RELIABLE MEMORY STORAGE BY NATURAL REDUNDANCY

An Undergraduate Research Scholars Thesis

by

JACOB MINK

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Anxiao Jiang

May 2018

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

Reliable Memory Storage by Natural Redundancy

Jacob Mink
Department of Computer Science
Texas A&M University

Research Advisor: Dr. Anxiao Jiang
Department of Computer Science and Engineering
Texas A&M University

Non-volatile memories are becoming the dominant type of storage devices in modern computers because of their fast speed, physical robustness and high data density. However, there still exist many challenges, such as the data reliability issues due to noise. An important example is the memristor, which uses programmable resistance to store data. Memristor memories use the crossbar architecture and suffer from the sneak-path problem: when a memristor cell of high resistance is read, it can be mistakenly read as a low-resistance cell due to low-resistance sneak-paths in the crossbar that are parallel to the cell.

In this work, we study new ways to correct errors using the inherent redundancy in stored data (called Natural Redundancy), and combine them with conventional error-correcting codes. In particular, we define a Huffman encoding for the English language based on a repository of books. In addition, we study data stored using convolutional codes and use natural redundancy to verify if decoded codewords are valid or invalid.

We present statistics over the Viterbi Algorithm and its ability to decode convolutional

codewords, then discuss Yen's Algorithm, an augmentation of the Viterbi Algorithm. Finally, we present an efficient algorithm to search for a list of the most likely codewords, and choose a codeword that meets the criteria of both natural redundancy and the ECC as the decoding solution. We find that this algorithm is no more powerful than Yen's Algorithm in terms of decoding noisy convolutional codewords, but does present some interesting ideas for further exploration across multiple fields of study.

DEDICATION

To my family, who support my studies in a field about which they hear more than they ever wanted.

ACKNOWLEDGMENTS

In appreciation to Dr. Anxiao Jiang, my research advisor, for helping me to get started in research and for his constant willingness to put me back on the right track when it comes to understanding the requirements.

For the help of Jerry Cheng, without whom I may not have been able to complete the experiments surrounding Yen's Algorithm. His experience and clarity in the matter were extremely helpful in completing this thesis.

In appreciation of my father's ever-ready ear; he was always willing to listen to me babble about code and help me debug on a system he couldn't even see, regardless of his understanding of the topic. His sacrifice of time helped mine go by much quicker.

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The need to store information has been of great importance to mankind since the dawn of civilization. With the development of spatially constant memory such as cave drawings, to the advent of paper information and books, to the creation of the first computers, we see the necessity and effect that memory has on our world. As our computers grow in power and our computational needs grow in complexity, our memory storage must become faster and more robust.

Current memory technology relies on flash memory to function; it is faster than hard drives and is non-volatile, but it does not have the speed and endurance needed by the fast-evolving big data era [1]. Futuristic memory technologies may rely on chemical or electrical properties of materials that change based on certain applications. For example, phase-change memory, described in the Proceedings of the IEEE shows us that the chemical properties of a material can be exploited to represent binary digits. [2, 3] IBM has also published on their racetrack memory, which can be conceptualized as tiny electrical elevators to move information from place to place. [4]

Another important technology, and the focus of this thesis, is the memristor. Similar in function to a basic transistor, a memristor holds a value by way of resistance. Mohanty describes the necessary elements to a memristor in IEEE Potentials, namely that the memristor changes its resistance depending on the time of voltage application and the strength of voltage application. [5, 6] By specific voltage applications, a memristor can be made to hold a binary 1 or 0, and an array of memristors can be used to hold large scale memory. The main advantage of memristors is that power is only required to read or set a bit in memory, but not to maintain the memory over time. The memrisot has the potential to be faster and denser than flash memory. However, there is a physical problem in its structure, known as the Sneak-Path Problem.

In the following, we first give an overview of the memristor technology and error-correcting codes (ECCs) that are commonly used in storage systems, including nonvolatile memories. We then introduce the challenges that still face memristor crossbar memories, and in particular, the sneak-path problem. We then introduce an idea for improving the error correction performance of error correction codes, which is to combine the residual redundancy (called Natural Redundancy) in data (which can be either uncompressed or compressed imperfectly) with the redundancy added by ECCs (called Artificial Redundancy) for better decoding. We then summarize the main topics we explore in this research work.

1.1 Overview of Memristor Technology

The memristor was first introduced in 1971 by Leon O. Chua as one of the basic elements of circuits, including resistors, inductors, and capacitors. The behavior of a memristor is similar to a resistor with the ability to hold a state. The technology has been realized in the lab using M-R, M-L, and M-C mutators. [7] The physical properties of memristors present an opportunity to improve the performance of current data storage technology.

Possibly the most important property of memristors to data storage is that the resistance of a memristor changes based on voltage applied and length of time of application. Because of this, the resistance can be read easily by a very small voltage application and then used as data storage in and of itself. Therefore, we can create a memristor crossbar array, a simple connection of grid-arranged wires using memristors. In the design of a memristor crossbar array, since memristors hold non-discrete values in their resistance, a threshold must be specified to distinguish between a 0 and a 1. By convention, a 0 is a high-resistance memristor while a 1 is a low-resistance memristor.

1.2 Error Correcting Codes (ECCs) for Storage Systems

An inherent problem in storage systems is the detection of and recovery from read errors in data. A single cell or more in a memory array may become corrupt, which can result in fatal errors as scale increases. Error correcting codes are a method by which errors can be mitigated. While there are many types of ECCs, we focus on convolutional codes as our primary method of error detection and correction.

A convolutional code can be expressed as a redundancy based on received bits. In general, a convolutional code produces $r > 1$ bits based on a subset of the message to send; the rate of a convolution code is the number $1/r$. If x is the message to send, the r redundant bits $p_i[n]$ are generated by functions of the form

$$p_i[n] = (\sum_{j=0}^{k-1} g_i[j]x[n-j]) \bmod 2 \tag{1.1}$$

where $1 \leq i < r$ and n ranges across the length of the message to send. The term g_i is a generator polynomial for the bit p_i, having k coefficients specifying what subset of the message to consider. Note that we assume there is padding by $k-1$ 0s in the beginning of x to ensure validity of every bit p_i. [8]

An alternate expression for a convolutional code is by a finite state automaton. In essence, it is a discrete Markov model expanded over time. [9] As an example, see Figure 1.1, the convolutional code we used.

1.3 Challenges for Memristor Crossbar Memories

Although memristors do not fail when there is a lack of power, they do have an inherent physical problem that makes their usage difficult. The sneak-path problem is a hardware based issue in memristor crossbar memories used for memory storage. At the most basic, when attempting to read a given cell (i, j) that happens to contain a 0, if there are is set of

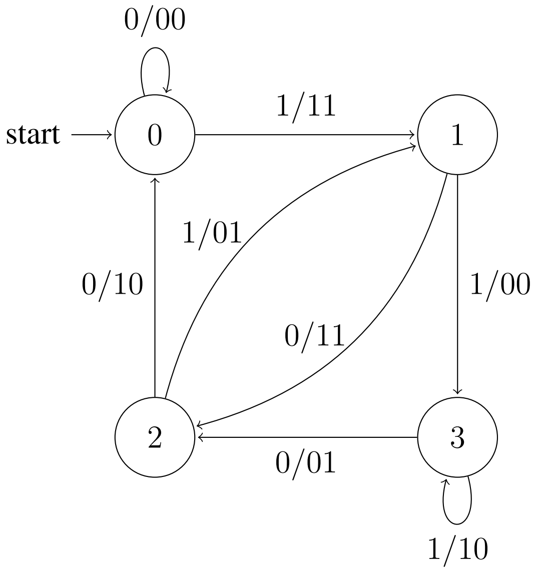

Figure 1.1: A finite state automaton representing a $1/2$ convolutional code

1s in parallel with cell (i, j), then the cell could be mistaken for a logical 1. The reason for this error is that since the 1s are at a lower resistance than the 0s, a path of least resistance can be created with less resistance than the 0 cell. [10, 11]

Mathematically, the sneak-path problem can be expressed as such: Given a binary array A of size $m \times n$, we say that there is a sneak path of length $2k + 1$ affecting the cell at position (i, j) if $a_{i,j} = 0$ and there exist $2k$ positive integers $1 \leq r_1, ..., r_k \leq m$ and $1 \leq c_1, ..., c_k \leq n$ for some $k \geq 1$ such that the following $2k + 1$ cells satisfy [10]

$$a_{i,c_1} = a_{r_1,c_1} = a_{r_1,c_2} = ... = a_{r_{k-1},c_k} = a_{r_k,c_k} = a_{r_k,j} = 1 \tag{1.2}$$

Several hardware solutions have been proposed for the sneak-path problem in an attempt to discuss the problem at its direct source. [11, 12, 13, 14]

1.4 Using Natural Redundancy for Error Correction

Natural redundancy is very helpful in the field of error correction. A redundant message is much more likely to be decoded than a message that is not redundant, so we can create redundant representations for characters in such a way that they can be decoded [15, 16, 17, 18]. To do this to an entire message is to encode it. In the case of a convolutional code, it is encoded mathematically and the result is longer by a factor r than the original string.

Once the encoded message has been created from the original message, errors can be introduced to it. Whether these errors come from the encoded message being sent across a channel or are due to some other random element, the codeword must be decoded to return a string with as little difference from the original message as possible. The Viterbi Algorithm is one way in which the encoded message can be decoded. It is a variation of Dijkstra's Algorithm on a stage-graph structure. The stage-graph is developed from the finite state automaton for the convolutional code as shown in Figure 1.2. In fact, Figure 1.2 can be extended to include more stages and handle longer messages. [19]

First, we remember that the Hamming distance between two binary words is, most simply, the number of bits different between the two. [20] The Viterbi Algorithm works by assigning an edge weight, $w(e)$ to every edge by the Hamming distance between the received encoded bits and what the edge transition is expected to produce. Next, each state i, j is assigned an initial node weight of a very large value, $w(i, j)$. The node at stage 0, state 0 is given a node weight of 0, since every message is encoded starting in that state. For each state j in every stage i, we compute the number $w(e) + w(i, j)$ for the edges leave state i, j. Each node j in stage $i + 1$ then stores the edge yielding the minimum number as its parent edge and let its node weight $w(i + 1, j) = w(e) + w(i, j)$. The algorithm completes once every node has been visited, except for those in the last stage (they need

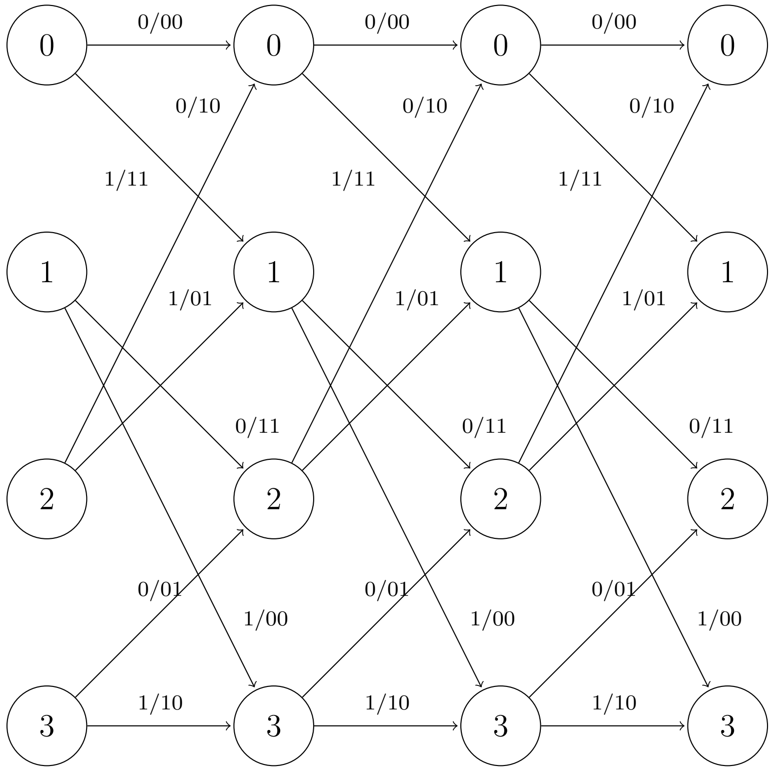

Figure 1.2: A stage graph developed from the finite state automaton in Figure 1.1

not be computed since they do not have any further connections). Note that often, extra 0s are appended to the end of the message before encoding to ensure that the shortest path in this graph starts at $0, 0$ and ends at $n - 1, 0$, where n is the number of stages. The edges are then traced back to yield the codeword of minimum Hamming distance based on the sent message. [19, 21]

1.5 Summary of Research Topics

Many areas were covered over the course of this research. In particular, memristor physical issues, the analysis of texts for character frequencies and the generation of Huffman Codes for the characters, error correcting codes and their decoding algorithms, and shortest path and k-shortest path algorithms were studied to prepare and develop the research.

The physical limitations of memristors, in particular the sneak-path problem, served as motivation for the research. To represent the sneak-path problem, we used Huffman Codes for a very large selection of texts to simulate bits stored in memory. The research into error correcting codes went towards choosing an optimal method by which to encode and decode. Finally, k-shortest path algorithms are the method by which we improve the performance of our decoding method.

2. ANALYSIS OF THE SNEAK PATH PROBLEM IN MEMRISTORS

It is important to know exactly how problematic the sneak-path problem is for memristor crossbar arrays to evaluate the usefulness of this research. We simulated the behavior of memristor storage and the sneak-path problem by the following method:

1. Create a matrix of size 4×4 and populate it with uniformly random 0s and 1s.

2. Set the values at $(0, 0)$, $(N/4 - 1, N/4 - 1)$, and $(N/2 - 1, N/2 - 1)$ to 0 where N is the matrix size.

3. For a matrix of size $N \times N$, create

   (a) N^2 nodes $R(i, j)$ for $1 \leq i \leq N$ and $1 \leq j \leq N$ that represent the crossing points at the N row nodes, adjacent to its left and right neighbors (with no wrapping).

   (b) N^2 nodes $C(i, j)$ for $1 \leq i \leq N$ and $1 \leq j \leq N$ that represent the crossing points at the N column nodes, adjacent above and below (with no wrapping).

   (c) edges between the node $R(i, j)$ and $C(i, j)$ if the value at cell $a_{i,j}$ in the matrix is 1.

4. Breadth-First Search starting at $R(i, j)$ for $(i, j) = \{(0, 0), (N/4-1, N/4-1), (N/2-1, N/2-1)\}$ and attempt to find the node $C(i, j)$. If this node is found, return True. Otherwise, return False.

5. Repeat steps 1 through 4 several times and collect data.

6. Repeat steps 1 through 5 using sizes 8×8, 16×16, 32×32, 64×64, and 128×128.

The data in Figure 2.1 was collected from simulations using 6 sizes of memory: 4×4, 8×8, 16×16, 32×32, 64×64, and 128×128. Each size was polled 1000 times at 3 positions, (0, 0), ($\frac{N}{4} - 1$, $\frac{N}{4} - 1$), and ($\frac{N}{2} - 1$, $\frac{N}{2} - 1$), where the matrices are size $N \times N$, on different, uniformly random matrices (except for the cells in question, which were held at 0). The data below reflects three separate curves over the size of the memory corresponding to the three different positions.

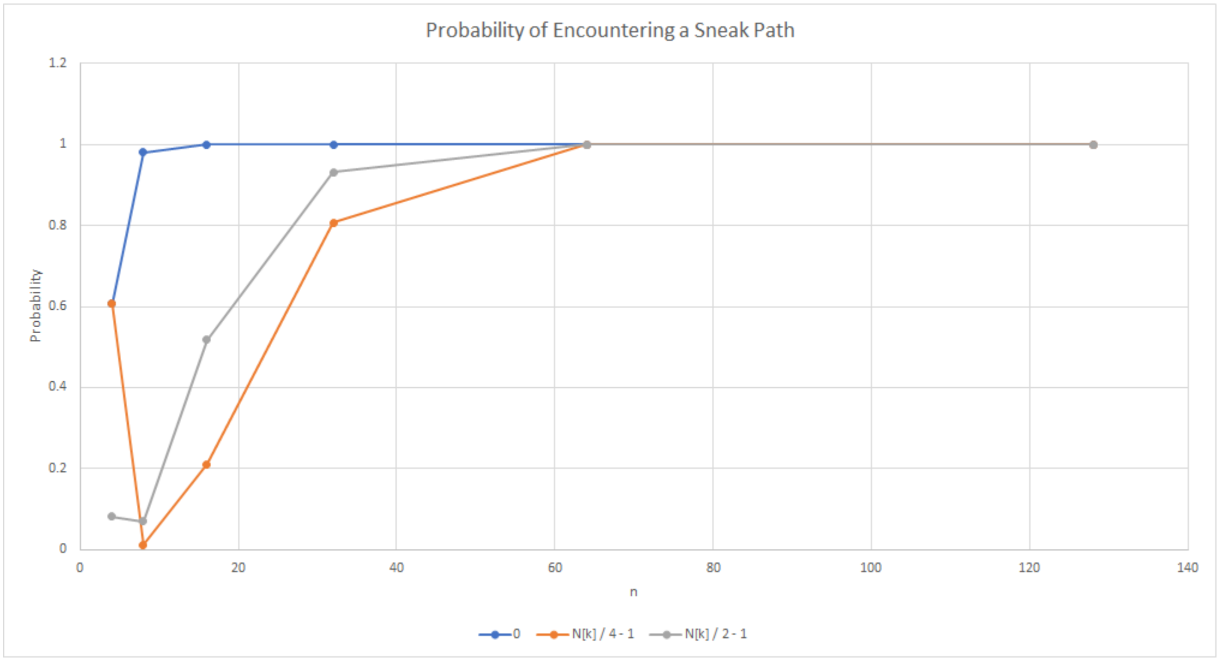

Figure 2.1: The probability of a sneak-path error occurring in various sizes of memristor arrays

The data in Figure 2.1 tells us that as the size of the memristor array increases, the probability of finding a sneak-path error increases to 100% relatively quickly. Physically, however, the path might have too large a resistance to actually be a sneak-path. The data in Figure 2.2 shows the probability of finding a sneak-path error at the locations (0, 0),

$(\frac{N}{4}-1, \frac{N}{4}-1)$, and $(\frac{N}{2}-1, \frac{N}{2}-1)$ on the memristor array, where $N \times N$ is the size of the array. These sizes were, again, 4×4, 8×8, 16×16, 32×32, 64×64, and 128×128.

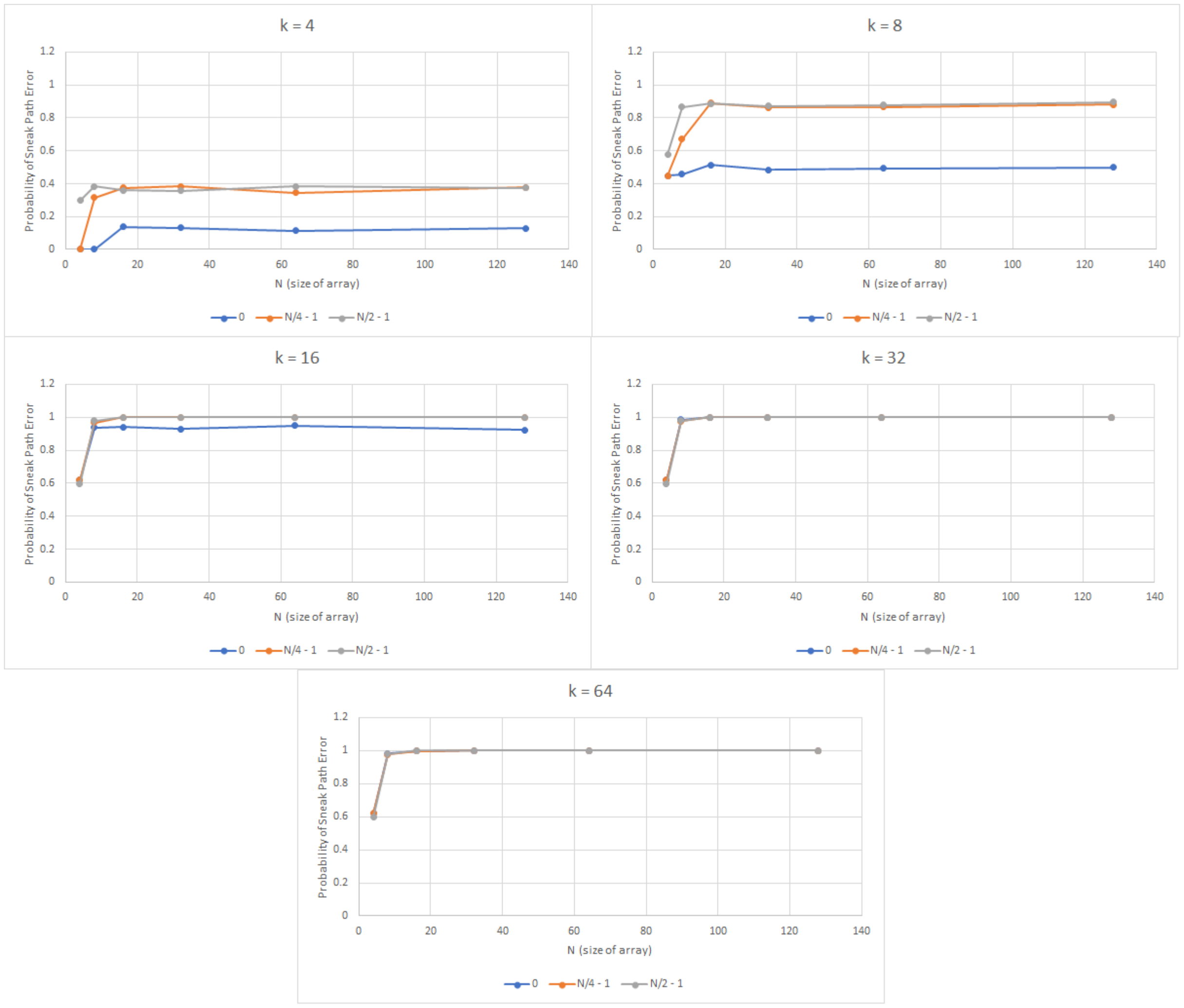

Figure 2.2: The probability of a sneak-path error occuring with a maximum path length, k, in various sizes of memristor arrays

We can see by the data in Figure 2.2 that for wires with high resistance per wire segment (low k), the probability of finding a so-called k-sneak-path levels out at around 40%, but as the resistance per wire segment decreases, we see similar results to the original data.

Obviously, the Sneak-Path Problem is very common in a memristor array, and can occur in various locations around the array. In this thesis, we introduce a method of encoding and decoding words in memory in polynomial time by which any memory-read errors can be corrected to produce valid words.

3. DESIGN OF HUFFMAN CODES FOR THE ENGLISH LANGUAGE

Huffman coding is an optimal compression method, and is useful for converting texts into binary strings. Using a specific optimal Huffman code for a set of texts allows us to represent and analyze data in a more simple way than language processing would allow. One of the most important aspects of the Huffman code is that all the codewords are prefix-free. To explain, a set containing two words 000 and 0001 would not be prefix-free since $"000" \subset "0001"$. This property is helpful in decoding, since it allows for $O(n)$ processing the bit-string.

3.1 Research Huffman Code

In our research, we used the Gutenberg corpus from Python's NLTK library to source our texts. [22] Our code was designed in the following way:

1. Across all texts in the Gutenberg corpus, collect the frequencies of every character.
2. Additionally, store the valid words from every text for later use.
3. From the list of frequencies, construct a binary tree with the property that letters only appear at leaf nodes and the maximum frequency letters are closer to the root.
4. Run DFS on the tree to compute the strings that represent each letter (left child is a '0', right child is a '1')

The Huffman code in Table 3.1 was designed by this previous procedure.

3.2 Basic Decoding from Huffman Compression

The main problem after encoding a text by the Huffman code is how to decode the results. Due to the prefix-free nature of the Huffman code, we can simply read bits until

Table 3.1: An optimal Huffman code for the English language based on the texts provided by the Gutenberg corpus

character	bit-string	character	bit-string	character	bit-string
s	0000	\n	00010	k	0001100
O	000110100	F	0001101010	Y	00011010110
z	00011010111	S	000110110	J	0001101110
!	0001101111	x	0001110000	4	0001110001
2	000111001	C	0001110100	G	0001110101
?	0001110110	_	0001110111000	V	0001110111001
*	000111011101000	/	000111011101001000	@	000111011101001001000
>	000111011101001001001	æ	00011101110100100100	=	0001110111010010010101010
î	0001110111010010010101011	è	00011101110100100100110	<	00011101110100100100111
→	00011101110100100110000	+	000111011101001001100110	∼	000111011101001001100111
é	00011101110100100110010	%	00011101110100100110011	$	000111011101001001110
&	00011101110100101	X	0001110111010011	Q	00011101110101
U	0001110111011	0	00011101111	.	0001111
e	001	n	0100	h	0101
o	0110	g	011100	v	0111010
I	01110110	’	011101110	E	0111011110
R	0111011111	y	011110	c	011111
a	1000	l	10010	w	100110
,	100111	d	10100	:	10101000
\n{space}	10101001	9	10101010000	8	10101010001
3	1010101001	7	10101010100	6	10101010101
W	1010101011	T	101010110	D	1010101110
P	10101011110	q	10101011111	f	101011
t	1011	m	110000	1	110001000
;	110001001	-	110001010	B	1100010110
N	11000101110	5	11000101111	H	1100011000
j	11000110010	(	1100011001100	)	1100011001101
}	110001100111000	‘	1100011001110010	[	11000110011100110
]	11000110011100111	Z	11000110011101	K	1100011001111
L	1100011010	M	1100011011	"	110001110
A	110001111	r	11001	u	110100
b	1101010	p	1101011		
i	11011	{space}	111		

we find a letter. As an example, suppose we encoded the letters "abc". This would be represented as "10001101010011111". Then we can see that, taking the first three bits, "100" is not a letter, but adding another bit gives us the Huffman codeword for the letter "a". Similarly, if we take the next set of letters "110101", we do not have a letter. But if we add the next bit to this, it gives us the Huffman codeword for "b". Finally, our remaining string will decode to the letter c. This procedure is formalized by the following rules:

1. Given a Huffman encoded bit-string S, take a bit b from the front of S.

2. Continue taking bits from the front of S until it is a Huffman codeword.
3. If a potential codeword is longer than the longest codeword, then the encoding is not valid.
4. Otherwise, all of the string S will be consumed and the result will be a decoded set of letters.

4. NEW DECODING ALGORITHMS FOR CONVOLUTIONAL CODES

4.1 The Viterbi Algorithm

The Viterbi Algorithm is the most basic method of decoding a convolutional code. The main idea behind the algorithm is that we minimize the Hamming distance, or number of single-bit errors, between the input string and all of the potential Huffman codewords.

Operating on the lattice structure in Figure 1.2, the Viterbi Algorithm computes the shortest path from the node at stage 0, state 0 to the node at the last stage, state 0. The edges are weighted using the Hamming distance between n-bit pairs and the production bits from each edge, where $n = 1/r$, inverse of the rate of our convolutional code. Therefore, given an input string S of size k we can express the algorithm by the following:

1. For every edge i in every stage j of the lattice, take n bits, $X \subseteq S$ from S starting at index nj and assign a weight $w(e_{ij})$ to be the Hamming distance between X and the production on e_{ij}

2. Set every node x to have a value $v(x)$ of ∞ except for the node at stage 0, state 0, which we give a value of 0

3. For every node x, compute the node y connecting to it such that $v(y) + w(e_{y \to x})$ is minimized and store that node as the survivor of x

4. Once the the minimum node for the node in the last stage, state 0 is found, trace the minimum nodes back to the start to yield our decoded string of minimum Hamming distance

We examine the performance of the Viterbi Algorithm for computing the original string

Huffman codeword string we encoded by our convolutional code by comparing the percentage of single-bit errors between the two. The data is shown in Figure 4.1. To acquire this data, we followed the procedure:

1. Generate the Huffman code from the Gutenberg corpus texts
2. Select a text, encode it by the Huffman code
3. Select a random portion of the text
4. Encode the random portion of the text by the convolutional code to yield the convolution string
5. Introduce symmetric bit error to the convolution string at a probability p
6. Run the convolution string through the Viterbi Algorithm
7. Compare the resulting string to the original selection of the text bit-wise and compute the probability of seeing a bit error across the whole string; store this number
8. Repeat steps 3 - 7 many times
9. Repeat steps 2 - 8 many times using different probabilities of error p

4.2 k-Shortest Path Algorithm

Assuming the Viterbi Algorithm fails to find a shortest path containing valid words (i.e. both the bit-strings decode to valid letters, and the letters form valid words), the next step would be to check the next shortest paths.

Possibly the most prominent algorithm for computing so-called k-shortest paths in a graph is Yen's Algorithm. Using path $k-1$ to find path k, it works by creating a set of every possible single-edge deviation from path $k-1$ and using the minimum weight path

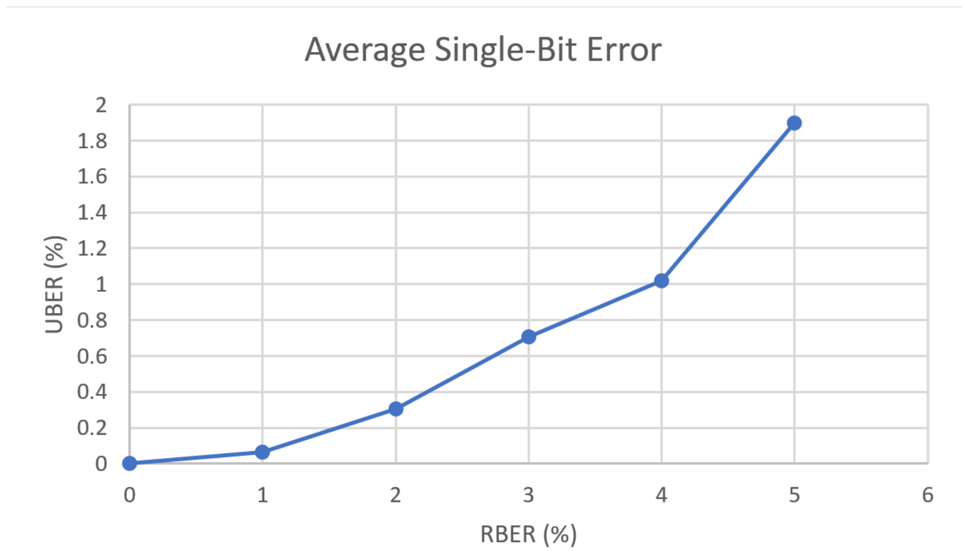

Figure 4.1: The average bit error between initial Huffman codeword string and the resulting Viterbi decoded string

from the set as path k. [23, 24, 25] For our purposes, this is extremely useful; since there are only ever two edges from any node, we have predictable complexity of the algorithm.

Given a lattice set up in the same way as previously, we can compute k-shortest paths by the following algorithm:

1. Run the Viterbi Algorithm once to compute the 1st shortest path p_1
2. Using p_1, run Yen's Algorithm for k paths to find the k-shortest path

The data in Figure 4.2 shows the performance of Yen's Algorithm vs. the Viterbi Algorithm. The procedure to collect this data was as follows:

1. Generate the Huffman code from the Gutenberg corpus texts
2. Select a text, encode it by the Huffman code
3. Select a random portion of the text

4. Encode the random portion of the text by the convolutional code to yield the convolution string

5. Introduce symmetric bit error to the convolution string at a probability p

6. Run the convolution string through the Viterbi Algorithm to yield the 1st shortest path

7. If all of the words in the 1st shortest path are valid, skip to the comparison step using the shortest path for both the Viterbi Algorithm and Yen's Algorithm.

8. Otherwise, we use Yen's Algorithm on the shortest path until one of two things occurs

   (a) a completely valid path is found

   (b) a cap on the number of paths to check is reached; in this case, we use the path with the maximum number of valid words

9. Compare the resulting strings to the original selection of the text bit-wise and compute the probability of seeing a bit error across each string; store these numbers

10. Repeat steps 3 - 7 many times

11. Repeat steps 2 - 8 many times using different probabilities of error p

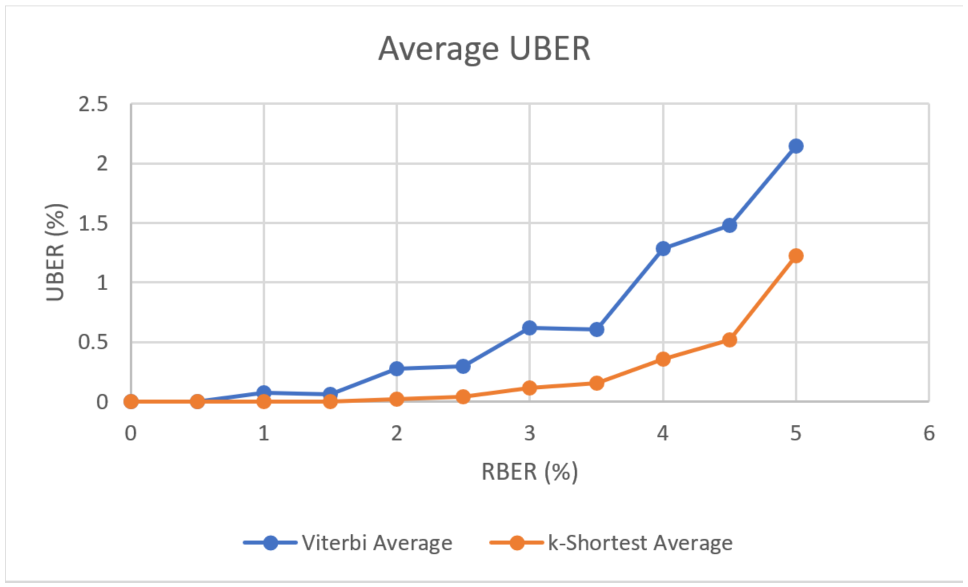

Figure 4.2: The performance of the k-Shortest Path Algorithm vs. the Viterbi Algorithm

The data in Figure 4.3 shows how many paths were required to find a completely valid path before reaching a cap number of paths. The data was collected as follows:

1. Generate the Huffman code from the Gutenberg corpus texts
2. Select a text, encode it by the Huffman code
3. Select a random portion of the text
4. Encode the random portion of the text by the convolutional code to yield the convolution string
5. Introduce symmetric bit error to the convolution string at a probability p
6. Run the convolution string through the Viterbi Algorithm to yield the 1st shortest path

7. If all of the words in the 1st shortest path are valid, our pathway was found in 1 step; store this value
8. Otherwise, we use Yen's Algorithm on the shortest path until one of two things occurs:
   (a) a completely valid path is found; store the number of paths checked
   (b) a cap on the number of paths to check, k is reached; store the number k
9. Repeat steps 3 - 7 many times
10. Repeat steps 2 - 8 many times using different probabilities of error p

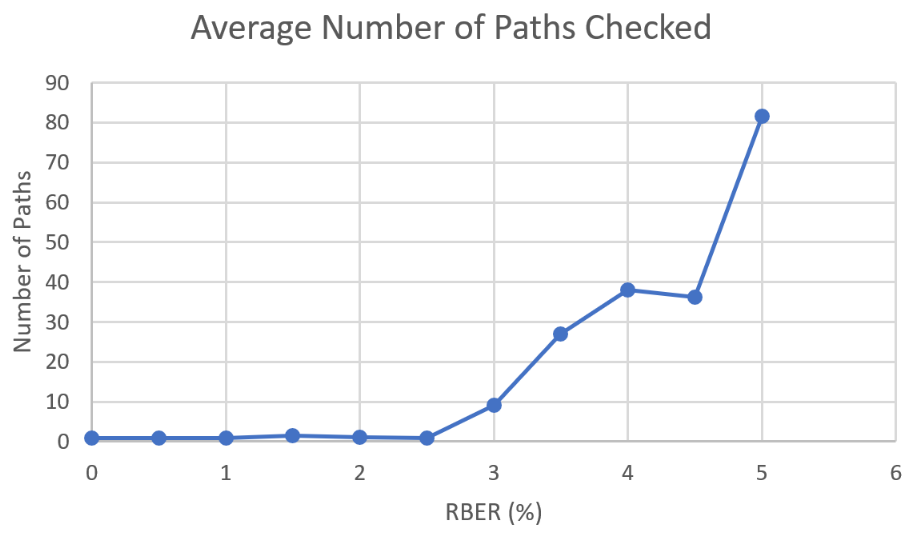

Figure 4.3: The performance of the k-Shortest Path Algorithm in terms of number of pathways checked

4.2.1 Validity

What does it mean to say that a path is "valid"? The definition of validity for our purposes is two-fold.

1. All the sets of bits represent valid letters by our Huffman code

2. All the collections of letters represent valid words by our dictionary

Therefore, in our decoding process, we can first decode from the shortest path to a string of characters. If any of the sets of bits was not valid, then the path is not valid. If it was valid, we can tokenize this string on spaces to yield what should be our words. If any one of these words, except for the first and last words, is not in our dictionary, then our path is not valid.

5. DECODING ALGORITHM AND PERFORMANCE FOR ASYMMETRIC ERRORS

One of the first things to notice about Yen's Algorithm is that while it is often very successful in finding valid codewords, it can take up to 2^n pathways to find the valid codeword, where n is the number of bits provided initially. Next, we can observe that as Yen's Algorithm computes pathways, it can come extremely close to an entirely valid pathway but end up creating more error than solving it. For example, suppose we initially encoded the phrase "the dog jumps over the moon," and after some computation with Yen's Algorithm, arrive at a pathway that reads "the dog jumps ovtr the moon." Obviously, the word "ovtr" should be the word "over." Our next pathway could yield another string "the drg jumda ovtr them moon," which has much more bit error than before. This is problematic in our attempt to derive an efficient decoding algorithm, since it makes little sense to move past pathways that we know are extremely close to correct.

5.1 Validating a Path in a Graph G

We first consider what makes a pathway valid, and how close to correct a valid section must be.

Suppose p is a path found in a stage-graph G represented as a list of nodes and $words$ is a list of words that we know to be valid. Also, define two numbers k_1 to be the single-word minimum length and k_2 to be the multi-word length. For example, if $k_1 = 3$, then the word "bath" satisfies it; if $k_2 = 4$, then the sentence "this is a short sentence" satisfies it. If this is the first path found (i.e. shortest path), we validate this pathway by the following procedure:

1. Convert the edges for each node $p[i]$ and $p[i + 1]$ in the pathway into the bits they

represent

2. Use the usual method of decoding a Huffman codeword to convert the bits into words

3. If we do not find a valid Huffman codeword, return a completely invalid pathway. Otherwise, let $tokens$ be the tokenization of the string returned by our Huffman decoding algorithm

4. Let $valids$ be an empty list of valid regions and let $i = 0$ be an index into $tokens$

5. While $i < len(tokens)$ do

    (a) If $tokens[i]$ and the next $k_2 - 1$ words are valid, add them to $valids$ as a valid region and increment i by k_2

    (b) If $tokens[i]$ is valid and $len(tokens[i]) > k_1$, add it to $valids$ as a valid region and increment i by 1

    (c) Otherwise, add $tokens[i]$ as an invalid region and increment i by 1

6. Return the list of valid regions

We can also decide if a pathway is valid in a very simple manner. We consider the case in which the first and last words in the path are inherently invalid, i.e. the pathway begins and ends in the middle of a word. Let p be a pathway.

1. Tokenize p into a list $tokens$

2. Let $b = true$ be a boolean

3. For each t in $tokens$ do:

    (a) If t is the first token and the path contains the first node of the graph, skip

    (b) If t is the last token and the path contains the last node of the graph, skip

    (c) Otherwise $b\& = t$ is valid

4. Return b

This procedure ensures we can validate sub-paths as well as full pathways through the graph structure.

5.2 k-Shortest Sub-Path Algorithm

Our algorithm exploits an important property of decoding with a Huffman code and a convolutional code: we can be sure of certain words that are valid in the Huffman code via decoding by the convolutional code. Our intuition here is that when we check any path for valid words, we can store the sections of the path we know to be valid and avoid computing over these sub-paths again.

We leverage the structure of the Yen's Algorithm to compute sub-path shortest paths. Let G be a graph and let k be a threshold number of paths we wish to check.

1. Find the shortest path p_0 in the G by the Viterbi Algorithm
2. Let $valids$ be the result of the validation procedure on p_0
3. Let $totalPath$ be an empty list
4. For each region r in $valids$ do:
    (a) If r is valid, simply add the region to $totalPath$
    (b) If r is invalid, run Yen's Algorithm on the region until a valid pathway is found or the threshold k is reached, then add the result to $totalPath$
5. Finally, return the resulting path

Let M be the number of stages in the graph, N be the number of states per stage, and k be the threshold number of paths we wish to check. The time complexity of the Viterbi Algorithm is $O(MN)$. The time complexity of Yen's Algorithm is $O(kM^2N)$ Finally, we see that the k-Shortest Sub-Path Algorithm runs in $O(kM^2N)$ time since in the worst case it simply runs Yen's Algorithm on the entire graph.

5.3 Results

Using the usual procedure for collecting data, we compared the average single-bit error of the Viterbi Algorithm, Yen's Algorithm, and the k-Shortest Sub-Path Algorithm. The only structural change for the experiment was adding the k-Shortest Sub-Path to the procedure. The results of this experiment are in Figures 5.1 and 5.2.

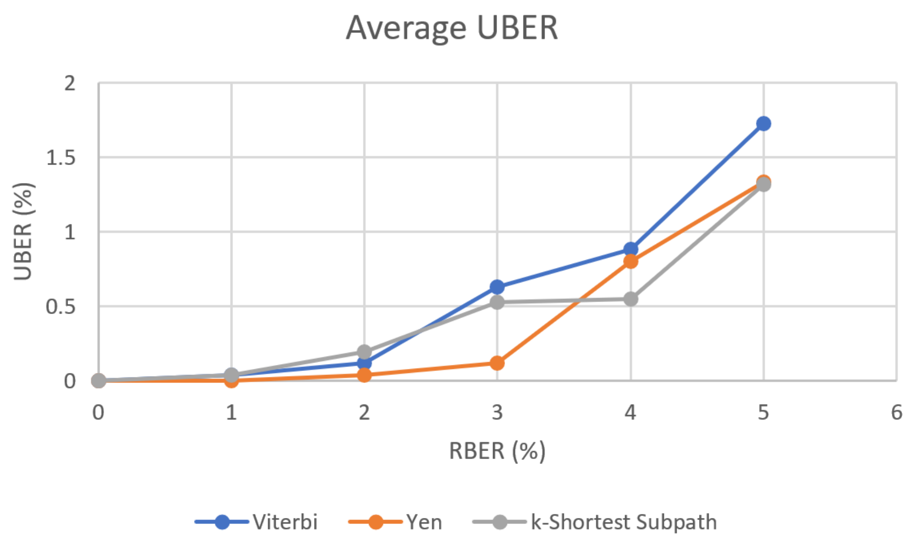

Figure 5.1: The average single-bit error across our three algorithms with RBER from 0% to 5%

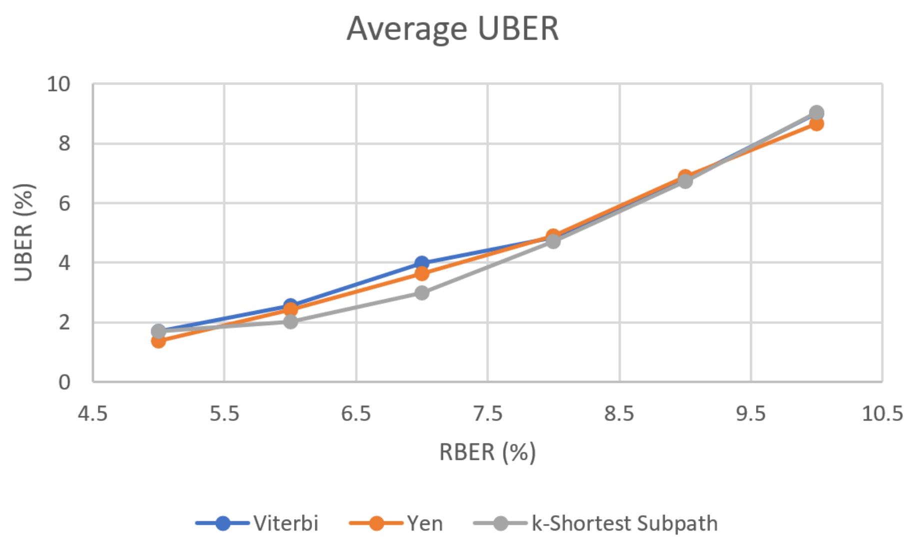

Figure 5.2: The average single-bit error across our three algorithms with RBER from 5% to 10%

From these results, we see that with our current k-Shortest Sub-Path Algorithm, the performance is not much better than Yen's Algorithm, and can often be worse. On lower percentages, we see that Yen's Algorithm outperforms the k-Shortest Sub-Path Algorithm consistently, but at higher percentages, the k-Shortest Sub-Path Algorithm runs as accurately as Yen's Algorithm.

6. CONCLUSIONS AND FUTURE DIRECTIONS

In developing our algorithm, our main consideration was computing less of a pathway than Yen's Algorithm, and we hoped to perform better from this method. Our approach is on the naive side of this concept, however. In particular, we use Yen's Algorithm on sub-paths of the first shortest path until they either find a valid sub-path or reach a threshold value. A potential source of error in this step is loss of path-context in these sub-paths.

Another issue that is easy to identify is that our language processing is rudimentary. The algorithm would greatly benefit from some form of natural language processing, especially for tokenizing and classifying individual words as parts of the path or not. The validation procedure would be more robust, and the verification of a string as valid would be much more accurate with the addition of NLP techniques. [26]

With regards to the first issue, we can see that our current usage of Yen's Algorithm in the k-Shortest Sub-Path Algorithm may not be the best approach. One idea for improving this is to modify Yen's Algorithm to be a "next shortest path" algorithm instead of computing all k shortest paths. In this way, we could implement a depth-first style subgraph validation procedure, in which once a sub-path has valid sub-paths, we can ignore our new valid pieces.

We can see from 1.2 that the base structure for the Viterbi Algorithm resembles the structure of a neural network. This begs the question: how can we apply deep learning principles to the subgraph search we implemented? In the future, the accuracy of the solutions returned by Yen's Algorithm could be refined with deep learning concepts such as node pruning. Even further, the conceptualization of data storage as we have proceeded could be abstracted further to a purely deep neural network approach. A network could be trained to behave as a "more accurate" Viterbi Algorithm, which is one of the overarching goals of this research.

Finally, we should once more address the base issue here: the memristor crossbar array. Our abstraction from the physical concept to a simple send/receive problem across a channel has proven useful and yielded some interesting results. The Viterbi Algorithm, while useful in theory, would not be practical for use in a memristor array, since it is not accurate enough. Yen's Algorithm, on the other hand, presents a much more promising approach in terms of accuracy. Then next thing to consider with Yen's Algorithm is speed; the Viterbi Algorithm is quite slow. Yen's Algorithm depends entirely on the base shortest-path algorithm for its running time, so using a faster, more state-of-the-art algorithm would increase the speed. The k-Shortest Sub-Path Algorithm, while a good idea in our theoretical application, does not produce the results necessary to pass Yen's Algorithm applied to our particular problem. Therefore, more development and thought is needed into the structure of the stage graph and the abilities of the Viterbi Algorithm (or other shortest path algorithm).

In summary, we present several algorithms with the potential to solve the original sneak path error: the Viterbi, Yen, and k-Shortest Sub-Path Algorithms. While they all are interesting from a theoretical standpoint in terms of error correction, it is Yen's Algorithm that seems to have the best results. In fact, we see that Yen's Algorithm yields less than 1.5% error when tested on representations of data stored in memory. To improve Yen's Algorithm, more advanced techniques may be required, including natural language processing and deep learning.

REFERENCES

[1] I. Poole, "What is Flash Memory?." `http://www.radio-electronics.com/info/data/semicond/memory/what-is-flash-memory-basics-tutorial.php`.

[2] H. S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase Change Memory," *Proceedings of the IEEE*, vol. 98, pp. 2201–2227, Dec 2010.

[3] "Altered States." `http://www.economist.com/node/21560981`.

[4] "Racetrack Memory: The Future of Data Storage." `http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/racetrack/`.

[5] S. P. Mohanty, "Memristor: From Basics to Deployment," *IEEE Potentials*, vol. 32, pp. 34–39, May 2013.

[6] A. Mills, "Making a new generation of memristors for digital memory and computation." `https://phys.org/news/2016-02-memristors-digital-memory.html`.

[7] L. Chua, "Memristor-The missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, pp. 507–519, September 1971.

[8] MIT, "Convolutional Coding." `http://web.mit.edu/6.02/www/f2010/`

`handouts/lectures/L8.pdf`.

[9] E. W. Weisstein, "Markov Process." WolframMathWorld.

[10] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, "Sneak-path constraints in memristor crossbar arrays," in *2013 IEEE International Symposium on Information Theory*, pp. 156–160, July 2013.

[11] M. A. Zidan, H. A. H. Fahmy, M. M. Hussain, and K. N. Salama, "Memristor-based memory: The sneak paths problem and solutions," *Microelectronics Journal*, vol. 44, no. 2, pp. 176 – 183, 2013.

[12] M. A. Zidan, A. M. Eltawil, F. Kurdahi, H. A. H. Fahmy, and K. N. Salama, "Memristor Multiport Readout: A Closed-Form Solution for Sneak Paths," *IEEE Transactions on Nanotechnology*, vol. 13, pp. 274–282, March 2014.

[13] C. M. Jung, J. M. Choi, and K. S. Min, "Two-Step Write Scheme for Reducing Sneak-Path Leakage in Complementary Memristor Array," *IEEE Transactions on Nanotechnology*, vol. 11, pp. 611–618, May 2012.

[14] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, "Memristor Based Computation-in-memory Architecture for Data-intensive Applications," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, (San Jose, CA, USA), pp. 1718–1725, EDA Consortium, 2015.

[15] J. K. Wolf, "An Introduction to Error Correcting Codes Part 1."

`http://circuit.ucsd.edu/~yhk/ece154c-spr17/pdfs/ErrorCorrectionI.pdf`.

[16] A. A. Jiang, P. Upadhyaya, E. F. Haratsch, and J. Bruck, "Correcting Errors by Natural Redundancy," *Proc. Information Theory and Applications (ITA)*, 2017.

[17] A. A. Jiang and J. Bruck, "Is There A New Way to Correct Errors," *Proc. Information Theory and Applications (ITA) Workshop*, 2015.

[18] A. A. Jiang, P. Upadhyaya, E. F. Haratsch, and J. Bruck, "Error Correction by Natural Redundancy for Long Term Storage," *Proc. Non-Volatile Memories Workshop (NVMW)*, March 2017.

[19] MIT, "Viterbi Decoding of Covolutional Codes." `http://web.mit.edu/6.02/www/f2010/handouts/lectures/L9.pdf`.

[20] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[21] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, pp. 260–269, April 1967.

[22] S. Bird, E. Loper, and E. Klein, *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

[23] M. Sherwood, L. Gewali, H. Selvaraj, and V. Muthukumar, "A fast and simple algo-

rithm for computing m shortest paths in stage graph," vol. 30, 01 2004.

[24] D. Eppstein, "Finding the K Shortest Paths," *SIAM J. Comput.*, vol. 28, pp. 652–673, Feb 1999.

[25] J. Y. Yen, "Finding the K Shortest Loopless Paths in a Network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[26] A. A. Jiang, Y. Li, and J. Bruck, "Error correction through language processing," *Proc. IEEE Information Theory Workshop (ITW)*, 2015.

[27] J. Tyson, "How Flash Memory Works." `http://computer.howstuffworks.com/flash-memory.ht`.