd indices is a $\mathcal{K}$-snake.*

Proof. According to Equation 2.1, a "push-to-the-top" operation on an odd index is a composition with an odd-length cycle (which is an even permutation). Thus, the codewords in a Gray code using only such operations are all with the same sign.

On the other hand, an adjacent transposition is an odd permutation, thus, flipping the sign of the permutation it acts on. It follows that in a list of codewords, all with the same sign, there are no two codewords which are adjacent in G_n, i.e., the Gray code is a $\mathcal{K}$-snake. □

Lemma 1 saves us the need to check whether a Gray code is in fact a $\mathcal{K}$-snake, at the cost of restricting the set of allowed transitions (and the size of the resulting code, although Theorem 3 and Theorem 4, presented below, work to mitigate this concern). In particular, if n is even, the last element cannot be moved.

By starting with an even permutation, and using only "push-to-the-top" operations on odd indices, we get a sequence of even permutations. Thus, throughout this part, the context of the alternating group A_{2n+1} is assumed, where $n \in \mathbb{N}$.

The construction we are about to present is recursive in nature. As a base for the recursion, we note that three consecutive "push-to-the-top" operations on the 3rd index of permutations in A_3 constitute a complete cyclic $(3, 3, \mathcal{K})$-snake:

$$C_3 = ([1, 2, 3], [3, 1, 2], [2, 3, 1]) .$$

We shall extend C_3 to the next order as a running example alongside the general construction below.

Now, assume that there exists a cyclic $(2n-1, M_{2n-1}, \mathcal{K})$-snake, C_{2n-1}, and let

$$t_{k_1}, t_{k_2}, \ldots, t_{k_{M_{2n-1}}}$$

be the sequence of transformations generating it, where k_j is odd for all $j \in [M_{2n-1}]$. We also assume that $k_1 = 2n-1$ (this requirement, while perhaps appearing arbitrary, is actually quite easily satisfied. Indeed, every sufficiently large cyclic $\mathcal{K}$-snake over S_{2n-1} must, w.l.o.g., satisfy it. We shall make it a point to demonstrate that this holds for our construction).

We fix arbitrary values for $a_0, a_1, \ldots, a_{2n-2}$ such that

$$\{a_0, a_1, \ldots, a_{2n-2}\} = [2n+1] \setminus \{1, 3\}\,. \tag{3.1}$$

For all $i \in [2n-1]$ we define

$$\sigma_0^{(i)} = [1, a_i, 3, a_{i+1}, \ldots, a_{i+2n-2}],$$

where the indices are taken modulo $2n-1$, and such that we indeed have $\sigma_0^{(i)} \in A_{2n+1}$, i.e., $\sigma_0^{(i)}$ is an even permutation (one simple way of achieving this is to choose them in ascending order).

Example 4. *We recall that C_3 is generated by the operations (t_3, t_3, t_3), which satisfy our requirement. As suggested above, we order $[5] \setminus \{1, 3\}$ in ascending order, i.e.,*

$$(a_0, a_1, a_2) = (2, 4, 5)\,.$$

We define the following permutations as starting points for our construction

$$\begin{aligned} \sigma_0^{(0)} = \sigma_0^{(3)} &= [1, 2, 3, 4, 5] \\ \sigma_0^{(1)} &= [1, 4, 3, 5, 2] \\ \sigma_0^{(2)} &= [1, 5, 3, 2, 4] \end{aligned}$$

and readily verify that they are all even.

We now define for all $i \in [2n-1]$ and $j \in [M_{2n-1}]$ the permutation

$$\sigma^{(i)}_{j(2n+1)} = \underline{t}_{k_j}\left(\sigma^{(i)}_{(j-1)(2n+1)}\right),$$

i.e., we construct cycles corresponding to a mirror view of C_{2n-1} on all but the two first elements of $\sigma^{(i)}_0$ (which, as we recall, are $(1, a_i)$).

Example 5. *In our running example, we define the following permutations:*

$$\begin{aligned}
\sigma^{(0)}_5 &= \underline{t}_3\sigma^{(0)}_0 = [1,2,4,5,3] & \sigma^{(1)}_5 &= \underline{t}_3\sigma^{(1)}_0 = [1,4,5,2,3]\\
\sigma^{(0)}_{10} &= \underline{t}_3\sigma^{(0)}_5 = [1,2,5,3,4] & \sigma^{(1)}_{10} &= \underline{t}_3\sigma^{(1)}_5 = [1,4,2,3,5]\\
\sigma^{(0)}_{15} &= \underline{t}_3\sigma^{(0)}_{10} = [1,2,3,4,5] & \sigma^{(1)}_{15} &= \underline{t}_3\sigma^{(1)}_{10} = [1,4,3,5,2]
\end{aligned}$$

$$\begin{aligned}
\sigma^{(2)}_5 &= \underline{t}_3\sigma^{(2)}_0 = [1,5,2,4,3]\\
\sigma^{(2)}_{10} &= \underline{t}_3\sigma^{(2)}_5 = [1,5,4,3,2]\\
\sigma^{(2)}_{15} &= \underline{t}_3\sigma^{(2)}_{10} = [1,5,3,2,4]
\end{aligned}$$

and resume our construction.

We now note the following properties of our construction:

Lemma 2. *Let $i, k \in [2n-1]$ and $j, l \in [M_{2n-1}]$. The following are equivalent:*

1. *The permutations $\sigma^{(i)}_{j(2n+1)}$ and $\sigma^{(k)}_{l(2n+1)}$ are cyclic shifts of each other.*
2. *$\sigma^{(i)}_{j(2n+1)} = \sigma^{(k)}_{l(2n+1)}$.*
3. *$i = k$ and $j = l$.*

Proof. First, if $\sigma^{(i)}_{j(2n+1)}$ is a cyclic shift of $\sigma^{(k)}_{l(2n+1)}$, since

$$\sigma^{(i)}_{j(2n+1)}(1) = 1 = \sigma^{(k)}_{l(2n+1)}(1)$$

then necessarily

$$\sigma^{(i)}_{j(2n+1)} = \sigma^{(k)}_{l(2n+1)}.$$

It then follows that

$$a_i = \sigma^{(i)}_{j(2n+1)}(2) = \sigma^{(k)}_{l(2n+1)}(2) = a_k,$$

hence $i = k$. Moreover, since the two permutations' last $n-1$ elements agree, and $t_{k_1}, t_{k_2}, \ldots, t_{k_{M_{2n-1}}}$ induce a Gray code, we necessarily have $j = l$.

Finally, that the last statement implies the first is trivial. □

Lemma 3. *For all $i \in [2n-1]$ it holds that*

$$\sigma^{(i)}_{M_{2n-1}(2n+1)} = \sigma^{(i)}_0.$$

Proof. The transformations $t_{k_1}, t_{k_2}, \ldots, t_{k_{M_{2n-1}}}$ induce a cyclic code, and the claim follows directly. □

Therefore we have constructed $2n-1$ cycles comprised of cyclically non-equivalent permutations (although, at this point they are not generated by "push-to-the-top" operations).

It shall now be noted that

$$\underline{t}_k = t^{2n}_{2n+1} t_{2n+2-k}.$$

Hence, if we define for all $i \in [2n-1]$, $0 \leq j < M_{2n-1}$, and $1 < m \leq 2n$, the permutations

$$\begin{aligned}\sigma^{(i)}_{j(2n+1)+1} &= t_{2n+2-k_{j+1}} \sigma^{(i)}_{j(2n+1)} \\ \sigma^{(i)}_{j(2n+1)+m} &= t^{m-1}_{2n+1} \sigma^{(i)}_{j(2n+1)+1},\end{aligned}$$

then it holds that

$$\sigma^{(i)}_{(j+1)(2n+1)} = t_{2n+1} \sigma^{(i)}_{j(2n+1)+2n}.$$

Our observation from one paragraph above means that at this point we have $2n-1$ disjoint cycles, which we conveniently denote

$$C^{(i)}_{2n+1} = \left(\sigma^{(i)}_0, \sigma^{(i)}_1, \ldots, \sigma^{(i)}_{M_{2n-1}(2n+1)-1}\right),$$

for all $i \in [2n-1]$ (for ease of notation, we let $C^{(0)}_{2n+1} = C^{(2n-1)}_{2n+1}$).

Example 6. *In our construction, the cycles we produced are shown in Figure 3.1.*

Each of the cycles is of size $(2n+1)M_{2n-1}$, is generated by "push-to-the-top" operations, and contains all cyclic shifts of elements present in our previous version of that

$$
\begin{array}{lll}
\sigma_0^{(0)} = t_5\sigma_{14}^{(0)} = [\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4},\mathbf{5}] & \sigma_0^{(1)} = t_5\sigma_{14}^{(1)} = [\mathbf{1},\mathbf{4},\mathbf{3},\mathbf{5},\mathbf{2}] & \sigma_0^{(2)} = t_5\sigma_{14}^{(2)} = [\mathbf{1},\mathbf{5},\mathbf{3},\mathbf{2},\mathbf{5}] \\
\sigma_1^{(0)} = t_3\sigma_0^{(0)} = [3,1,2,4,5] & \sigma_1^{(1)} = t_3\sigma_0^{(1)} = [3,1,4,5,2] & \sigma_1^{(2)} = t_3\sigma_0^{(2)} = [3,1,5,2,4] \\
\sigma_2^{(0)} = t_5\sigma_1^{(0)} = [5,3,1,2,4] & \sigma_2^{(1)} = t_5\sigma_1^{(1)} = [2,3,1,4,5] & \sigma_2^{(2)} = t_5\sigma_1^{(2)} = [4,3,1,5,2] \\
\sigma_3^{(0)} = t_5\sigma_2^{(0)} = [4,5,3,1,2] & \sigma_3^{(1)} = t_5\sigma_2^{(1)} = [5,2,3,1,4] & \sigma_3^{(2)} = t_5\sigma_2^{(2)} = [2,4,3,1,5] \\
\sigma_4^{(0)} = t_5\sigma_3^{(0)} = [2,4,5,3,1] & \sigma_4^{(1)} = t_5\sigma_3^{(1)} = [4,5,2,3,1] & \sigma_4^{(2)} = t_5\sigma_3^{(2)} = [5,2,4,3,1] \\
\sigma_5^{(0)} = t_5\sigma_4^{(0)} = [\mathbf{1},\mathbf{2},\mathbf{4},\mathbf{5},\mathbf{3}] & \sigma_5^{(1)} = t_5\sigma_4^{(1)} = [\mathbf{1},\mathbf{4},\mathbf{5},\mathbf{2},\mathbf{3}] & \sigma_5^{(2)} = t_5\sigma_4^{(2)} = [\mathbf{1},\mathbf{5},\mathbf{2},\mathbf{4},\mathbf{3}] \\
\sigma_6^{(0)} = t_3\sigma_5^{(0)} = [4,1,2,5,3] & \sigma_6^{(1)} = t_3\sigma_5^{(1)} = [5,1,4,2,3] & \sigma_6^{(2)} = t_3\sigma_5^{(2)} = [2,1,5,4,3] \\
\sigma_7^{(0)} = t_5\sigma_6^{(0)} = [3,4,1,2,5] & \sigma_7^{(1)} = t_5\sigma_6^{(1)} = [3,5,1,4,2] & \sigma_7^{(2)} = t_5\sigma_6^{(2)} = [3,2,1,5,4] \\
\sigma_8^{(0)} = t_5\sigma_7^{(0)} = [5,3,4,1,2] & \sigma_8^{(1)} = t_5\sigma_7^{(1)} = [2,3,5,1,4] & \sigma_8^{(2)} = t_5\sigma_7^{(2)} = [4,3,2,1,5] \\
\sigma_9^{(0)} = t_5\sigma_8^{(0)} = [2,5,3,4,1] & \sigma_9^{(1)} = t_5\sigma_8^{(1)} = [4,2,3,5,1] & \sigma_9^{(2)} = t_5\sigma_8^{(2)} = [5,4,3,2,1] \\
\sigma_{10}^{(0)} = t_5\sigma_9^{(0)} = [\mathbf{1},\mathbf{2},\mathbf{5},\mathbf{3},\mathbf{4}] & \sigma_{10}^{(1)} = t_5\sigma_9^{(1)} = [\mathbf{1},\mathbf{4},\mathbf{2},\mathbf{3},\mathbf{5}] & \sigma_{10}^{(2)} = t_5\sigma_9^{(2)} = [\mathbf{1},\mathbf{5},\mathbf{4},\mathbf{3},\mathbf{2}] \\
\sigma_{11}^{(0)} = t_3\sigma_{10}^{(0)} = [5,1,2,3,4] & \sigma_{11}^{(1)} = t_3\sigma_{10}^{(1)} = [2,1,4,3,5] & \sigma_{11}^{(2)} = t_3\sigma_{10}^{(2)} = [4,1,5,3,2] \\
\sigma_{12}^{(0)} = t_5\sigma_{11}^{(0)} = [4,5,1,2,3] & \sigma_{12}^{(1)} = t_5\sigma_{11}^{(1)} = [5,2,1,4,3] & \sigma_{12}^{(2)} = t_5\sigma_{11}^{(2)} = [2,4,1,5,3] \\
\sigma_{13}^{(0)} = t_5\sigma_{12}^{(0)} = [3,4,5,1,2] & \sigma_{13}^{(1)} = t_5\sigma_{12}^{(1)} = [3,5,2,1,4] & \sigma_{13}^{(2)} = t_5\sigma_{12}^{(2)} = [3,2,4,1,5] \\
\sigma_{14}^{(0)} = t_5\sigma_{13}^{(0)} = [2,3,4,5,1] & \sigma_{14}^{(1)} = t_5\sigma_{13}^{(1)} = [4,3,5,2,1] & \sigma_{14}^{(2)} = t_5\sigma_{13}^{(2)} = [5,3,2,4,1]
\end{array}
$$

Figure 3.1: 3 disjoint cycles in A_5: $C_5^{(0)}, C_5^{(1)}, C_5^{(2)}$. The permutations in bold are those from Example 5.

cycle. We merge these cycles into a single cycle in the following theorem.

Theorem 1. *Given a cyclic $(2n-1, M_{2n-1}, \mathcal{K})$-snake using only "push-to-the-top" operations on odd indices, and such that its first transformation is t_{2n-1}, there exists a cyclic $(2n+1, M_{2n+1}, \mathcal{K})$-snake with the same properties, whose size is*

$$M_{2n+1} = (2n-1)(2n+1)M_{2n-1}.$$

Proof. Since $k_1 = 2n-1$, it holds for all $i \in [2n-1]$ that $\sigma_1^{(i)} = t_3\sigma_0^{(i)}$, and we recall $\sigma_2^{(i)} = t_{2n+1}\sigma_1^{(i)}$. More explicitly,

$$\begin{aligned}
\sigma_1^{(i)} &= [3,1,a_i,a_{i+1},\ldots,a_{i+2n-2}] \\
\sigma_2^{(i)} &= [a_{i+2n-2},3,1,a_i,a_{i+1},\ldots,a_{i+2n-3}],
\end{aligned}$$

where, again, the indices are taken modulo $2n-1$. Thus for all $i \in [2n-2]$ we have

$$t_3\sigma_1^{(i)} = [a_i,3,1,a_{i+1},\ldots,a_{i+2n-2}] = \sigma_2^{(i+1)}$$

and $t_3\sigma_1^{(2n-1)} = \sigma_2^{(1)}$.

Let E denote the left-shift operator, and so

$$E^2 C_{2n+1}^{(i)} = \left(\sigma_2^{(i)}, \sigma_3^{(i)}, \ldots, \sigma_{M_{2n-1}(2n+1)-1}^{(i)}, \sigma_0^{(i)}, \sigma_1^{(i)}\right).$$

By the above observations we conclude that

$$C_{2n+1} = E^2 C_{2n+1}^{(0)}, E^2 C_{2n+1}^{(1)}, \ldots, E^2 C_{2n+1}^{(2n-2)}$$

is a cyclic $(2n+1, M_{2n+1}, \mathcal{K})$-snake, consisting of

$$M_{2n+1} = (2n-1)(2n+1)M_{2n-1}$$

permutations. The code C_{2n+1} obviously uses t_{2n+1}, and so some cyclic shift of it has it as its first transition (in fact, for every $i \in [2n-1]$ one has $\sigma_3^{(i)} = t_{2n+1}\sigma_2^{(i)}$, and in particular, $E^2 C_{2n+1}^{(0)}$ has t_{2n+1} as its first transition, and so does C_{2n+1}). Finally, it is easily verifiable that all "push-to-the-top" operations are on odd indices. □

Example 7. *Our running example ends with the full construction of a* $(5, 45, \mathcal{K})$*-snake,* C_5*, from Theorem 1, shown in Figure 3.2.*

$[5,3,1,2,4]$	$\sigma_2^{(0)}$	$[2,3,1,4,5]$	$\sigma_2^{(1)}$	$[4,3,1,5,2]$	$\sigma_2^{(2)}$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$[1,2,4,5,3]$	$\sigma_5^{(0)}$	$[1,4,5,2,3]$	$\sigma_5^{(1)}$	$[1,5,2,4,3]$	$\sigma_5^{(2)}$
$[4,1,2,5,3]$	$\sigma_6^{(0)}$	$[5,1,4,2,3]$	$\sigma_6^{(1)}$	$[2,1,5,4,3]$	$\sigma_6^{(2)}$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$[1,2,5,3,4]$	$\sigma_{10}^{(0)}$	$[1,4,2,3,5]$	$\sigma_{10}^{(1)}$	$[1,5,4,3,2]$	$\sigma_{10}^{(2)}$
$[5,1,2,3,4]$	$\sigma_{11}^{(0)}$	$[2,1,4,3,5]$	$\sigma_{11}^{(1)}$	$[4,1,5,3,2]$	$\sigma_{11}^{(2)}$
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
$[1,2,3,4,5]$	$\sigma_0^{(0)}$	$[1,4,3,5,2]$	$\sigma_0^{(1)}$	$[1,5,3,2,4]$	$\sigma_0^{(2)}$
$[3,1,2,4,5]$	$\sigma_1^{(0)}$	$[3,1,4,5,2]$	$\sigma_1^{(1)}$	$[3,1,5,2,4]$	$\sigma_1^{(2)}$

Figure 3.2: A $(5, 45, \mathcal{K})$-snake constructed by Theorem 1. Down arrows stand for an omitted sequence of t_5 transformations. The transition from column to column uses a single t_3 transformation..

We now turn to consider the size and rate of the constructed codes, and show that their rate asymptotically tends to 1.

Theorem 2. *The size of $\mathcal{K}$-snakes constructed in Theorem 1 behaves asymptotically as*

$$|C_{2n+1}| = M_{2n+1} = \frac{(2n)!(2n+1)!}{n!^2 \cdot 2^{2n}} \sim \frac{1}{\sqrt{\pi n}} |S_{2n+1}|,$$

which leads to an asymptotic rate of 1.

Proof. Starting from our base case of a complete cyclic $(3, 3, \mathcal{K})$-snake, we define for all $n \in \mathbb{N}$ the ratio

$$D_{2n+1} = \frac{M_{2n+1}}{(2n+1)!},$$

which is the size of our constructed code over the total size of S_{2n+1}. We note that

$$\frac{D_{2n+1}}{D_{2n-1}} = \frac{M_{2n+1} \cdot (2n-1)!}{(2n+1)! \cdot M_{2n-1}} = \frac{2n-1}{2n}.$$

Therefore, since $D_3 = \frac{1}{2}$, we have for all $2 \leq n \in \mathbb{N}$ that

$$D_{2n+1} = \frac{1}{2} \prod_{m=2}^{n} \frac{2m-1}{2m} = \frac{(2n)!}{n!^2 \cdot 2^{2n}}.$$

Using Stirling's approximation one observes

$$\begin{aligned} \lim_{n\to\infty} D_{2n+1}\sqrt{\pi n} &= \lim_{n\to\infty} \frac{(2n)!\sqrt{\pi n}}{n!^2 \cdot 2^{2n}} \\ &= \lim_{n\to\infty} \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}\sqrt{\pi n}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\right)^2 \cdot 2^{2n}} = 1. \end{aligned}$$

Moreover, one can now readily verify that

$$\lim_{n\to\infty} R(C_{2n+1}) = \lim_{n\to\infty} \frac{\log_2 M_{2n+1}}{\log_2 |S_{2n+1}|} = 1.$$

□

Section 3.2 will focus on exploring the possible size of $\mathcal{K}$-snakes in general.

Before we conclude this part, we recall that flash memory cells suffer long-time damage from erasure cycles, and therefore it is desirable to minimize the number of times such cycles are required.

A property of rank-modulation cell programming is that an erasure of an entire cell block is required only when a specific cell is to exceed its maximal permitted charge level. It is therefore of interest to analyze the rate with which our constructed codes increase the charge level of any given cell.

Repeated "push-to-the-top" operations on a given cell will result in a fast increase in that cell's charge level, and growing gaps between it and the charge levels of other cells. It is therefore most cost-economic, in the sense that it delays the need for a time-consuming erasure and reprogramming cycle, to employ a programming strategy which retains the charge levels of individual cells as balanced as possible. Such balanced Gray codes were constructed in [8].

In this part's context, this goal is achieved if and only if every two subsequent incidents in a cyclic $(2n+1, M, \mathcal{K})$-snake where a "push-to-the-top" operation is applied to a certain cell are separated by at most $2n+1$ operations on other cells. Our family of codes nearly achieves this goal:

Proposition 1. *For every permutation $\sigma \in C_{2n+1}$, in the $\mathcal{K}$-snake constructed in Theorem 1, there exists another $\sigma' \in C_{2n+1}$ such that $\sigma(1) = \sigma'(1)$, following it by no more than $2n+3$ steps.*

Proof. Recall that

$$C_{2n+1} = E^2 C^{(0)}_{2n+1}, E^2 C^{(1)}_{2n+1}, \ldots, E^2 C^{(2n-2)}_{2n+1}.$$

By the nature of our construction, for $n \geq 2$, every "push-to-the-top" operation, on all but the last rank in the code, appears either as part of the pattern

$$\ldots, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}, t_i, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}, \ldots$$

or as

$$\ldots, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}, t_3, t_3, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}, \ldots$$

It is therefore the case that there exist $0 \leq k \leq 2n$ and $j \in [n]$ such that the transformations used in C_{2n+1} after σ are of the following two forms:

1. $\underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{k}, t_{2j+1}, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}$

2. $\underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{k}, t_3, t_3, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}$

In the second case, one notes:

$$\sigma(1) = \begin{cases} t_{2n+1}^{2n-1} t_3^2 \sigma(1) & k = 0 \\ t_3^2 t_{2n+1} \sigma(1) & k = 1 \\ t_3 t_{2n+1}^2 \sigma(1) & k = 2 \\ t_{2n+1}^{2n+1-k} t_3^2 t_{2n+1}^k \sigma(1) & k > 2. \end{cases}$$

Finally, in the first case, we note that

$$\sigma(1) = \begin{cases} t_{2n+1}^{2n-k} t_{2j+1} t_{2n+1}^k \sigma(1) & k < 2j+1 \\ t_{2j+1} t_{2n+1}^k \sigma(1) & k = 2j+1 \\ t_{2n+1}^{2n+1-k} t_{2j+1} t_{2n+1}^k \sigma(1) & k > 2j+1. \end{cases}$$

□

It is of interest to note that, of all cases discussed in the last proof, the second case where $k > 2$ is the only situation in which another instance of programming to the specific cell fails to occur in $2n+2$ steps, i.e., for the large majority of cases (in all but $\frac{2n-1}{M_{2n+1}}$ of them), the construction of Theorem 1 yields optimally-behaving codes in this respect.

3.2 Bounds on $\mathcal{K}$-Snakes

We now turn our attention to bounding the parameters of $\mathcal{K}$-snakes. We begin by noting a simple upper bound on the size of $\mathcal{K}$-snakes.

Theorem 3. *If C is an $(n, M, \mathcal{K})$-snake then*

1. $M \leq \frac{1}{2} |S_n|$.
2. $M = \frac{1}{2} |S_n|$ *if and only if for all* $\{\alpha, \beta\} \in E_n$ *it holds that* $\alpha \in C$ *or* $\beta \in C$.

Proof. Every $\alpha \in S_n$ has exactly $(n-1)$ neighbors in G_n. When we sum the edges for every vertex in G_n, each edge in E_n is counted precisely twice, hence

$$|E_n| = \frac{n-1}{2} \cdot |S_n| = \frac{n!(n-1)}{2}.$$

On the other hand, for every $\alpha, \beta \in C$ and $e_1, e_2 \in E_n$ such that $\alpha \in e_1$ and $\beta \in e_2$ clearly $e_1 \neq e_2$. It follows that there are no less than $M(n-1)$ distinct edges in E_n. Hence

$$M \leq \frac{1}{2} |S_n| .$$

Finally, we note that $M = \frac{1}{2} |S_n|$ iff $M(n-1) = |E_n|$, iff every edge in E_n contains a (unique) element of C. □

It is worth mentioning, at this point, that this upper-bound might not be tight. Indeed, we know by Theorem 2 that

$$\frac{M_{2n+1}}{\frac{1}{2} |S_{2n+1}|} \sim \frac{2}{\sqrt{\pi n}},$$

and no constructions are currently known which attain the upper bound, except for the trivial case of C_3.

The codes we constructed in the previous part use only "push-to-the-top" operations on odd indices. We would now like to show that using even a single "push-to-the-top" operation on an even index can never result in a code attaining the bound of Theorem 3 with equality. We first require a simple lemma.

Lemma 4. *Let C be a $\mathcal{K}$-snake over S_n. If $\sigma, \sigma' \in C$ and there exists a path in G_n of odd length between them, then that path contains an edge both of whose endpoints are not in C.*

Proof. Consider such a path of odd length in G_n, connecting σ and σ'. Now color the vertices of C black, and those of $S_n \setminus C$ white. Since C is a $\mathcal{K}$-snake, no edge in E_n has both its ends colored black. In the path above the vertices cannot alternate in color since σ and σ' are colored black and the path has odd length. It follows that there is an edge in the path with both ends colored white, as claimed. □

With this lemma in hand, we can now further bound the size $\mathcal{K}$-snakes employing a "push-to-the-top" operation on an even index.

Theorem 4. *If an $(n, M, \mathcal{K})$-snake C contains a "push-to-the-top" operation on an even index then*

$$M \leq \frac{1}{2}\left|S_n\right| - \Theta(n) < \frac{1}{2}\left|S_n\right|.$$

Proof. Let $C = (\sigma_1, \ldots, \sigma_M)$. We take $i \in [M-1]$ such that $\sigma_{i+1} = t_{2m}(\sigma_i)$, where $2m \in [n]$. Then σ_i and σ_{i+1} have different signs. We will also find it convenient to denote

$$r = \sigma_i(2m) \in [n].$$

We shall construct as many distinct paths in Kendall's τ adjacency graph G_n connecting σ_i with σ_{i+1}, knowing they must all have odd lengths, and therefore by Lemma 4 they each contain an edge completely disjoint from C. We will then show that these edges are all distinct, allowing us to improve upon the bound of Theorem 3.

One natural such path is generated by subsequently applying to σ_i the adjacent transpositions $(j, j+1)$ for $j = 2m-1, 2m-2, \ldots, 1$. By taking more care before applying these transpositions, we shall arrive at more paths.

Consider the set of adjacent transpositions that do not involve the index $2m$, namely

$$T = \{(j, j+1) \mid j \in [n-1] \setminus \{2m-1, 2m\}\}.$$

For every subset $B \subseteq T$ of size $|B| \leq 2$, we generate a new permutation ω^B by applying to σ_i the elements of B (in some arbitrary order, say from smallest to largest indices). Naturally, for two distinct such subsets, B and B', we have $\omega^B \neq \omega^{B'}$, but still $\omega^B(2m) = r = \omega^{B'}(2m)$.

We can now apply to ω^B the aforementioned transpositions in the following way:

$$\begin{aligned}
\omega_0^B &= \omega^B \\
\omega_j^B &= \omega_{j-1}^B(2m-j, 2m-j+1); \quad j \in [2m-1],
\end{aligned}$$

and for every choice of subset we have $\omega_{2m-1}^B(1) = r$. Clearly, we can generate σ_{i+1}

from ω^B_{2m-1} by reversing the effect of B's elements (the actual transpositions required are altered by the change of index for r, but all other elements retain their relative positions with respect to each other. Formally, we need to apply the elements of B in the reverse order, but whenever $(j, j+1) \in B$ such that $j < 2m$ we instead apply the adjacent transposition $(j+1, j+2)$).

Now, note that if $\omega^B_k = \omega^{B'}_l$ then in particular

$$2m - k = (\omega^B_k)^{-1}(r) = (\omega^{B'}_l)^{-1}(r) = 2m - l,$$

hence $k = l$. Therefore, the induced permutation on $[n]\backslash\{r\}$ agrees as well. This, however, is impossible unless $B = B'$. Hence, any two paths of this sort can only intersect in the first step of obtaining ω^B from σ_i (or the last step from ω^B_{2m-1} to σ_{i+1}), i.e., in the first (or last) edge of the path.

Finally, by Lemma 4 each path hereby described contains an edge disjoint from C. Note that it cannot be its first or last edge (since $\sigma_i, \sigma_{i+1} \in C$), hence these edges are all distinct. It follows (in the same manner used in the proof of Theorem 3) that, where N denotes the number of subsets of T with cardinality 2 or less, we have

$$M(n-1) \leq |E_n| - N = \frac{n-1}{2} |S_n| - N,$$

and naturally $N = \binom{n-3}{2} + (n-3) + 1 = \Theta(n^2)$.

□

Before concluding this section, we note that the upper-bound of Theorem 4 is still higher than M_{2n+1}, the size of codes generated by the construction of Theorem 1. See Chapter 5 for some ad-hoc results of codes with optimal sizes.

3.3 Successor Calculation and Ranking Algorithms

We now turn to present algorithms associated with the codes we constructed in the previous sections. The algorithms are brought here for completeness of presentation, and are straightforward derivations from the construction. We shall, therefore, only provide

an intuitive sketch of correctness for them, as we shall later do in the section corresponding to l_∞-snakes.

In order to use the codes described in Theorem 1 in the implementation of a logic cell (with M_{2n+1} levels), importance is known to the ability of efficiently increasing the cell's level. That is, one needs to know, for every given permutation in the code, the appropriate "push-to-the-top" operation required to produce the subsequent permutation.

For the code C_{2n+1} from Theorem 1, the function $\texttt{Successor}_{\mathcal{K}}\left(n, [b_1, \ldots, b_{2n+1}]\right)$ takes as input a permutation in the code, and returns as output the index i of the required transformation t_i. It is assumed throughout this part that the elements $\{a_i\}_{i=0}^{2n-2}$ from Equation 3.1, used in our construction, are known, and we will denote them with superscript (n) to indicate order when it is not clear from context. Furthermore, we require a function

$$\texttt{Ind}_n(b) : [2n+1] \setminus \{1,3\} \to [0, 2n-2]$$

which returns the unique index such that $a_{\texttt{Ind}_n(b)} = b$. We assume $\texttt{Ind}_n$ runs in $O(1)$ time[1]. One possible way, among many, of achieving this is by defining:

$$a_i^{(n)} = \begin{cases} 2 & i = 0 \\ i+3 & i \geq 1 \end{cases} \qquad \texttt{Ind}_n(b) = \begin{cases} 0 & b = 2, \\ b-3 & b \geq 4. \end{cases}$$

Finally, we naturally assume validity of the input in all procedures.

Our strategy will be to identify the vertices in C_{2n+1} which require a transformation other than t_{2n+1}. Those are either permutations with leading 1's (those on which we initially performed "push-to-the-bottom" operations in our construction), or the last permutation in each $E^2 C_{2n+1}^{(j)}$. In the latter case we need only apply t_3, where the former requires translation of the $a_i^{(n)}$'s according to their respective positions in the originating permutation of each $C_{2n+1}^{(j)}$, and a recursive run of $\texttt{Successor}_{\mathcal{K}}$ to determine the correct "push-to-the-bottom" operation to be performed.

It shall be noted at this point that a degree of freedom exists in the cyclic shift of C_{2n-1} one applies to construct each $C_{2n+1}^{(j)}$ (one only needs to confirm that the first

[1]Though the integers used throughout are of magnitude $O(n)$, and so may require $O(\log n)$ bits to represent, we tacitly assume (as in [8]) all simple integer operations, e.g., assignment, comparison, addition, etc., to take $O(1)$ time.

"push-to-the-top" operation shall be on the last index). This shift shall be denoted by the following bijection for every order $n \in \mathbb{N}$ and index $j \in [2n-1]$:

$$ {}^{n}_{j}\!\downarrow : \{3\} \cup \left\{ a_i^{(n)} \right\}_{i \neq j} \longrightarrow [2n-1], $$

defined such that the "push-to-the-bottom" operation applied to

$$ \left[1, a_j^{(n)}, b_1, \ldots, b_{2n-1} \right] \in C_{2n+1}^{(j)} $$

matches the "push-to-the-top" operation applied in C_{2n-1} to

$$ \left[{}^{n}_{j}\!\downarrow b_{2n-1}, {}^{n}_{j}\!\downarrow b_{2n-2}, \ldots, {}^{n}_{j}\!\downarrow b_1 \right]. $$

We shall further denote its inverse as ${}^{n}_{j}\!\uparrow$. These two bijections can be implemented in $O(1)$ time, for example, by taking as a starting point C_{2n-1}'s $(2n-4)$-ranked permutation

$$ \left[a_0^{(n-1)}, \ldots, a_{2n-4}^{(n-1)}, 3, 1 \right], $$

and defining accordingly

$$ {}^{n}_{j}\!\downarrow b = \begin{cases} 1 & b = 3 \\ 3 & \texttt{Ind}_n(b) = j+1 \\ a^{(n-1)}_{(j-\texttt{Ind}_n(b)-1) \bmod (2n-1)} & \text{otherwise,} \end{cases} \tag{3.2} $$

where $\texttt{Ind}_n(b) = j+1$ is checked modulo $2n-1$, as well as

$$ {}^{n}_{j}\!\uparrow b = \begin{cases} 3 & b = 1 \\ a^{(n)}_{(j+1) \bmod (2n-1)} & b = 3 \\ a^{(n)}_{(j-\texttt{Ind}_n(b)-1) \bmod (2n-1)} & \text{otherwise.} \end{cases} \tag{3.3} $$

Lemma 5. $\texttt{Successor}_{\mathcal{K}}$ *runs in* $O(1)$ *amortized time.*

Proof. We first note that by the nature of our construction the element 1 appears in the

Function $\texttt{Successor}_{\mathcal{K}}\left(n, [b_1, \ldots, b_{2n+1}]\right)$

input : $n \in \mathbb{N}$, A permutation $[b_1, \ldots, b_{2n+1}] \in C_{2n+1}$
output : An odd $i \in \{3, \ldots, 2n+1\}$ that determines the transition t_i to the next permutation in C_{2n+1}

if $n = 1$ **then**
  return 3
if $b_1 = 3$ *and* $b_2 = 1$ *and* $\forall 3 \leq i \leq 2n : (\texttt{Ind}_n(b_{i+1}) - \texttt{Ind}_n(b_i)) \equiv 1 \pmod{2n-1}$ **then**
  return 3
if $b_1 = 1$ **then**
  $j \leftarrow \texttt{Ind}_n(b_2)$
  $i \leftarrow \texttt{Successor}_{\mathcal{K}}\left(n-1, \left[{}^{n}_{j}\!\downarrow b_{2n+1}, {}^{n}_{j}\!\downarrow b_{2n}, \ldots, {}^{n}_{j}\!\downarrow b_3\right]\right)$
  return $2n + 2 - i$
return $2n + 1$

leading index precisely $(2n-1) \cdot M_{2n-1}$ times, which constitutes $\frac{1}{2n+1}$ of the code's size. The pair $(3, 1)$ leads no more (and in fact strictly less) permutations.

Therefore, if we let E_n denote the expected number of steps performed by $\texttt{Successor}_{\mathcal{K}}$ when called on input of length $2n+1$, then we note the recursive connection

$$\begin{aligned} E_n &\leq O(1) + \frac{1}{2n+1} O(n) + \frac{1}{2n+1} \left(O(n) + E_{n-1}\right) \\ &= O(1) + \frac{1}{2n+1} E_{n-1}. \end{aligned}$$

Developing this inequality recursively, there exists $L \in \mathbb{N}$ such that

$$\begin{aligned} E_n \leq & L + \frac{1}{2n-1} E_{n-1} \\ & \leq \left(1 + \frac{1}{2n-1}\right) L + \frac{1}{(2n-1)(2n-3)} E_{n-2} \leq \\ & \vdots \\ & \leq \left(1 + \frac{1}{2n-1} + \frac{n-2}{(2n-1)(2n-3)}\right) L + \frac{n! 2^n}{(2n)!} E_1, \end{aligned}$$

and so $E_n = O(1)$. □

To use C_{2n+1} in the implementation of a logic cell, one also needs a method of computing a given permutation's rank in the code. We implement the function $\texttt{Rank}_{\mathcal{K}}\left([b_1, \ldots, b_{2n+1}]\right)$

Function $\texttt{Rank}_{\mathcal{K}}\left([b_1, \ldots, b_{2n+1}]\right)$

input : A permutation $[b_1, \ldots, b_{2n+1}] \in C_{2n+1}$
output : The rank $k \in \{0, \ldots, M_{2n+1} - 1\}$ associated with the given permutation in C_{2n+1}

if $n = 1$ **then**
  return $3 - b_2$
$i \leftarrow \min\{l \in [2n+1] \mid b_l = 1\}$
$j \leftarrow \texttt{Ind}_n\left(b_{(i \bmod (2n+1))+1}\right)$
for $l \leftarrow 1$ **to** $2n - 1$ **do**
  $c_l \leftarrow {}^{n}_{j}\!\downarrow b_{((i-l-1) \bmod (2n+1))+1}$
$r \leftarrow \left(\texttt{Rank}_{\mathcal{K}}\left([c_1, \ldots, c_{2n-1}]\right) - r^{(j)}_{2n+1}\right) \bmod M_{2n-1}$
$rn \leftarrow ((2n+1)(r-1) - 1 + ((i-2) \bmod (2n+1))) \bmod ((2n+1)M_{2n-1})$
return $(2n+1)M_{2n-1} \cdot j + rn$

which receives as input a permutation $[b_1, \ldots, b_{2n+1}] \in C_{2n+1}$ and returns its rank in

$$C_{2n+1} = E^2C^{(0)}_{2n+1}, E^2C^{(1)}_{2n+1}, \ldots, E^2C^{(2n-2)}_{2n+1},$$

in the order indicated by that notation. The assumptions made in the previous part are still in effect. Moreover, we will require knowledge of the cyclic shift of C_{2n-1} used in the construction of each $C^{(j)}_{2n+1}$, which we retain in the form of $r^{(j)}_{2n+1}$, the rank of permutation in C_{2n-1} which was chosen as a starting point. For example, in the method suggested by Equation 3.2 and Equation 3.3, we have

$$r^{(j)}_{2n+1} = 2n - 4$$

for all $j \in [2n-1]$.

We use the following method: first identify the position of 1 in the permutation, and the following element, which gives us both the subcode the permutation belongs to and the cyclic shift in our mock "push-to-the-bottom" operation. Armed with that information we then scan the permutation backwards and translate the $a^{(n)}_j$'s indices according to the subcode in the same way we did in $\texttt{Successor}_{\mathcal{K}}$. After that, a recursive run of $\texttt{Rank}_{\mathcal{K}}$ will give us the permutation's position in its subcode, which we will combine with the cyclic shift to produce the correct rank, taking $r^{(j)}_{2n+1}$ into account and remembering that C_{2n+1} is constructed of the $E^2C^{(j)}_{2n+1}$'s rather than the $C^{(j)}_{2n+1}$'s.

Lemma 6. *The function* $\texttt{Rank}_{\mathcal{K}}$ *operates in* $O(n^2)$ *steps.*

Proof. We note that $\texttt{Rank}_{\mathcal{K}}$ performs $O(n)$ operations before calling upon itself with an order reduced by one. It therefore operates in $O(n^2)$ time. □

Unranking permutations, i.e., the process of assigning to a given rank in $[0, M_{2n+1}-1]$ the corresponding permutation in the C_{2n+1}, might also be needed if one requires the logic cell to perform as more than a counter. We implement a function $\texttt{Unrank}_{\mathcal{K}}(n,k)$ which returns as output the k-ranked permutation in C_{2n+1}.

Naturally, all assumptions made above still hold. We will follow the same general method used for $\texttt{Rank}_{\mathcal{K}}$, i.e., we shall compute $j \in [2n-1]$ such that the given rank belongs to $\sigma \in E^2 C^{(j)}_{2n+1}$, then adjust the rank to indicate the correct position in $C^{(j)}_{2n+1}$. It will then remain to compute the correct permutation in the "push-to-the-bottom" cycle using a recursive run, and shift it the required number of times.

Function $\texttt{Unrank}_{\mathcal{K}}\,(n,k)$

input : $n \in \mathbb{N}$; rank $k \in [0, M_{2n+1}-1]$
output : The permutation $[b_1,\ldots,b_{2n+1}]$ which is kth in C_{2n+1}

if $n=0$ **then**
  return $[1]$
$j \leftarrow \left\lfloor \frac{k}{(2n+1)\cdot M_{2n-1}} \right\rfloor$
$pos \leftarrow k \bmod ((2n+1)M_{2n-1})$
$perm \leftarrow \left(\left\lfloor \frac{pos+1}{2n+1} \right\rfloor + 1 + r^{(j)}_{2n+1} \right) \bmod M_{2n-1}$
$shift \leftarrow (pos+2) \bmod (2n+1)$
$[c_1,\ldots,c_{2n-1}] \leftarrow \texttt{Unrank}_{\mathcal{K}}(n-1, perm)$
return $t^{shift}_{2n+1}\left[1, a^{(n)}_j, {}^{n}_{j}\!\uparrow c_{2n-1}, {}^{n}_{j}\!\uparrow c_{2n-2}, \ldots, {}^{n}_{j}\!\uparrow c_1\right]$

Lemma 7. *The function* $\texttt{Unrank}_{\mathcal{K}}$ *operates in* $O(n^2)$ *steps as well.*

Proof. Follows exactly the same lines as our proof to Lemma 6. □

Chapter 4

The ℓ_∞-Metric and ℓ_∞-Snakes

The ℓ_∞-metric is induced on S_n by the embedding in $\mathbb{Z}^n$ implied by the vector notation. More precisely, for $\alpha, \beta \in S_n$ one defines

$$d_\infty(\alpha, \beta) = \max_{i \in [n]} |\alpha(i) - \beta(i)| .$$

We use the ℓ_∞-metric to model a different kind of noise-mechanism than that modeled by Kendall's τ-metric, namely spike noise. In this model, the rank of each memory cell is assumed to have been changed by a bounded amount (see [23]).

Error-correcting and -detecting codes in S_n for the ℓ_∞-metric are referred to in [23] as *limited-magnitude rank-modulation codes* (LMRM codes). In that paper, constructions of such codes achieving non-vanishing normalized distance and rate are presented. Moreover, bounds on the size of optimal LMRM codes are proven. In particular, it has been shown [23, Th. 20] that if C is an $(n, M, 2)$-LMRM then

$$M \leq \frac{n!}{2^{\lfloor n/2 \rfloor}}.$$

Using a simple translation to an extremal problem involving permanents of $(0, 1)$-matrices (see [36]), this is also the best possible bound using the set-antiset method. For our needs, it follows that the size of every n-length ℓ_∞-snake is bounded by this term. We shall present a construction of ℓ_∞-snakes achieving this upper-bound by a factor of $\left\lfloor \frac{n}{2} \right\rfloor 2^{\lceil n/2 \rceil}$, which we will show achieves an asymptotic rate of 1.

4.1 Construction

In order to use the code constructions presented in [8], we first prove the following lemma.

Lemma 8. *Both constructions in [8, Th. 4,7], when applied recursively, yield complete cyclic n-RMGC's containing both "push-to-the-top" operations t_2 and t_n.*

Proof. The proposition was, while not fully stated, actually proven in [8, Th. 4].

For [8, Th. 7], we shall assume that the recursive process was applied to a length-$(n-1)$ Gray code satisfying these conditions (as is the case with the base example given in that article). The resulting code uses t_n by definition. Moreover, since the original code used t_{n-1}, the resulting code uses $t_{n-(n-1)+1} = t_2$. □

This lemma now allows for the construction of a basic building block which we will later use.

Lemma 9. *Let $\{a_j\}_{j=1}^n$, $n \geq 2$, be a set of integers of the same parity. Let*

$$\sigma = [x, a_1, a_2, \ldots, a_n, b_{n+2}, b_{n+3}, \ldots, b_m] \in S_m$$

be a permutation such that the parity of x differs from that of the elements of $\{a_j\}_{j=1}^n$. Then there exists a (non-cyclic) $(m, n + (n-1)!, \ell_\infty)$-snake starting with σ and ending with the permutation

$$t_2 t_{n+1}^n(\sigma) = [a_2, a_1, a_3, a_4, \ldots, a_n, x, b_{n+2}, b_{n+3}, \ldots, b_m] .$$

Proof. Let $\sigma_0, \ldots, \sigma_{n+(n-1)!-1}$ denote the codewords of the claimed code, and denote by $t_{k_1}, \ldots, t_{k_{n+(n-1)!-1}}$ the list of transformations generating it.

We set $\sigma_0 = \sigma$. For all $i \in [n]$ we let $\sigma_i = t_{n+1}^i(\sigma)$, i.e., $t_{k_i} = t_{n+1}$. Quite clearly, any two of these $n+1$ permutations are at ℓ_∞-distance at least 2 apart, since the a_j's share parity.

Now, by Lemma 8 there exists a complete cyclic $(n-1)$-RMGC starting with σ_n, with its last operation being t_2. We therefore let $t_{k_{n+i}}$ for $i \in [(n-1)!]$ represent that code, hence $t_{k_{n+(n-1)!}} = t_2$ and $\sigma_{n+(n-1)!} = \sigma_n$ (we then, obviously, omit the last trans-

formation as well as the repeated codeword $\sigma_{n+(n-1)!}$). These $(n-1)!$ permutations, $\sigma_n, \ldots, \sigma_{n+(n-1)!-1}$, also represent an ℓ_∞-snake, for the same reason.

Finally, take $0 \leq k < n$ and $0 \leq l < (n-1)!$, and observe σ_k and σ_{n+l}. Suppose $d_\infty(\sigma_k, \sigma_{n+l}) \leq 1$. Then in particular $|a_{n-k} - x| = 1$. Moreover, if $k = n-1$ then $|x - a_n| = 1$, but then a_n's position in σ_k correlates to one of $\{a_j\}_{j=1}^{n-1}$ in σ_{n+l}, in contradiction. Therefore $k \leq n-2$, but then a_n's position in σ_{n+l} (nth from left) correlates to that of a_{n-k-1} in σ_k, where $1 \leq n-k-1 \leq n-1$, again in contradiction. This concludes our proof. □

Example 8. *We shall start this example with the permutation*

$$\sigma = [1, 2, 4, 6, 3, 5]\,.$$

We will also require a complete 2-RMGC, which clearly comprises of two subsequent t_2 operations. We are now ready to present a (non-cyclic) $(6, 3+2!, l_\infty)$-snake:

$$\begin{bmatrix}1\\2\\4\\6\\3\\5\end{bmatrix} \underset{t_4}{\rightarrow} \begin{bmatrix}6\\1\\2\\4\\3\\5\end{bmatrix} \underset{t_4}{\rightarrow} \begin{bmatrix}4\\6\\1\\2\\3\\5\end{bmatrix} \underset{t_4}{\rightarrow} \begin{bmatrix}2\\4\\6\\1\\3\\5\end{bmatrix} \underset{t_2}{\rightarrow} \begin{bmatrix}4\\2\\6\\1\\3\\5\end{bmatrix}$$

An additional t_2 operation is called for by our complete 2-RMGC, but we omit it. We also note that any permutation on the odd *element in this example will not change its properties as an l_∞-snake.*

Having this building block in hand, we continue to describe a construction of a cyclic ℓ_∞-snake. The construction follows by dividing the ranks in a length-n permutation into even and odd elements, and covering permutations on each half separately.

Theorem 5. *For all $4 \leq n \in \mathbb{N}$ there exists an (n, M, ℓ_∞)-snake of size*

$$M = \left\lceil \frac{n}{2} \right\rceil ! \left(\left\lfloor \frac{n}{2} \right\rfloor + \left(\left\lfloor \frac{n}{2} \right\rfloor - 1 \right) ! \right).$$

Proof. To simplify notations, we start by noting that $[n]$ has $p = \left\lceil \frac{n}{2} \right\rceil$ odd elements and $q = \left\lfloor \frac{n}{2} \right\rfloor$ even ones. We shall use that notation throughout this proof.

Using [8, Th. 4,7] we take a complete cyclic p-RMGC using the operations

$$t_{\alpha(1)}, t_{\alpha(2)}, \ldots, t_{\alpha(p!)}.$$

Moreover, we use Lemma 9 to come by a (q, M_q, ℓ_∞)-snake of size $M_q = q + (q-1)!$ given by the operations

$$t_{\beta(1)}, t_{\beta(2)}, \ldots, t_{\beta(q+(q-1)!-1)}.$$

As the origin for the code we construct we use

$$\sigma_0 = [1, 2, 4, \ldots, 2q, 3, \ldots, 2p-1]\,.$$

For all $i \in [p!]$ and $j \in [q + (q-1)! - 1]$ we define the sequence of transformations generating the code as

$$\begin{aligned} t_{k_{(i-1)(q+(q-1)!)+j}} &= t_{\beta(j)} \\ t_{k_{i(q+(q-1)!)}} &= t_{\alpha(i)+q} \end{aligned}$$

and where, naturally, the codewords satisfy $\sigma_i = t_{k_i}(\sigma_{i-1})$.

We start by noting that, for all $i \in [p!]$, the permutation $\sigma_{(i-1)(q+(q-1)!)}$ satisfies the requirements of Lemma 9 as a simple matter of induction. It follows that for all $i \in [p!]$ the permutations

$$\left\{\sigma_{(i-1)(q+(q-1)!)+1}, \sigma_{(i-1)(q+(q-1)!)+2}, \ldots, \sigma_{i(q+(q-1)!)-1}\right\}$$

are at ℓ_∞-distance of at least 2 apart.

Furthermore, for $i, i' \in [p!]$, $i < i'$, since the code generated by $t_{\alpha(1)}, t_{\alpha(2)}, \ldots, t_{\alpha(p!)}$ is indeed a Gray code, we are assured that for all $0 \leq j, j' \leq q + (q-1)! - 1$ the last $p-1$ elements of both $\sigma_{(i-1)(q+(q-1)!)+j}$ and $\sigma_{(i'-1)(q+(q-1)!)+j'}$ are all odd and represent two distinct permutations, hence

$$d_\infty\left(\sigma_{(i-1)(q+(q-1)!)+j}, \sigma_{(i'-1)(q+(q-1)!)+j'}\right) \geq 2.$$

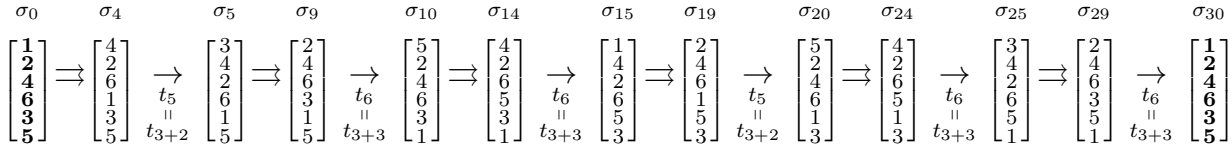

Figure 4.1: A $(6, 3!(3+2!), l_\infty)$-snake constructed by Theorem 5. Double-arrows stand for an omitted sequence of transitions generating Example 8, i.e., t_4, t_4, t_4, t_2.

Finally, we note that

$$t_{\alpha(p!)}\left(\sigma_{p!(q+(q-1)!)-1}\right) = \sigma_0,$$

since the code provided by $t_{\alpha(1)}, t_{\alpha(2)}, \ldots, t_{\alpha(p!)}$ is cyclic and $o(t_2) = 2$ divides $p!$. □

Example 9. *For this example, using an order of* 6 *as the last example (i.e.,* $p = q = 3$*), we refer to [8, Th. 4,7] for a complete cyclic* 3*-RMGC as well as the aforementioned* 2*-RMGC. One such is created by the transitions:*

$$t_2, t_3, t_3, t_2, t_3, t_3.$$

Moreover, we have our $(6, 3+2!, l_\infty)$*-snake from the last example, generated by (recall that it's not cyclic):*

$$t_4, t_4, t_4, t_2.$$

In Figure 4.1 we start our cyclic $(6, 3!(3+2!), l_\infty)$*-snake in the same permutation as we did the last example, and use the generating transitions of the code presented in the last example as a building block.*

One sees that we indeed have a cyclic l_∞*-snake of size* 30*.*

We note that by switching the roles of odd and even numbers in Theorem 5 we can construct an (n, M, ℓ_∞)-snake of size

$$M = \left\lfloor \frac{n}{2} \right\rfloor ! \left(\left\lceil \frac{n}{2} \right\rceil + \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right)! \right).$$

However, the resulting code is strictly smaller for odd n.

Theorem 6. *The* ℓ_∞*-snakes constructed in Theorem 5 have an asymptotic-rate of* 1*.*

Proof. Let C_n denote the ℓ_∞-snake of length n constructed by Theorem 5. Using the crude

$$\left(\frac{n}{e}\right)^n \leq n! \leq n^n$$

the proof is a matter of simple calculation:

$$\begin{aligned}
\lim_{n\to\infty} R(C_n) &= \lim_{n\to\infty} \frac{\log_2\left(\left\lceil\frac{n}{2}\right\rceil!\left(\left\lfloor\frac{n}{2}\right\rfloor + \left(\left\lfloor\frac{n}{2}\right\rfloor - 1\right)!\right)\right)}{\log_2(n!)} \\
&\geq \lim_{n\to\infty} \frac{2\log_2\left(\left(\left\lfloor\frac{n}{2}\right\rfloor - 1\right)!\right)}{\log_2(n!)} \\
&\geq \lim_{n\to\infty} \frac{(n-4)\log_2\left(\frac{n-4}{2e}\right)}{n\log_2 n} = 1.
\end{aligned}$$

□

4.2 Successor Calculation and Ranking Algorithms

Finding the correct "push-to-the-top" operation to propagate a given permutation to the following one is naturally dependent upon one's ability to do the same with the $\left\lceil\frac{n}{2}\right\rceil$- and $\left(\left\lfloor\frac{n}{2}\right\rfloor - 1\right)$-RMGC's used in our construction. We therefore assume to have the function $\texttt{Succ}([a_1, a_2, \ldots, a_n])$ which accepts as input a permutation $[a_1, a_2, \ldots, a_n] \in S_n$ and returns the correct transformation used in the codes we used. Furthermore, we assume to have the function $\texttt{Rn}([a_1, a_2, \ldots, a_n])$ which returns the respective rank of the input permutation in that code, where the identity permutation is assumed to have rank zero. Finally, we shall use an auxiliary function $\texttt{sw} : S_n \to S_n$ defined by $\texttt{sw}(\sigma) = (1,2) \circ \sigma$ (which naturally operates in $O(n)$ steps).

The function $\texttt{Successor}_\infty([a_1, \ldots, a_n])$ then returns as output the index i of the required transformation t_i to produce the subsequent permutation in the code from $[a_1, \ldots, a_n]$. It operates by considering the following cases: in each block of Lemma 9 one computes the proper index by propagating the leading element of odd rank as long as that is needed, then applying `Succ` to the permutation on the elements of even ranks (where one distinguishes between blocks in which $2, 4$ were switched). Only the last permutation of each block calls for applying `Succ` to the permutation on the elements of odd ranks.

Function $\texttt{Successor}_\infty\left([a_1,\ldots,a_n]\right)$

input : A permutation $[a_1,a_2,\ldots,a_n]$
output : $i\in\{2,3,\ldots,n\}$ that determines the transition t_i to the next permutation in the ℓ_∞-snake from Theorem 5

1 $q\leftarrow\left\lfloor\frac{n}{2}\right\rfloor$; $p\leftarrow\left\lceil\frac{n}{2}\right\rceil$
2 **if** $a_{q+1}\equiv 0 \pmod 2$ **then**
3 **return** $q+1$
4 **if** $\texttt{Rn}(\left[\frac{a_{q+1}+1}{2},\ldots,\frac{a_n+1}{2}\right])\equiv 0 \pmod 2$ **then**
5 **if** $[a_1,\ldots,a_q]=[4,2,6,\ldots,2q]$ **then**
6 **return** $q+\texttt{Succ}\left(\left[\frac{a_{q+1}+1}{2},\ldots,\frac{a_n+1}{2}\right]\right)$
7 **return** $\texttt{Succ}\left(\left[\frac{a_1}{2},\ldots,\frac{a_q}{2}\right]\right)$
8 **if** $[a_1,\ldots,a_q]=[2,4,\ldots,2q]$ **then**
9 **return** $q+\texttt{Succ}\left(\left[\frac{a_{q+1}+1}{2},\ldots,\frac{a_n+1}{2}\right]\right)$
10 **return** $\texttt{Succ}\left(\texttt{sw}\left(\left[\frac{a_1}{2},\ldots,\frac{a_{q-1}}{2}\right]\right)\right)$

Lemma 10. *If the functions* $\texttt{Succ},\texttt{Rn}$ *operate in* L_n,M_n *steps respectively in the average case, then* $\texttt{Successor}_\infty$ *has an average runtime of* $O\left(n+L_{q-1}+M_p\right)$.

Proof. We partition our proof by return cases. $\texttt{Successor}_\infty$ exits at line 3 in precisely $\frac{q}{q+(q-1)!}$ of cases, in which case it returns within a fixed number of operations.

It exits at lines 6, 9 in $\frac{1}{q+(q-1)!}$ of cases, in which case it operates in at most (depending on the data structures in use) $O(n)+M_p+L_p$ steps in the average case.

Finally, $\texttt{Successor}_\infty$ returns from lines 7, 10 in $\frac{(q-1)!-1}{q+(q-1)!}$ of cases, after performing $O(n)+M_p+L_{q-1}$ steps.

In every sensible implementation of $\texttt{Succ}$ (i.e., where we assume $\frac{L_p-L_{q-1}}{q+(q-1)!}\to 0$) we then have an amortized runtime of $O\left(n+L_{q-1}+M_p\right)$. □

We now note that by [8, Th. 7,10] we may assume $\texttt{Succ}$ to operate in $O(1)$ steps in the average case, and by [8, Part III-C] (which also relies on [37]) we assume $\texttt{Rn}$ runs in $O(n)$ steps, yielding an average runtime of $O(n)$ for $\texttt{Successor}_\infty$.

We shall also present the function $\texttt{Rank}_\infty(n,[a_1,\ldots,a_n])$ that, given a permutation in the ℓ_∞-snake presented in Section 4.1, returns that permutation's rank in the code. This function uses the function $\texttt{Rn}$ discussed above as well, and works by considering the same cases discussed above.

Lemma 11. *If the function* $\texttt{Rn}$ *operates in* M_n *steps, then* $\texttt{Rank}_\infty$ *has a runtime of* $O(n+M_p)$ *(in the average or worst case respectively).*

Function $\texttt{Rank}_\infty\left([a_1, \ldots, a_n]\right)$

input : A permutation $[a_1, a_2, \ldots, a_n]$ in the ℓ_∞-snake from Theorem 5
output : $k \in \mathbb{N}$ that represents the given permutation's rank in the code

1 $q \leftarrow \lfloor \frac{n}{2} \rfloor$; $p \leftarrow \lceil \frac{n}{2} \rceil$
2 **if** $a_{q+1} \equiv 0 \pmod 2$ **then**
3 $i \leftarrow \min\{j \in [n] \mid a_j \not\equiv 0 \pmod 2\}$
4 **return** $i - 1 + (q + (q-1)!) \cdot \texttt{Rn}\left(\left[\frac{a_i+1}{2}, \frac{a_{q+2}+1}{2}, \ldots, \frac{a_n+1}{2}\right]\right)$
5 $R \leftarrow \texttt{Rn}\left(\left[\frac{a_{q+1}+1}{2}, \ldots, \frac{a_n+1}{2}\right]\right)$
6 **if** $R \equiv 0 \pmod 2$ **then**
7 **return** $q + (q + (q-1)!) \cdot R + \texttt{Rn}\left(\left[\frac{a_1}{2}, \ldots, \frac{a_{q-1}}{2}\right]\right)$
8 **return** $q + (q + (q-1)!) \cdot R + \texttt{Rn}\left(\texttt{sw}\left(\left[\frac{a_1}{2}, \ldots, \frac{a_{q-1}}{2}\right]\right)\right)$

Function $\texttt{Unrank}_\infty\left(n, k\right)$

input : $4 \leq n \in \mathbb{N}$; rank $k \in \mathbb{N}$
output : The permutation $[a_1, a_2, \ldots, a_n]$ which is kth in the (n, M, ℓ_∞)-snake from Theorem 5

1 $q \leftarrow \lfloor \frac{n}{2} \rfloor$; $p \leftarrow \lceil \frac{n}{2} \rceil$
2 $R \leftarrow \left\lfloor \frac{k}{q+(q-1)!} \right\rfloor$; $r \leftarrow (k \bmod (q + (q-1)!))$
3 $[b_1, \ldots, b_p] \leftarrow \texttt{UnR}(p, R)$
4 **if** $r \geq q$ **then**
5 $[a_1, \ldots, a_{q-1}] \leftarrow \texttt{UnR}(q-1, r-q)$
6 **if** $R \equiv 1 \pmod 2$ **then**
7 $[a_1, \ldots, a_{q-1}] \leftarrow \texttt{sw}\left([a_1, \ldots, a_{q-1}]\right)$
8 **return** $[2a_1, \ldots, 2a_{q-1}, 2q, 2b_1 - 1, \ldots, 2b_p - 1]$
9 **if** $R \equiv 0 \pmod 2$ **then**
10 **return** $[2, 4, 6, \ldots, 2r, 2b_1 - 1, 2(r+1), \ldots, 2q, 2b_2 - 1, \ldots, 2b_p - 1]$
11 **return** $[4, 2, 6, \ldots, 2r, 2b_1 - 1, 2(r+1), \ldots, 2q, 2b_2 - 1, \ldots, 2b_p - 1]$

Proof. We partition our proof by return condition once more. If the program exits from 4 then it performed $O(q) + M_p$ steps.

If it exits from 7 or 8 then it performed $O(1) + M_p + M_{q-1}$ steps. □

Again, by results discussed above, we note that $\texttt{Rank}_\infty$ runs in $O(n)$ steps in the average case.

As mentioned before, unranking permutations in the code might also be required. For that purpose we implement the function $\texttt{Unrank}_\infty(n, k)$, accepting as input the length of the code and a specific rank and returning the implied permutation. We will assume the existence of a similar function $\texttt{UnR}$ for the construction used in Section 4.1, where again we assume the unit permutation to have rank zero.

Once more, our implementation and estimate of $\texttt{Unrank}_\infty$'s runtime relies heavily on

that of its auxiliary functions.

Lemma 12. *If the function* `UnR` *operates in* N_n *steps, then* $\texttt{Unrank}_\infty$ *runs in* $O(n + N_p)$ *steps.*

Proof. One notes that the only operations in $\texttt{Unrank}_\infty$ that take more than a fixed number of steps are calls for `sw` (taking $O(n)$), calls for `UnR`, and, depending on the data structures in use, concatenation of indices (at most $O(n)$ as well). The claim follows. □

Again, it shall be noted that, relying on Lemma 8 and [8, Part III-C], $\texttt{Unrank}_\infty$ can be performed in $O(n^2)$ operations.

Chapter 5

Conclusion

In this work we explored rank-modulation snake-in-the-box codes under both Kendall's τ-metric and the ℓ_∞-metric. In both settings we presented a construction yielding codes with rates asymptotically tending to 1, and presented auxiliary functions for the production of the successor permutation, as well as ranking and unranking for permutations in the codes, to facilitate implementation of the given codes for logical cells in flash memory. We also proved upper-bounds on the size of $\mathcal{K}$-snakes, and presented known bounds for ℓ_∞-snakes.

However, we note that despite their optimal asymptotic rates, the presented codes fail to achieve the given bounds by a constant fraction. As it is not presently known whether the upper-bounds presented and referenced in this work are achievable, we were unable to show how close the codes generated by our constructions come to being optimal with respect to their sizes. A computer search for *cyclic* codes, performed on S_5, yielded $(5, M, \mathcal{K})$-snakes of maximal size $M = 57$ (for comparison, the construction from Theorem 1 yields a $(5, 45, \mathcal{K})$-snake). While an abundance of such codes were found (well over 500 nonequivalent codes), they all were in fact codes over A_5. For completeness, we present one of those codes in Figure 5.1.

It shall be noted that a complete (but not cyclic) $(5, 60, \mathcal{K})$-snake over A_5 can easily be constructed from each cyclic code we tested by generating the skipped coset of S_3 with two t_3 operations, followed by a t_5 operation and the given code, in order. However, we do not currently know whether $(2n + 1, \frac{(2n+1)!}{2}, \mathcal{K})$-snakes over A_{2n+1} exist for every length.

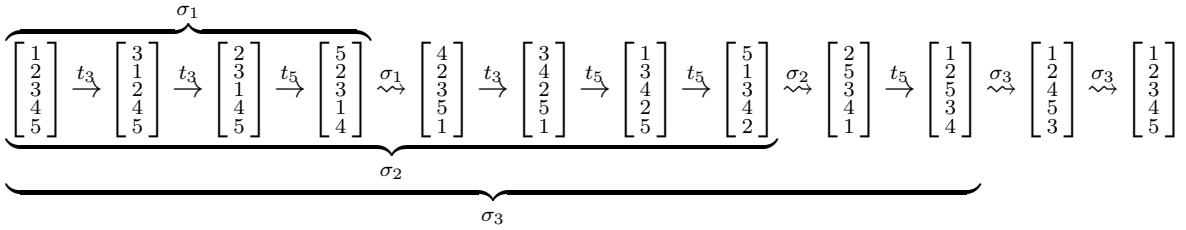

Figure 5.1: A $(5, 57, \mathcal{K})$-snake generated by a computer search. Squiggly arrows stand for a repetition of the transitions defined by the braces.

In a recent paper based on the work presented here, Horovitz and Etzion [38] were able to improve upon these results, and recursively construct $(2n+1, \tilde{M}_{2n+1}, \mathcal{K})$-snakes with $\lim_{n\to\infty} \frac{\tilde{M}_{2n+1}}{|S_{2n+1}|} \simeq 0.4338$. They did so by partitioning A_{2n+1} into $2n(2n+1)$ 'copies' of A_{2n-1}, each containing an induced copy of a $\mathcal{K}$-snake C_{2n-1}. Constructing a $(2n+1, \tilde{M}_{2n+1}, \mathcal{K})$-snake by concatenation of some of these copies, they were able to cast the problem into the framework of 3-*uniform hypergraphs* (studied in [39]), where nodes represent the induced $\mathcal{K}$-snakes. More precisely, they showed that a tree in such graphs (a subgraph containing no cycles) induced a (non-unique) concatenation of the copies, and were able to prove the existence of maximal sized trees, containing all but one of these $2n(2n+1)$ nodes, thereby constructing C_{2n+1} with $\tilde{M}_{2n+1} = [2n(2n+1) - 1]\, \tilde{M}_{2n-1}$. For comparison, recall that in the construction presented in our work, one has $M_{2n+1} = [(2n-1)(2n+1)]\, M_{2n-1}$. Note in particular that $\tilde{M}_5 = 57$, the size of a maximal $\mathcal{K}$-snake in A_5.

While computer searches in the symmetric group of a higher order appear to be infeasible, we include one more peculiar result: each maximal code we tested in A_5 skipped 3 permutations who all agree on $4, 5$, i.e., it skipped a coset of S_3 (of the same kind forfeited by the construction of [38]). We earlier conjectured that this phenomenon may persist, since the codes generated by Theorem 1 of lengths 7 and 9 display it as well (several cosets of S_5 and S_7 were absent, respectively), but Horovitz and Etzion have constructed $(7, 2515, \mathcal{K})$- and $(9, 181433, \mathcal{K})$-snakes, which disproves the notion. They have conjectured that the maximal achievable size of a $(2n+1, M, \mathcal{K})$-snake is $M = \frac{(2n+1)!}{2} - (2n-1)$. The authors are unaware of any results either proving this as an upper bound for $\mathcal{K}$-snakes or demonstrating the existence of such codes for $n > 4$, as of yet.

These results, along with the bounds we showed in Theorem 4 and Theorem 3 give rise to the following conjecture: For all $n \in \mathbb{N}$ a cyclic $\mathcal{K}$-snake exists over A_n whose size is no less than that of every cyclic $\mathcal{K}$-snake over S_n.

In addition, searches done in a computer for ℓ_∞-snakes for lengths $4, 5, 6$ returned codes of size $6, 30, 90$ respectively, suggesting that perhaps the upper-bound of [23, Th. 20] is achievable. Moreover, in these cases we were able to find codes generated only by "push-to-the-top" operations on the last two indices. A code for each length is presented in Figure 5.2 in binary representation (conveniently written in octal notation), where zeroes stand for t_n's and ones for t_{n-1}'s. Searches for higher lengths again seem infeasible.

n	Defining Transitions
4	55
5	0212206063
6	010204410222042124446130162347

Figure 5.2: $(4, 6, \ell_\infty)$-, $(5, 30, \ell_\infty)$- and $(6, 90, \ell_\infty)$-snakes generated by a computer search. All codes represented by a sequence of "push-to-the-top" operations, applied in order to the identity permutation, where zeroes stand for t_n's and ones for t_{n-1}'s. The binary strings are given in octal notation and should be read from left to right.

Bibliography

[1] J. Brewer and M. Gill, *Nonvolatile Memory Technologies with Emphasis on Flash.* Wiley-IEEE Press, 2008.

[2] A. Jiang, V. Bohossian, and J. Bruck, "Rewriting codes for joint information storage in flash memories," *IEEE Trans. on Inform. Theory*, vol. 56, no. 10, pp. 5300–5313, Oct. 2010.

[3] R. L. Rivest and A. Shamir, "How to reuse a "write-once" memory," *Inform. and Control*, vol. 55, pp. 1–19, 1982.

[4] A. Jiang, V. Bohossian, and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," in *Proceedings of the 2007 IEEE International Symposium on Information Theory (ISIT2007), Nice, France*, Jun. 2007, pp. 1166–1170.

[5] E. Yaakobi, A. Vardy, P. H. Siegel, and J. K. Wolf, "Multidimensional flash codes," in *Proc. of the Annual Allerton Conference*, 2008.

[6] A. Jiang, M. Langberg, M. Schwartz, and J. Bruck, "Trajectory codes for flash memory." *IEEE Trans. on Inform. Theory*, vol. 59, no. 7, pp. 4530–4541, 2013.

[7] F. Chierichetti, H. Finucane, Z. Liu, and M. Mitzenmacher, "Designing floating codes for expected performance," *IEEE Trans. on Inform. Theory*, vol. 56, no. 3, pp. 968–978, Mar. 2010.

[8] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank modulation for flash memories," *IEEE Trans. on Inform. Theory*, vol. 55, no. 6, pp. 2659–2673, Jun. 2009.

[9] N. Papandreou, H. Pozidis, T. Mittelholzer, G. F. Close, M. Breitwisch, C. Lam, and E. Eleftheriou, "Drift-tolerant multilevel phase-change memory," in *Proceedings of the 3rd IEEE International Memory Workshop (IMW), Monterey, CA, U.S.A.*, May 2011, pp. 22–25.

[10] F. Gray, "Pulse code communication," March 1953, U.S. Patent 2632058.

[11] C. D. Savage, "A survey of combinatorial Gray codes," *SIAM Rev.*, vol. 39, no. 4, pp. 605–629, Dec. 1997.

[12] A. Nijenhuis and H. S. Wilf, *Combinatorial algorithms for computers and calculators*, ser. Computer science and applied mathematics. New York: Academic Press, 1978.

[13] J. Robinson and M. Cohn, "Counting seqeuences," *IEEE Trans. on Comput.*, vol. C-30, pp. 17–23, May 1981.

[14] D. J. Amalraj, N. Sundararajan, and G. Dhar, "Data structure based on gray code encoding for graphics and image processing," vol. 1349, 1990, pp. 65–76.

[15] C. Faloutsos, "Gray codes for partial match and range queries," *IEEE Trans. on Software Eng.*, vol. 14, pp. 1381–1393, 1988.

[16] T. Etzion, "Optimal codes for correcting single errors and detecting adjacent errors," *IEEE Trans. on Inform. Theory*, vol. 38, no. 4, pp. 1357–1360, Jul 1992.

[17] M. Schwartz and T. Etzion, "The structure of single-track gray codes," *IEEE Trans. on Inform. Theory*, vol. 45, no. 7, pp. 2383–2396, Nov 1999.

[18] C. C. Chang, H. Y. Chen, and C. Y. Chen, "Symbolic Gray code as a data allocation scheme for two-disc systems," *Comput. J.*, vol. 35, pp. 299–305, 1992.

[19] R. Sedgewick, "Permutation generation methods," *ACM Computing Surveys*, vol. 9, no. 2, pp. 137–164, Jun. 1977.

[20] A. Jiang, M. Schwartz, and J. Bruck, "Correcting charge-constrained errors in the rank-modulation scheme," *IEEE Trans. on Inform. Theory*, vol. 56, no. 5, pp. 2112–2120, May 2010.

[21] A. Barg and A. Mazumdar, "Codes in permutations and error correction for rank modulation," *IEEE Trans. on Inform. Theory*, vol. 56, no. 7, pp. 3158–3165, Jul. 2010.

[22] A. Mazumdar, A. Barg, and G. Zémor, "Constructions of rank modulation codes," *IEEE Trans. on Inform. Theory*, vol. 59, no. 2, pp. 1018–1029, 2013.

[23] I. Tamo and M. Schwartz, "Correcting limited-magnitude errors in the rank-modulation scheme," *IEEE Trans. on Inform. Theory*, vol. 56, no. 6, pp. 2551–2560, Jun. 2010.

[24] T. Kløve, T.-T. Lin, S.-C. Tsai, and W.-G. Tzeng, "Permutation arrays under the Chebyshev distance," *IEEE Trans. on Inform. Theory*, vol. 56, no. 6, pp. 2611–2617, Jun. 2010.

[25] W. H. Kautz, "Unit-distance error-checking codes," *IRE Trans. on Electronic Computers*, vol. EC-7, no. 2, pp. 179–180, Jun. 1958.

[26] F. Harary, J. P. Hayes, and H.-J. Wu, "A survey of the theory of hypercube graphs," *Computers and Mathematics with Applications*, vol. 15, no. 4, pp. 277–289, 1988.

[27] H. L. Abbot and M. Katchalski, "On the construction of snake in the box codes," *Utilitas Math.*, vol. 40, pp. 97–116, 1991.

[28] S. Hood, D. Recoskie, J. Sawada, and D. Wong, "Snakes, coils, and single-track circuit codes with spread k," *Journal of Combinatorial Optimization*, vol. 30, no. 1, pp. 42–62, July 2015.

[29] Z. Wang and J. Bruck, "Partial rank modulation for flash memories," in *Proceedings of the 2010 IEEE International Symposium on Information Theory (ISIT2010), Austin, TX, U.S.A.*, Jun. 2010, pp. 864–868.

[30] E. En Gad, M. Langberg, M. Schwartz, and J. Bruck, "Constant-weight Gray codes for local rank modulation," *IEEE Trans. on Inform. Theory*, vol. 57, no. 11, pp. 7431–7442, Nov. 2011.

[31] E. En Gad, A. Jiang, and J. Bruck, "Compressed encoding for rank modulation," in *Proceedings of the 2011 IEEE International Symposium on Information Theory (ISIT2011), St. Petersburg, Russia*, Aug. 2011, pp. 884–888.

[32] E. En Gad, M. Langberg, M. Schwartz, and J. Bruck, "Generalized Gray codes for local rank modulation," *IEEE Trans. on Inform. Theory*, vol. 59, no. 10, pp. 6664–6673, Oct. 2013.

[33] M. Kendall and J. D. Gibbons, *Rank Correlation Methods.* Oxford University Press, NY, 1990.

[34] A. Mazumdar, A. Barg, and G. Zémor, "Constructions of rank modulation codes," in *Proceedings of the 2011 IEEE International Symposium on Information Theory (ISIT2011), St. Petersburg, Russia*, Aug. 2011, pp. 834–838.

[35] M. Deza and H. Huang, "Metrics on permutations, a survey," *J. Comb. Inf. Sys. Sci.*, vol. 23, pp. 173–185, 1998.

[36] M. Schwartz and I. Tamo, "Optimal permutation anticodes with the infinity norm via permanents of $(0,1)$-matrices," *J. Combin. Theory Ser. A*, vol. 118, pp. 1761–1774, 2011.

[37] M. Mares and M. Straka, "Linear-time ranking of permutations," *Algorithms-ESA*, pp. 187–193, 2007.

[38] M. Horovitz and T. Etzion, "Constructions of snake-in-the-box codes for rank modulation," *IEEE Trans. on Inform. Theory*, vol. 60, no. 11, pp. 7016–7025, Nov 2014.

[39] A. Goodall and A. de Mier, "Spanning trees of 3-uniform hypergraphs," *Advances in Applied Mathematics*, vol. 47, pp. 840–868, 2011.

תקציר

לאור סכמת הקידוד 'rank-modulation' עם שימושים לזכרונות flash, אנו בוחנים קודי גריי המסוגלים לזהות שגיאה בודדת, המוכרים כקודי snake-in-the-box. אנחנו בוחנים שתי מטריקות שגיאה: מטריקת τ של קנדל, הממדלת רעש בעל מטען חסום, ומטריקת ℓ_∞ הממדלת רעש בעל עוצמה חסומה. בשני המקרים אנו בונים קודי snake-in-the-box עם קצב המתכנס אסימפטוטית ל- 1. אנחנו גם מציעים אלגוריתמים יעילים לקידום רמה בקוד, בנוסף לקידוד וגילוי בסכימה זו. לבסוף, אנו בוחנים חסמים כלליים לגודל קודים שכאלו.

אוניברסיטת בן-גוריון בנגב
הפקולטה למדעי ההנדסה
המחלקה להנדסת חשמל ומחשבים

קודי גריי מגלי-שגיאות עבור Rank-Modulation

חיבור זה מהווה חלק מהדרישות לקבלת תואר מגיסטר בהנדסה

מאת: יונתן יחזקאלי

מנחה: פרופ' משה שוורץ

חתימת המחבר: _____ תאריך: 15/06/2016

אישור המנחה: _____ תאריך: 15/06/2016

_____ תאריך: _____

אישור יו"ר ועדת תואר שני מחלקתית: _____ תאריך: 15/06/2016

סיוון תשע"ד

יוני 2016

אוניברסיטת בן-גוריון בנגב
הפקולטה למדעי ההנדסה

המחלקה להנדסת חשמל ומחשבים

קודי גריי מגלי-שגיאות עבור Rank-Modulation

חיבור זה מהווה חלק מהדרישות לקבלת תואר מגיסטר בהנדסה

מאת: יונתן יחזקאלי

סיוון תשע"ד

יוני 2016