ADAPTIVE SINGLE-PASS COMPRESSION OF UNBOUNDED INTEGERS

by

Christopher Rice Hyatt B.B.A

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2017

Committee Members:

Dan Tamir, Chair

Yan Yan

Apan Qasem

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

I would like to dedicate this thesis to my family and friends that had to deal with me through this adventure. I would also like to thank my professors and advisors for their guidance.

ACKNOWLEDGEMENTS

I would like to thank Dr. Tamir and all of the Texas State staff for their continued support in my education.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS v

LIST OF FIGURES vii

ABSTRACT ix

CHAPTER

I. INTRODUCTION 1

II. BACKGROUND 3

2.1 Data Compression 3

2.2 Types of Compression 5

2.3 Compression Algorithms 7

2.4 Search Engine Indexing 16

2.5 Distributions 17

III. LITERATURE REVIEW 18

IV. EXPERIMENTAL SETUP 21

V. EXPERIMENTAL RESULTS 28

5.1 Experiment 1: Compressing synthetic data using excess tree growth (parsing one byte) 28

5.2 Experiment 2: Compressing Silesia data using excess tree growth (parsing one byte) 30

5.3 Experiment 3: Compressing synthetic data using excess tree growth (parsing two bytes) 32

5.4 Experiment 4: Compressing data using excess tree growth (parsing whole 32-bits) 33

5.5 Experiment 5: Compressing synthetic data using probability-limited tree growth (parsing one byte) 38

5.6 Experiment 6: Compressing Silesia data using probability-limited tree growth (parsing one byte) 41

5.7 Experiment 7: Compressing synthetic data using probability-limited tree growth (parsing two byte) 44

5.8 Experiment 8: Compressing data using probability-limited tree growth (parsing whole 32-bits) 46

VI. RESULTS EVALUATION 51

VII. CONCLUSION AND FUTURE RESEARCH 54

REFERENCES 56

LIST OF FIGURES

Figure **Page**

1. Figure 1: Tunstall tree example....7
2. Figure 2: Excess tree growth....11
3. Figure 3: Probability-limited tree growth tree example....12
4. Figure 4: δ-T with the excess tree growth process....14
5. Figure 5: Synthetic results using excess tree growth and parsing with one byte....28
6. Figure 6: Silesia results using excess tree growth part A....30
7. Figure 7: Silesia results using excess tree growth part B....31
8. Figure 8: Synthetic results using excess tree growth and parsing with two bytes....32
9. Figure 9: Synthetic results using excess tree growth and parsing with four bytes....34
10. Figure 10: Wikipedia results using excess tree growth and parsing with four bytes....35
11. Figure 11: File statistics for the Wikipedia gap dataset....36
12. Figure 12: Synthetic results using probability-limited tree growth and parsing with one byte....39
13. Figure 13: Silesia using the probability-limited tree growth method part A....42
14. Figure 14: Silesia using the probability-limited tree growth method part B....43
15. Figure 15: Synthetic results using probability-limited tree growth and parsing with two bytes....45
16. Figure 16: Synthetic results using probability-limited tree growth and parsing with four bytes....47

17. Figure 17: Synthetic results using probability-limited tree growth and parsing with four bytes....49
18. Figure 18: Statistics for Wikipedia gap data....50

ABSTRACT

Due to webpage creation, data gathering and storage, and the increasing data needed for machine learning, the amount of data in the world is rapidly growing. Organizations such as Google, Wikipedia, and the National Security Agency (NSA) use techniques to convert all of this data into searchable indexes of references. These indexes are growing as quickly as the data used to create them, and are an equal subject for compression.

This research explores the ability to dynamically compress data in one pass. This, effectively improves latency and throughput. To achieve dynamic compression in one pass, the combination of Tunstall and two other compression algorithms, Elias-Delta code and a variation of Group VarInt, are used. The two resulting pairs are Variable length nibbles with Tunstall (VLNT) and Delta-Tunstall (δ-T). This is the first known attempt at compressing data using these algorithm pairs.

These compression algorithms are applied to a number of different datasets. A synthetic dataset and a Wikipedia dataset are used to test the algorithms' ability to compress integers. The synthetic dataset is created using several probability distribution functions (PDF), such as geometric and Poisson. Meanwhile, the Wikipedia dataset is acquired from actual inverted indexes for a number of different Wikipedia search terms. VLNT and δ-T are also used to compress the members of the Silesia benchmarks dataset.

VLNT and δ-T show promise as good platforms for data compression and are recommended for future research focusing on single-pass compression of unbounded datasets.

CHAPTER 1

INTRODUCTION

In 2014, the Guardian reported that the NSA collects approximately 200 million text messages every day from locations around the world [1]. Presumably, the data is stored for later use in a number of treat analyses. These datasets need to be readily accessible to be useful in analytics, which means that effective searching and indexing techniques have to be in place.

These indexing techniques are established at companies such as Google and Bing to create index mappings of search terms. As of June 28, 2017, there are estimated to be at least 4.8 billion indexed webpages [2], a number that continues to grow as more webpages, blogs, and video content companies produce content.

Forbes and the International Data Corporation (IDC) predict that by 2019 the internet of things (IoT) and digital transformations for business will be supported by some form of artificial intelligence [3]. Further predictions state that by the same year, 110 million consumer devices with intelligent assistance will be installed in U.S. households. As artificial intelligence develops, the demand for data to train and refine the machine learning process will grow alongside it [4].

The rapid growth in stored and transmitted content is creating a challenge for organizations such as Google, the NSA, and Wikipedia to store index mappings for search terms to webpages. Fortunately, the inverted index algorithm makes these mappings possible [5]. However, inverted indexes require a large storage location as the number of webpages

and the amount of data grow. This thesis addresses the compression of these inverted indexes and other kinds of input streams.

This thesis presents two solutions (the VLNT and the δ-T compression algorithms) that address the challenge of compressing data in an age of rapid growth and multiple input formats. Both VLNT and δ-T borrow from Tunstall and Elias while expanding on the work of Anh, Glory, and Mulpuri in the area of compressing inverted indexes [6, 7, 8, 9, 10]. VLNT uses a variation of Group VarInt and Tunstall, while δ-T combines Elias-Delta and Tunstall. Both of these solutions explore variations of the Tunstall tree growth, while two other methods, excess tree growth and probability-limited tree growth, expand on the original Tunstall method. VLNT and δ-T expand Anh, Glory, and Mulpuri's work with inverted index compression by defining an adaptive unbounded approach.

The hypothesis guiding this thesis is that the combination of these algorithms enables an effective one-pass unbounded integers compression algorithm where effectiveness is measured by bit rate and compression ratio as well as potential for high throughput and low latency implementation.

The main contribution of this thesis is the introduction of two semi-fixed [1]compression algorithms that are not limited by finite alphabets. Additionally, the algorithms achieve high compression ratios using only one pass for multiple types of input sources. To the best of our knowledge this research is the first to address compression using VLNT and δ-T.

Three assumptions are made in this study: First, the input streams are memoryless. Second, there is no specific knowledge about the data stream before compression. Third,

[1] The term semi-fixed refers to compression that tends to assign fixed length code to symbols as system information is accumulated.

because the algorithm produces a uniquely decodable output stream decompression is well defined. Additionally, the encoder and decoder are in sync allowing the decoder to have the latest symbol lookup tree.

The solutions developed in this thesis perform well on a wide range of data, in some cases compressing within .28 bits per integer of the theoretical lower bound. In other cases, the solutions use half the number of bits to compress the same data as the industry standard, Group VarInt.

The rest of this paper is organized as follows: Chapter 2 summarizes the terms used throughout the study, and Chapter 3 is an overview of the research. Chapter 4 outlines the experiments while Chapter 5 examines the experimental results, and Chapter 6 evaluates the results. Finally, Chapter 7 concludes and considers future research.

CHAPTER 2

BACKGROUND

This chapter defines the terms that are used throughout this paper.

2.1 Data Compression

Data compression is the process of taking an input dataset and converting it into a smaller dataset. David Salomon, jokingly, suggests data compression is important because people are both hoarders and impatient [11]. This section defines a number of common terms related to data compression.

2.1.1 *Communication Bit Rate and Compression Ratio*

Communication bit rate is defined as the average number of bits allocated to transmitted symbols it can be approximated as $BR = \frac{total\ number\ of\ bits\ sent}{number\ of\ input\ symbols}$. For example, if it takes 3,000 bits to send 200 symbols, the bit rate would be 15 bits per symbol.

The compression ratio is defined as $CR = \frac{size\ of\ input\ stream}{size\ of\ output\ stream}$ [11]. For example, the initial input stream is 10,000 bits. The compressed output stream is 5,000 bits. Then the compression ratio is 2.

2.1.2 *Entropy*

Entropy is the theoretical lower bound on communication bit rate over a noiseless communication channel. Source X's entropy is defined by the equation: $E = \sum_1^n p_i\ log_2\left(\frac{1}{p_i}\right)$, where $\{p_1, \ldots, p_n\}$ are the probabilities of occurrence of source X's alphabet $A = \{a_1, \ldots, a_n\}$ [8].

2.1.3 *Dispersion*

Dispersion is defined by the equation: $D = \frac{Number\ of\ unique\ input\ symbols}{total\ number\ of\ source\ symbols}$. Dispersion attempts to describe the redundancy of a particular information source.

2.1.4 *Memory and Memoryless Sources*

An input data source is assumed to belong to one of two categories [11]. The first assumption type is that sources have memory. For a source to have memory it is assumed that any given symbol is dependent on the preceding symbols in the input stream. For example, the game blackjack is a source with memory. The odds of the next card in the deck change as more cards are dealt. Meanwhile, in the second assumption type, sources are memoryless. Memoryless sources are where any given symbol is independent of any other proceeding symbol. Ideal Roulette is an example of a memoryless source. For example, the chance of the ball landing on nine is the same no matter what number came before it.

2.1.5 *Assigning Probabilities to Symbols*

There are three main techniques used to gather the probabilities of the symbols: static codes, multi-pass algorithms, and adaptive methods. In static codes, a permanent table of probabilities is attained from a given set of training data. Meanwhile, the multi-pass algorithm parses the source at least twice – the first time to create the probability table, and the second time to perform the actual encoding. Finally, the adaptive method updates the probability table as it encodes.

2.2 Types of Compression

2.2.1 *Fixed-Length Codes*

Fixed-length codes (FLC) are also called block codes [12]. The benefit of fixed-length codes is their ease of use for computers; for instance, it is easier for a computer to decode fixed-length codes because it is able to parse the same length over the entire encoded data. The worst case when using fixed length codes results from not having the ability to assign a smaller length code to the most probable symbol.

2.2.2 *Variable-Length Codes*

Variable-length codes (VLCs) are codes that can vary in length [12]. Their advantage is that they enable allocating the smallest possible codes to the most common symbols, which, on average, enables better compression ratios. For example, Morse code assigns a single dot to the letter "e". The least probable inputs are assigned longer symbols.

To better understand the difference between FLC which caters to the worst case and VLC which addresses the average case, consider the following example. Given an alphabet, $A = \{X, Y, Z\}$ with the probability set $P, \{.90, .05, .05\}$. A fixed length code would assign a 2-bit code to each of the symbols in the alphabet. A variable length code might assign one

bit to X (e.g., 0) and two bits to Y, and Z (e.g., 10 and 11 respectively); thereby reducing the average bit rate. Hence, fixed length encoding of a 100-symbol source results in a 200-bit output. However, the variable length encoding assignment is likely to result in a compressed output of 110-bits.

2.2.3 *Fixed-to-Variable Compression*

David Salomon argues that fixed-to-variable encoding is the most significant compression method [12]. Assigning smaller codes to the most probable symbol leads to high compression ratios. A common algorithm in this category is Huffman [12].

2.2.4 *Variable-to-Fixed Compression*

Variable-to-fixed compression parses the source data into chunks with various sizes, and encodes each chuck into a fixed code size [12]. Tunstall is an example of a variable-to-fixed length compression algorithm. This provides hardware implementation advantage.

2.2.5 *Variable-to-Variable Compression*

Variable-to-variable compression is a combination of two different variable-length compression algorithms, whereby variable lengths of the source data are encoded with VLCs [12]. Its most common application is when combining different compression algorithms.

2.2.6 *Semi-Fixed Compression*

Semi-fixed compression is a concept that is used to explain the benefits of compressing using both a variable and fixed compression algorithm. For example, this paper produces a hybrid of Tunstall, a fixed-length code, and Elias-Delta, a variable-length code. In general, compression becomes more fixed than variable over time as more symbols become encoded. Decoding is easier as the length of the codes is fixed. Hence, overtime semi-fixed encoding reduces time spent decoding.

2.3 Compression Algorithms

This section discusses the different compression methods used to form the algorithms in this paper, and explains the details of the algorithms.

2.3.1 *Tunstall*

Tunstall coding is the base of the algorithms presented in this thesis, and is discussed throughout the research on data compression. Here, Tunstall defines an efficient method for creating a code tree for use in variable-to-fixed length compression. Tunstall's code tree is created by extending and grouping symbols on the basis of probabilities [9]. For example, given the alphabet $A = \{X, Y, Z\}$ with the probability set $P, \{.50, .30, .20\}$, the corresponding process of tree creation is displayed in Figure 1.

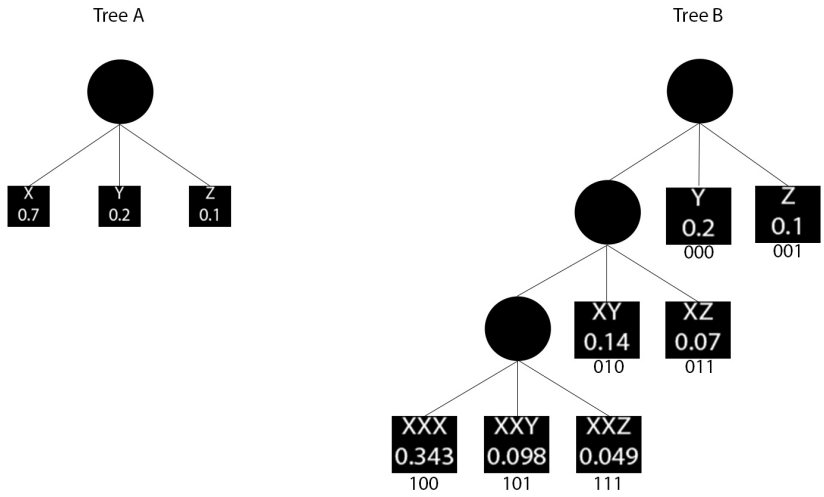

Figure 1: Tunstall tree example

In this example, the node of the tree with the largest probability is extended and removed from possible codes until the number of nodes equals 2^n, where n is the fixed length of the compression bit string.

2.3.2 *Elias-Gamma Code*

Elias-Gamma code is a VLC compression algorithm for unbounded non-negative integers that enables the compression of integers of unknown size. For instance, the range of numbers that fixed-length encoding is able to encode is dictated by the dataset's largest symbol value. In contrast, VLCs such as Elias-Gamma code remove the size constraint set by fixed-length encoding [10].

Given a non-negative integer, x, the Elias-Gamma representation is broken down into two parts. The first part is composed of $|\beta(x)| - 1$ zeros, where $|\beta(x)|$ is the length of the bit string representing the binary equivalent of x, $\beta(x)$. The second part is $\beta(x)$. The final compressed bit string is a number of zeros followed by the binary representation of the given number [10].

For example, the Elias-Gamma code for 12 is 0001100. In this case, $\beta(12) = '1100'$. Hence, $|\beta(x)| = 4$. The first three zeros result from $|\beta(12)| - 1$. The following set of bits, 1100, is $\beta\ (12)$. Finally, the concatenation of the two strings is 0001100. It should be noted that this code is uniquely decodable. Furthermore, in this example, compression would be achieved if the fixed length coding is assuming 32-bit integers whereby 12 is represented with 28 leading zeros followed by $\beta(12)$.

2.3.3 *Elias-Delta Code*

Elias-Delta code further encodes the Elias-Gamma code for a given non-negative number, x. In general, the Elias-Delta code replaces the first field of zeros of the Elias-Gamma code with the Elias-Gamma code of that field [10].

For example, here are the steps to find the Elias-Delta code when $x = 12$. First, find the Elias-Gamma code of x, 0001100. Next, replace the leading 000 part with the Elias-

Gamma code of 4, which is 00100. Finally, remove the most significant bit of the second part of the 0001100, (1100), and append to the second step's result. This results in 00100100. Elias-Delta code outperforms Elias-Gamma code for large integers (in terms of bit rate). Elias-Omega code expands Elias-Delta code by iteratively operating on the leading 0s field. It outperforms (in terms of bit rate) Elias-Delta code for very large integers. Practically, considering computational complexity and compression efficiency, Elias-Delta code might be a good compromise. For this reason, it is selected for use in this thesis. Additionally, it should be noted that the Elias-Delta code asymptotically achieves the theoretical bit rate limit [10].

2.3.4 *Group VarInt*

Group VarInt (GVAR), as discussed by Jeff Dean, a senior fellow at Google and lead designer for five of Google's indexing schemes, in his presentation, encodes four integer values into 5 to 17 bytes [13, 14]. This creates a byte-aligned variable code. The concept is to compress and group four 32-bit integers with the least number of bytes that are required to represent the given symbols. The first step is to assign a tag to each number based on the minimum number of bytes that represent the number; the tags 00, 01, 10, and 11 represent 1-byte, 2-byte, 3-byte, and 4-byte minimums. For example, 00 is assigned to 31 because one byte represents the truncated number. Next, the tags are ordered into a bit string header according to the sequence in which each of the numbers is read. Each of the truncated numbers is then truncated into a bit string. Finally, the header and truncated number strings are combined.

For example, encoding a list of integers A, where $A = \{31, 80, 255, 320\}$, results in the bit string: 00000001 00011111 01010000 11111111 00000001 01000000. The first three numbers are represented by 1-byte strings, and the last number, 320, is represented by a 2-

byte string. Comparing to VLC with 32-bit integers this encoding archives a compression ratio of $\frac{32\times4}{8\times6} = 2.\overline{6}$.

2.3.5 *Excess Tree Growth*

The excess tree growth algorithm, developed for this research effort, is similar to Tunstall's tree extension algorithm. The first step is to determine the minimum number of bits needed to represent all of the unique symbols in the tree. For example, if the algorithm has seen five unique symbols, then a total of three bits is needed to represent all five symbols in the tree. However, three bits encode eight symbols. Using the three extra codes, the algorithm grows the tree by three nodes to make efficient use of the excess code space. The parent node of the new nodes is determined by finding the leaf node with the maximum probability. Finally, the found parent node is extended by the lesser of either the difference between the total number of codes allowed by number of code bits or the number of unique symbols. If there are unused code combinations, then the leaf node with the next highest probability is extended.

Figure 2 shows the final tree for a source with only five unique symbols. The three nodes representing multiple symbols fill in the excess room.

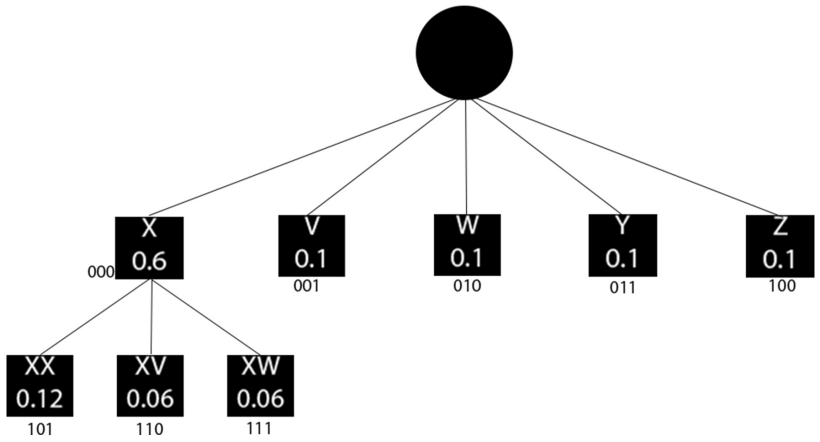

Figure 2: Excess tree growth

The generalized steps begin by setting the number of bits, N, to be equal to the minimum that satisfies $X \leq 2^N$, whereby X is the number of unique symbols seen. If $X < 2^N$, then the highest probable tree nodes are filled until $T = 2^N$, whereby T is the number of nodes in the tree. If multiple nodes with the same probability exist, lexicographical order is used. Tunstall follows the same process; however, if all combinations of parent node and distinctive symbols are represented in the children, it deletes a parent node. The excess tree growth algorithm keeps the parent nodes in case an unknown symbol follows the symbol represented by the parent node.

2.3.6 *Probability-Limited Tree Growth*

The probability-limited tree growth algorithm is another extension to Tunstall coding developed for this thesis. It grows the tree until a preset probability threshold is met, as illustrated by Figure 3. For example, given an alphabet $A = \{X, Y, Z\}$ with probabilities $P = \{.50, .30, .20\}$ and a threshold of .20:

In this example, tree A follows the completion of the threshold process. Notably, all of the leaf node probabilities are less than or equal to the threshold limit, .20. Tree B follows the application of the excess tree growth method to tree A. If the probabilities are the same, then the nodes are sorted in lexicographical order.

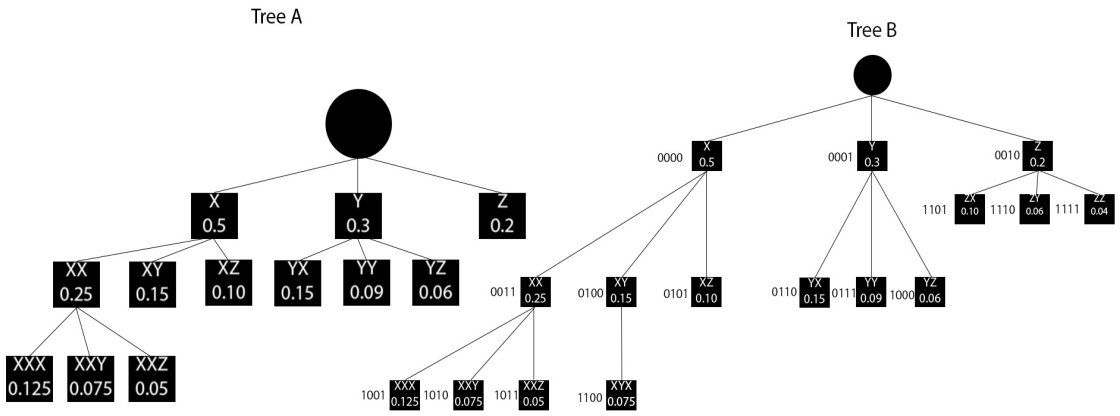

Figure 3: Probability-limited tree growth tree example

2.3.7 *δ-T*

δ-T is a combination of Elias-Delta and Tunstall compression algorithms. Elias-Delta is used to uniquely encode unknown data (e.g., unbounded integers) and Tunstall is used to assign code values to Delta-encoded data. The combination allows for a one-pass compression of unbounded integers.

Two changes are made to both the Tunstall tree growth algorithms and Elias-Delta algorithm to allow for the compression of unbounded integer data in one pass. First, the

value 1 is added to each of the unknown data inputs to be used to encode with Elias-Delta. This gets around the Elias-Delta constraint of numbers greater than 0. Second, one or more nodes of the Tunstall tree are reserved to be used as an exception codes. An exception code is used to tell the decoder that the code following the exception code has not occurred before and is encoded with Elias-Delta.

The following example is the process of encoding the input data, {0, 2, 4, 0, 0} using the excess tree growth method. However, the probability-limited tree growth can also be used to grow the Tunstall tree.

The first data input to be processed is 0. In order to use Elias-Delta, a 1 is added to the input and 1 is now used to represent 0. Because 0 is unknown to the encoder, an exception code and δ(0+1) are used to represent 0. Next, the Tunstall tree is updated with the following nodes {0:0, EXP:1}. EXP is used to represent the exception code. At this point the EXP is represented by 1. The next input read is 2, which is unknown. The encoder sends the EXP and δ(2+1). In this case, the encoded bit string is 10101. The Tunstall tree contains {0:00, 2:01, 00:10, EXP:11}. Next, 4 is read. The exception code, 11, and δ(4+1) are used to encode the 4. The Tunstall tree contains {0:00, 2:01, 4:10, EXP:11}. Next, 0 is read again. The node representing 0 is found in the tree. Because 0 is a known, the next input is read to look for sequences that may exist in the tree. In this case, the next input is 0 and the sequence 00 is not found in the tree. The bit string 00 is used to encode the input sequence 0 and is sent twice. Finally, the Tunstall tree is updated with the new probabilities of 0. Figure 4 illustrates the process of encoding the sequence of input data above. Figure 4.A shows the tree after the first read of 0. Figure 4.B shows the tree after reading 2. Figure 4.C shows the tree after reading 4. Figure 4.D shows the final tree after reading in the entire sequence.

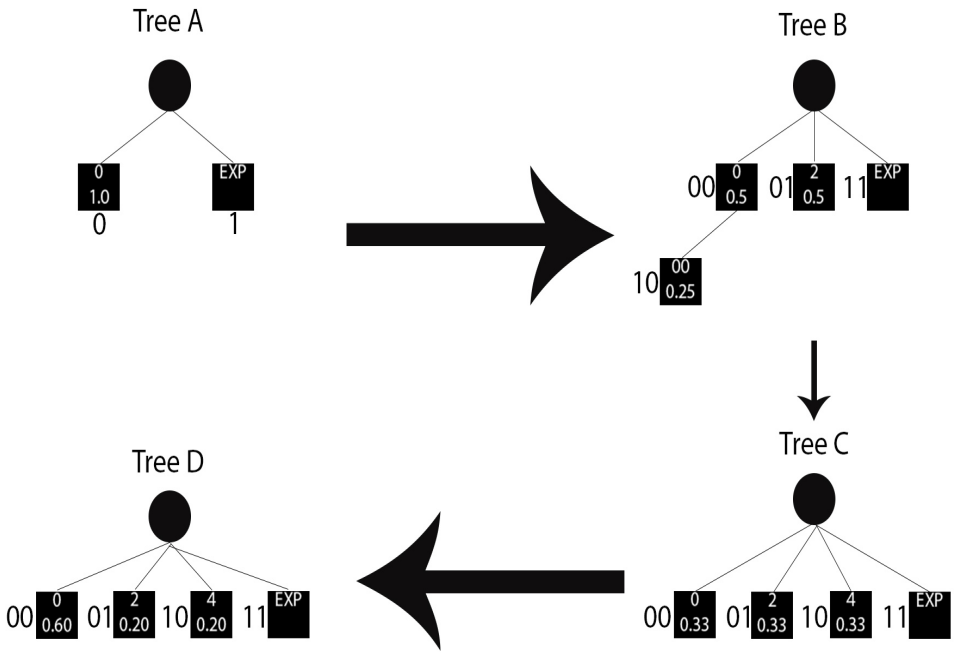

Figure 4: δ-T with the excess tree growth process

2.3.8 *VLN*

A nibble is a half of a byte or four bits. Variable-length nibbles (VLN) work like Group VarInt, but align on the nibble rather than on the bytes. The concept is to compress and group of four 32-bit integers with the fewest number of nibbles that are required to represent the given symbols. The first step is to assign a tag to each number based on the minimum number of nibbles that represent the number; the tags 00, 01, 10, and 11 represent 1-nibble, 2-nibbles, 4-nibbles, and 8-nibbles minimums per integer. For example, 00 is assigned to 15 because 1 nibble represents the truncated number. Next, the tags are ordered

into a bit string header according to the sequence in which each of the numbers is read. Each of the truncated numbers is then truncated into a bit string. Finally, the header and truncated number strings are combined. The header code 00 represents values of less than 16. The header code 01 represents values that are less than 256 but more than 15. The header code 10 represents values that are less than 65,792 but more than 255, and the final header code, 11, represents values greater than or equal to 65,792, respectively. The leading bit for numbers between 65,536 and 65,791 is dropped. For example, 65,536 represented in 32-bit form is 0000 0000 0000 0001 0000 0000 0000 0000 and 65,791 in 32-bit form is 0000 0000 0000 0001 0000 0000 1111 1111. If the encoder sends 10 and there is no bit set in the 3rd and 4th nibbles, then the decoder knows to set the 16th bit when decoding.

For example, encoding a sequence P, where P = {1, 255, 300, 80,000}, with VLN results in the following bit strings: 00011011 0001 11111111 0000000100101100 00000000000000010011100010000000. The first two bits are tags used to inform the decoder of the size of the encoded number.

The reason for introducing VLN is that it enables efficient semi-fixed compression. It should be noted that VLN can be used to encode up to 32 bits, but it does not provide effective compression for values that are larger than two bytes. The lack of compression comes from the fact that there is not a tag for any nibble between four and eight nibbles. This means that for values larger than four nibbles the full 32-bit representation is used.

2.3.9 *VLNT*

The VLNT algorithm combines VLN with Tunstall. Furthermore, the algorithm works in the same manner as δ-T, but sends VLN instead of Elias-Delta when encoding the first occurrence of an input. The algorithm provides better embodiment of semifixed coding

the δ-T since the code generated is composed of variable length of nibbles rather than variable length of bits.

However, our version of VLNT does not group symbols together when sending VLN when encoding the first occurrence. Rather, VLNT sends an exception code to identify to the decoder that it is being encoded with VLN. The exception also takes the place of a header and identifies the appropriate encoding based on the required number of nibbles. Depending on the parsing scheme the number of exceptions needed changes to represent the different number of nibbles.

For example, four exception are needed when reading the entire 32-bits. An exception is needed to represent each of the one, two, four, and eight nibbles. However, only two exceptions are needed when parsing one byte at a time.

Although VLN does not provide effective compression of values larger than two bytes, VLNT's use of Tunstall is able to compress values larger than two bytes as long as the values exist in the code tree.

2.4 Search Engine Indexing

2.4.1 *Inverted Index Construction*

This section discusses inverted indexes and their construction.

The inverted index is a fundamental data structure in information retrieval (IR) [5]. It consists of multiple token and document ID pairs stored in a dictionary. The tokens are terms within a given document. For example, let document A be "Hello world" containing only the words "hello" and "world". The tokens in the dictionary containing this document only might include "hello" and "world". The document ID is a unique serial number given to a document. Now, assume that two documents exist: "hello world" and "hello sally". The documents' IDs are assigned as 1 and 2, and the token and document ID tokens in this

example dictionary might be: <"hello", [1, 2]>, <"world", [1]>, and <"sally", [2]>. The lists of document IDs for a given token are sometimes called posting lists [5].

The inverted index construction process includes five basic steps [5]:

1. Create a collection of documents.
2. Create a list of tokens or terms used in each document by parsing each document.
3. Preprocess the tokens to produce a list of tokens.
4. Index each document ID with each token term.
5. Sort the terms in lexicographic order.

2.4.2 *Gap Construction*

The gap construction is the process of attaining the differential change between the consecutive document IDs [6]. Since the IDs are unique and are sorted by the web-crawler we cannot get a value of 0. Furthermore, this process adds redundancy to a given posting list, where, generally, the gaps between indexes are smaller on average than the index themselves. Hence, the distribution of gaps "prefers" small integers. Another reason to add recurrence to a posting list is that compression algorithms, including the GVI, δ-T, and VLNT, work best with a source with a large number of repeated symbols.

For example, given the first 10 items of the posting list for the Wikipedia search term "state" are:< "state"; 12, 25, 303, 305, 307, 308, 309, 324, 330, 332> and the result after gap construction is: < "state"; 12, 13, 278, 2, 2, 1, 1, 15, 6, 2 >. The resulting gap index has multiples of the same number and smaller average index value. It can be noted that a term that occurs more often in searches is likely to induce small dispersion generally has a smaller average gap and introduces more redundancy.

2.5 Distributions

2.5.1 *Geometric Probability Distribution Function (GPDF)*

The geometric probability distribution function denotes the probability that the first success in a set of binary independent trials that can yield a "success" or "no-success value; where, each trial has the same probability of success, happens on the K trial. The function is given by $Probability\ (first\ success\ at\ K) = (1-p)^{k-1}p$, where K > 0 and p is the probability of success in an individual trial. In this thesis, we have experimented with GPDFs where p is a member of the set $\{0.5, 0.1, 0.01\}$. These distributions, specifically the case of $p = 0.01$, are considered relatively reliable models of the distribution of inverted indexes gap files.

2.5.2 *Poisson Distribution (PD)*

Poisson distribution is a discrete probability function. For a given interval of time or space the Poisson distribution describes the probability of the number of times that an event will occur in that interval of time or space under the constraint that these events occur with a known average rate and are independent of past intervals. The function is $given\ by\ P\ (k\ events\ in\ interval) = \frac{\lambda^k e^{-\lambda}}{k!}$, where λ is the average rate of occurrence of the event in an interval. In this thesis, we have experimented with Poisson distributions, where λ=128. This distribution is considered a relatively reliable model of the distribution of network traffic of integers obtained from sensors.

CHAPTER 3

LITERATURE REVIEW

This chapter examines the background research and highlights the differences between state-of-the-art research and this paper's research.

Dual tree encoding is an idea developed by G. H. Freeman, has argued that combining the Tunstall and Huffman algorithms improves the compression rate relative to using either one alone [15, 16]. Freeman uses Huffman coding to create a code tree and then extends the Huffman code tree with the Tunstall algorithm. Next, he creates the code dictionary with the leaves of the resulting tree, and finally encodes the dictionary with another Huffman code tree. Freeman's algorithm assumes a finite alphabet and static usage, whereas our paper assumes a boundless alphabet and online adaptive approach.

Stubley introduces two adaptive solutions for the dual tree codes process; one method uses an exhaustive search method to find the optimal dual tree code, while the other uses a state tree to select from a large look up table of possible dual tree codes [17]. Each of these methods limits the source dataset to a binary alphabet with a Bernoulli distribution. In contrast, the δ-T and VLNT used in this research are algorithms that process a number of different sources, including unbounded integer sources.

Fabris, Sgarro, and Pauletti's four adaptive algorithms use Tunstall, and their algorithm develops criteria defining when it is appropriate to rebuild the code tree [18]. Although these are valid and interesting ideas, the authors limit themselves to a known alphabet and limit the Tunstall tree sizes. Our research, however, works with an unknown alphabet and limitless tree sizes.

Baer's two methods improve the complexity of two Tunstall construction methods [19]. First, Baer presents a linear time for a binary Tunstall tree; second, a method is presented for non-Bernoulli and Markov sources. Both of these methods require the probabilities of the source alphabet to be known in advance. This study does not use this requirement.

Glory and Domnic's method for compressing an inverted index uses extended Golomb code [7]. However, their extended Golomb coding ends up to be equivalent to Elias-Gamma coding. Hence, the authors address integer compression, which is discussed in this thesis; however, they do not address the idea of adaptive or online compression achieved by the combination of Tunstall and Elias-Delta coding.

Anh and Moffat's development is an efficient and effective process for compressing D-gaps in a list of integers [6]. This compression method returns a binary and fixed-size representation of the list of integers. Although Anh and Moffat's method is efficient and effective, it does not address an adaptive process.

Meanwhile, Klein and Shapira define a process to improve the Tunstall process of variable-to-fixed length code encoding, and describe a process to compress text using a method termed DynC [20]. This method reduces the size of the code tree by cutting a suffix tree in order to create an optimal code tree. The research differs from this study because it compresses text rather than integers, and full knowledge of the source content is required before compressing.

In conclusion, the research presented in this thesis offers a number of unique alternatives to the prior research. First, unknown alphabets and sources are assumed. Second, the code tree is not limited with a minimum or maximum size before the encoding process starts. Third, δ-T and VLNT adapt to changes in the source, and finally, the research concentrates on compressing integers and unformatted data sources.

CHAPTER 4

EXPERIMENTAL SETUP

This chapter discusses the setup methodology and summarizes all of the experiments, the results of which are discussed in Chapters 5 and 6.

A total of 8 experiments are run. Experiments 1 through 4 use the excess tree growth method, while experiments 5 through 8 use the probability-limited tree growth method. All of the experiments use one of three source datasets: synthetic, Wikipedia gaps, or the Silesia benchmarks. Experiments 4 and 8 use both the synthetic and Wikipedia gaps sources.

4.1 *Summary of Input Sources*

This section discusses the input sources and their general characteristics.

4.1.1 *Synthetic Data*

The synthetic data is derived from two general probability distributions: GPDF and PD. Four different datasets make up the synthetic input data: Data GPDF 0.5, Data GPDF 0.1, Data GPDF 0.01, and Poisson distribution (PD). Each of the inputs is comprised of positive integers distributed with different probabilities. For the GPDF inputs, the probabilities are 0.5, 0.1, and 0.01. The PD input uses a $\lambda = 128$.

1. **"Data GPDF 0.5"** contains 10,000 integers. The minimum integer value is 1 and maximum integer value is 14. The number of unique integer values is 14. The dispersion is 0.0014.
2. **"Data GPDF 0.1"** contains 10,000 integers. The minimum integer value is 1 and maximum integer value is 99. The number of unique integer values is 73. The dispersion is 0.0073.

3. **Data GPDF 0.01"** contains 10,000 integers. The minimum integer value is 1 and maximum integer value is 826. The number of unique integer values is 531. The dispersion is 0.0531.
4. **"Poisson distribution"** contains 10,000 integers. The minimum integer value is 89 and maximum integer value is 147. The number of unique integer values is 58. The dispersion is 0.0058.

4.1.2 *Wikipedia Gap Sources*

The Wikipedia gap sources are taken from the sorted inverted indexes for several common search terms obtained from Wikipedia at the end of the year 2015. The search terms used are "state", "grei", "Facebook", "Wikipedia", "Trump", and "rousei".

1. "**State**" is the biggest dataset and contains 1,237,789 integers. The minimum integer value is 1 and maximum is 37,064. The number of unique integers is 991. The dispersion is 0.0008.
2. **"Grei**" contains 49,973 integers. The minimum integer value is 1 and maximum is 421,836. The number of unique integers is 5,737. The dispersion is 0.115.
3. "**Facebook**" contains 25,451 integers. The minimum integer value is 1 and maximum is 438, 117. The number of unique integers is 6,293. The dispersion is 0.25.
4. **"Wikipedia"** contains 12,820 integers. The minimum integer value is 1 and the maximum is 464,231. The number of unique integers is 6,438. The dispersion is 0.50.
5. **"Trump"** contains 4,589 integers. The minimum integer value is 1 and the maximum is 496,835. The number of unique integers is 3,857. The dispersion

is 0.84.

6. "**Rousei**" is the smallest dataset and contains 211 integers. The minimum integer value is 754 and maximum is 1,932,275. The number of unique integers is 211. The dispersion is 1.

4.1.3 *Silesia Sources*

The Silesia sources are a standard dataset used to compare compression algorithms. The dataset is made up of 12 different datasets comprised of many different formats of data [21]. Dispersion is not discussed because all datasets produce a ratio close to 0.00001.

1. **"dickens"** is comprised of the text of the works of Charles Dickens. The total number of bytes is 10,192,446. The minimum value is 9 and maximum is 129. The number of unique values is 100.
2. **"Mozilla"** is comprised of the tarred executables of Mozilla 1.0. The total number of bytes is 51,220,480. The minimum value is 0 and maximum value is 255. The number of unique values is 256.
3. **"mr"** is comprised of a collection of MRI images. The total number of bytes is 9,970,564. The minimum value is 0 and maximum value is 255. The number of unique values is 256.
4. **"nci"** is comprised of database of chemical structures. The number of bytes is 33,553,445. The minimum value is 10 and maximum value is 118. The number of unique values is 62.
5. **"oofice"** is comprised of the dll from Open Office 1.01. The number of bytes is 6,152,192. The minimum value is 0 and maximum value is 255. The number of unique values is 256.
6. **"osdb"** is comprised of the Open Source Database Benchmark (osdb) in the

MySQL format. The number of bytes is 10,085,684. The minimum value is 0 and maximum is 255. The number of unique values is 256.

7. **"raymont"** is comprised of the book *Chlopi* by Wladyslaw Raymont. The number of bytes is 6,627,202. The minimum value is 0 and maximum value is 255. The number of unique values is 256.
8. **"samba"** is comprised of source code of Samba 2-2.3. The number of bytes is 21,606,400. The minimum value is 0 and maximum value is 255. The number of unique values is 256.
9. **"sao"** is comprised of the SAO star catalog in the form of bin data. The number of bytes is 7,251,944. The minimum value is 0 and maximum value is 255. The number of unique values is 256.
10. **"webster"** is comprised of the 1913 unabridged dictionary text. The minimum value is 10 and maximum value is 126. The number of unique values is 126.
11. **"XML"** is comprised of a collection of XML files. The number of bytes is 5,345,280. The minimum value is 0 and maximum value is 252. The number of unique values is 104.
12. **"x-ray"** is comprised of a collection of x-ray pictures. The number of bytes is 8,474,240. The minimum value is 0 and maximum value is 255. The number of unique values is 256.

4.2 *Experiment Summaries*

This section summarizes each of the eight experiments.

The following experiments show that the new compression algorithms, δ-T and VLNT, compress the data in all but a few cases. Compression is greater for synthetic and

Wikipedia inverted indexes gap than Silesia data. δ-T and VLNT speed up throughput and reduce latency by compressing in one-pass.

Experiment 1:

Experiment 1 compares δ-T and VLNT by parsing the input data one byte at time using the excess tree growth method and the synthetic dataset. Entropy is used for the comparison of the obtained results with the theoretical limit.

VLNT requires the use of two exceptions to correctly encode VLN. The reason for two and not four is because parsing at the byte boundary only allows for a maximum value of 255 or a length of two nibbles. There needs to be exceptions to indicate lengths of two nibbles or one nibble.

Experiment 2:

Experiment 2 compares δ-T and VLNT while parsing the data one byte at time. The tree grows using the excess tree growth method, and the data sources are obtained from the Silesia dataset. Entropy is used for comparisons.

VLNT requires the use of two exceptions to correctly encode VLN. The reason for two and not four is because parsing at the byte boundary only allows for a maximum value of 255 or a length of two nibbles.

Experiment 3:

Like experiment 1, experiment 3 compares δ-T and VLNT using the full-tree code assignment method, but parses the data two bytes at time. The excess tree growth method is again used to grow the tree, and the synthetic dataset is used to make comparisons. The results are compared to entropy and to the results of GVAR encoding.

VLNT requires three exceptions when parsing two bytes at a time. This allows the decoder to know if the information following the exception is one, two, or four nibbles in length.

GVAR only needs one bit to represent the two tags that tell the decoder if an encoded number is one byte or two bytes long.

Experiment 4:

Experiment 4 compares δ-T and VLNT using the full-tree code assignment method, 4-byte parsing, and the excess tree growth method. Experiment 4 also uses the synthetic and Wikipedia gap datasets. As in the previous experiment, entropy and GVAR are used for comparisons.

VLNT requires four exceptions when parsing four bytes at a time. This allows the decoder to know if the information following the exception is one, two, four, or eight nibbles in length.

GVAR requires two bits to represent the four tags that tell the decoder if an encoded number is one, two, three, or four bytes long.

Experiment 5:

Experiment 5 uses the same dataset and parsing settings as experiment 1 to compare δ-T and VLNT, but uses the probability-limited tree growth method. Experimental results are compared to entropy.

VLNT requires the use of two exceptions to correctly encode VLN. The reason for two and not four is because parsing at the byte boundary only allows for a maximum value of 255 or a length of two nibbles.

Experiment 6:

Experiment 6 compares δ-T and VLNT using the probability-limited tree growth method, which also uses the 1-byte parsing method to parse the Silesia dataset. The results are compared to entropy.

VLNT requires the use of two exceptions to correctly encode VLN. The reason for two and not four is because parsing at the byte boundary only allows for a maximum value of 255 or a length of two nibbles.

Experiment 7:

Experiment 7 compares δ-T and VLNT parsing the synthetic dataset two bytes at a time. Like experiments 5 and 6, experiment 7 uses the probability-limited tree growth method. Comparisons are made with entropy and the GVAR.

VLNT requires three exceptions when parsing two bytes at a time. This allows the decoder to know if the information following the exception is one, two, or four nibbles in length.

GVAR requires two bits to represent the four tags that tell the decoder if an encoded number is one, two, three, or four bytes long.

Experiment 8:

Experiment 8 uses the synthetic and Wikipedia gap datasets to compare δ-T and VLNT. Like experiment 4, the datasets are parsed using the 4-byte method, but experiment 8 uses the probability-limited tree growth method. As in experiment 7, experiment 8 uses entropy and the GVAR to make overall comparisons.

VLNT requires four exceptions when parsing four bytes at a time. This allows the decoder to know if the information following the exception is one, two, four, or eight nibbles in length.

GVAR requires two bits to represent the four tags that tell the decoder if an encoded number is one, two, three, or four bytes long.

CHAPTER 5

EXPERIMENTAL RESULTS

This chapter discusses the experimental results of each of the eight experiments.

5.1 Experiment 1: Compressing synthetic data using excess tree growth (parsing one byte)

Figure 5 presents the results of compressing the synthetic dataset using δ-T and VLNT. The compression results are compared to the entropy for each of the data streams. The entropy number shown is calculated by reading each integer as a full 32-bit number. Each of the bit rates for the data streams are presented as per the 32-bit rate, which is done by multiplying the per bytes bit rate by 4.

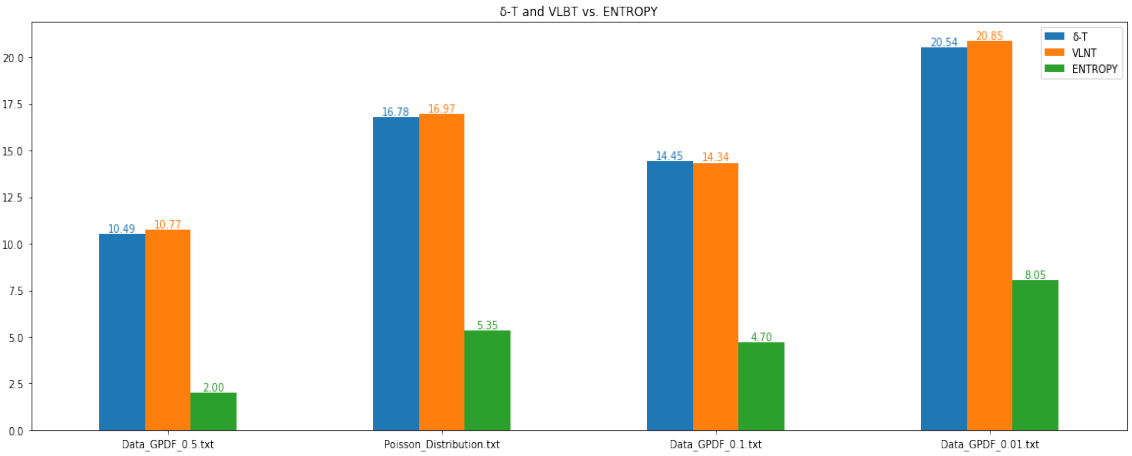

Figure 5: Synthetic results using excess tree growth and parsing with one byte

Two outcomes of compressing these datasets can be noted. First, the experiment's results underperform in relation to entropy. Second, where the number of independent symbols falls between two power of 2s affects whether δ-T or VLNT performs the worst.

The first observation on performance compared to entropy is the result of the parsing process, which adds symbols to all of the datasets. These additional symbols are not fully compressed out, leaving a compression dataset that is larger than the compression results from reading the entire 32-bit integer. For example, the number of encoded symbols sent when parsing one byte is 26,431 and 26,184 for δ-T and VLNT, respectively. Those numbers account for 2.6 times more symbols than in the original data left alone.

δ-T outperforms VLNT when the additional exceptions of VLNT force a deeper code tree. This is in part due to parsing, but the majority of the issues stem from where the number of independent symbols falls relative to a power of 2. For instance, the 15 independent values of Data GPDF 0.5 range from 0 to 14. The original dataset range is from 1 to 14 for a total of 14 independent numbers.

The additional symbol hinders the algorithm's compression rate in the case of Data GPDF 0.5. Assuming that δ-T is used, there is only one exception code. Hence, a total of 16, i.e. a power of 2, independent values for the results after the last unique symbol are encoded, whereby sequences are prevented.

However, compressing Data GPDF 0.5 using VLNT, the algorithm creates a deeper tree than δ-T. VLNT requires two exception codes for 1-byte parsing. The total number of independent codes equals 17 in this case, but because 17 is not a power of 2 the excess tree growth algorithm expands the rest of the tree, so that multiple combinations are now stored

in the dictionary. However, a lower bit rate is not always the result of additional sequences; additional combinations come at the cost of one more bit in the code sequence.

In conclusion, parsing a data source containing integers one byte at a time is counterproductive. The process creates additional symbols, which can lengthen the code. Ultimately, parsing causes the original data source to grow and results in additional work that the algorithm cannot compress fully to compensate. Finally, compression is achieved for every input in this experiment, since compression exists as long as the bit rate is below 32.

5.2 Experiment 2: Compressing Silesia data using excess tree growth (parsing one byte)

Figures 6 and 7 display the results ordered by entropy. Unlike experiment 1, the entropy numbers are created using the 1-byte parsing process. The reason for only considering the 1-byte parsing experiment with Silesia datasets is because it represents an unknown format.

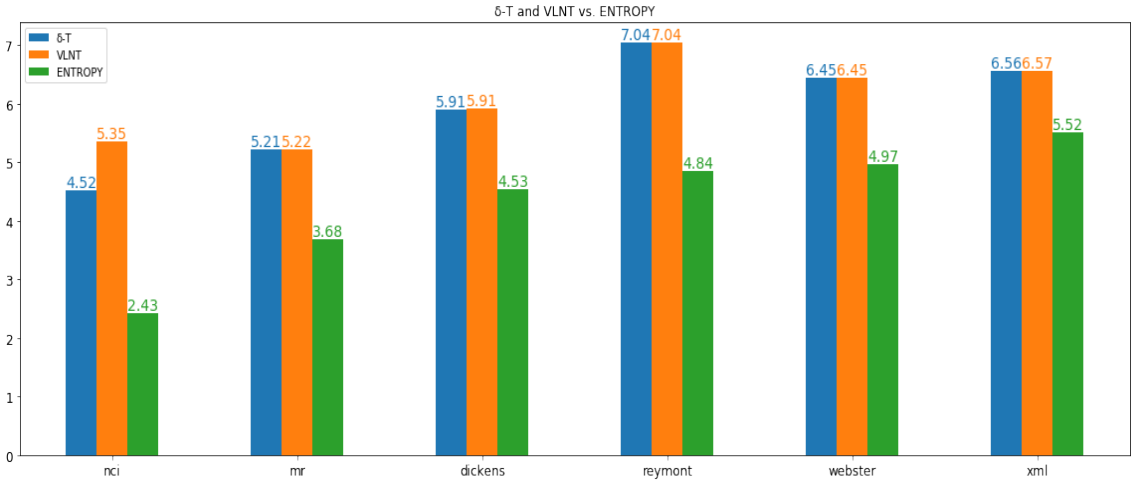

Figure 6: Silesia results using excess tree growth part A

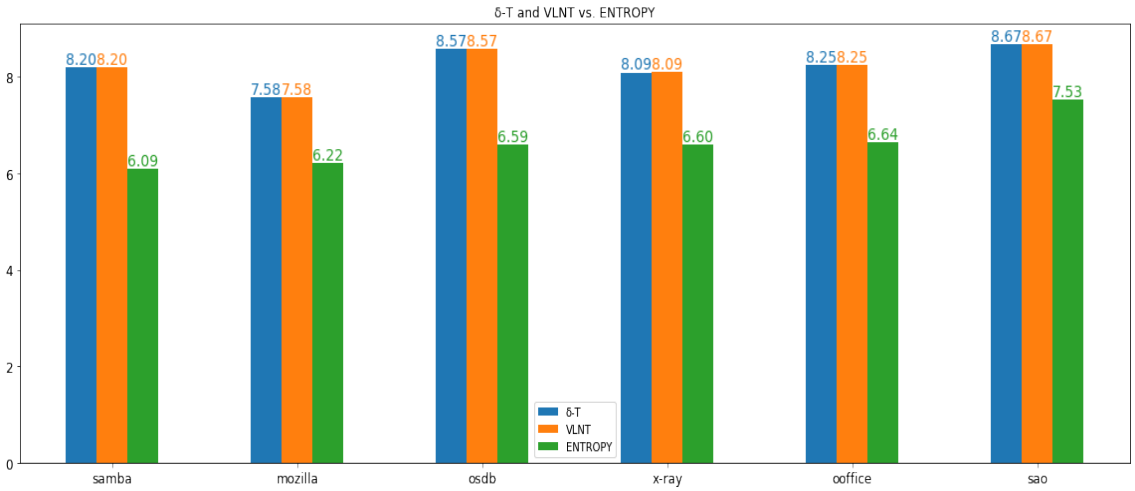

Figure 7: Silesia results using excess tree growth part B

Three observations can be gleaned from this experiment. First, the NCI data is the only dataset to show any significant difference between δ-T and VLNT. Second, the bit rates correlate with entropy. Third, datasets with fewer unique symbols do better than those with more.

The gap between δ-T and VLNT compressing the NCI data is caused by the fact that there are only 62 independent symbols in the dataset. A similar issue can be seen in the Data GPDF 0.5 data in experiment 1a. The one exception of δ-T leaves room for encoded sequences of symbols. Conversely, VLNT uses two exceptions, leaving the tree with 64 total codes; 64 being a power of two means that no room is left for sequences.

The graphs show that as entropy increases the bit rates increases for compression. In fact, the correlation between entropy and bit rate is .953. This results from the fact that the studied method performs better when the average symbol probability is higher. Higher probabilities mean better chances for sequences to exist in the input data.

NCI and dickens have a fewer number of unique symbols and tend to do better than those with more. This is because the fewer number of unique symbols with a relatively large

data stream increase the probability of sequences. However, mr has the same number of unique symbols as many of the datasets that perform considerably worse than mr. This is because mr is an MRI image and has more compressible sequences.

In conclusion, when dealing with large datasets there is little difference between the performance of δ-T and VLNT, although certain cases exist where δ-T outperforms VLNT. Entropy correlates well with bit rates when compressing the Silesia datasets. δ-T and VLNT's results rely on compressing sequences as seen in the case of mr, nci, and dickens. Finally, compression is only achieved on a select group of inputs where the bit rate is below 8.

5.3 Experiment 3: Compressing synthetic data using excess tree growth (parsing two bytes)

Figure 8 summarizes the results of experiment 3 using the synthetic datasets. Entropy and GVAR are used to compare the results. As in experiment 1, the entropy number is calculated using the full 32-bit integer dataset's probabilities, while GVAR numbers are calculated by compressing the datasets using the 2-byte parsing method. The resulting GVAR bit rates are multiplied by two to produce the per 32-bit rate.

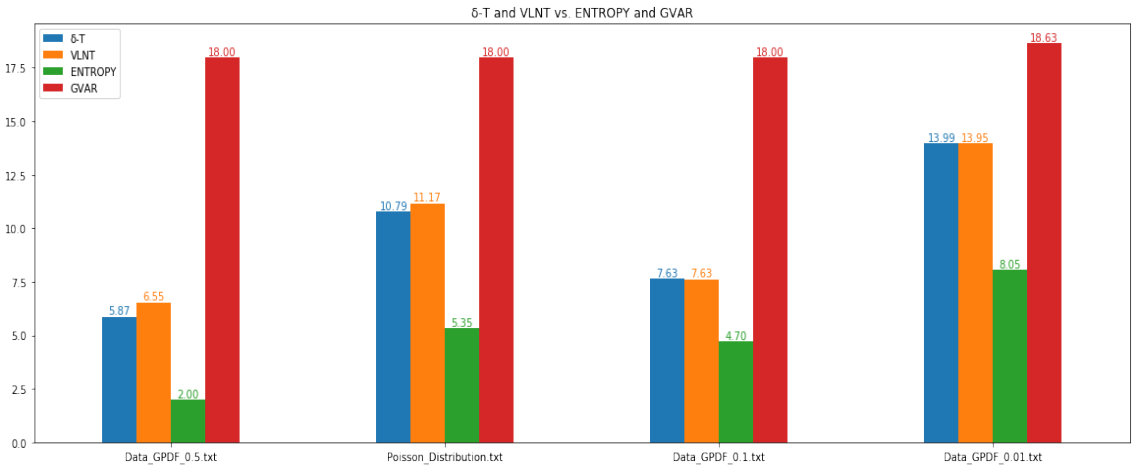

Figure 8: Synthetic results using excess tree growth and parsing with two bytes

Two observations can be drawn from these results. First, performance is poor when compared with entropy. Second, GVAR underperforms compared to δ-T and VLNT.

Performance in relation to entropy is same result as seen in experiment 1. Parsing increases the size of the data, and unless the additional symbols are fully compressed the bit rates underperform.

GVAR performance is linked to that of the datasets Data GPDF 0.5, Data GPDF 0.1, and Poisson distribution; no integer greater than 2^8 exists in those datasets. Adding an additional bit for the tag header produces an average of nine bits per symbol. Data GPDF 0.01's content of integers is larger than 2^8, and the larger integer values cause the slight increase in average bit rate.

In conclusion, the method of parsing integer datasets creates additional symbols that cannot be fully compressed. This additional work causes poor performance for VLNT, δ-T, and GVAR. Finally, compression is achieved on all the inputs in this experiment. Compression exist as long as the bit rate is less than 32.

5.4 Experiment 4: Compressing data using excess tree growth (parsing whole 32-bits)

Experiment 4 is divided into two parts: part A explains the compression of the synthetic datasets, and part B explains the compression of the Wikipedia gap datasets.

5.4.1 Experiment 4a: *Synthetic Datasets*

Figure 9 summarizes the results of experiment 4a. GVAR and entropy are used to compare the results, and are calculated by parsing using 4-byte boundaries. Similarly, the integers are parsed at the 4-byte boundary. Because all of the results represent bit rates per 32 bits, no multiplication is required.

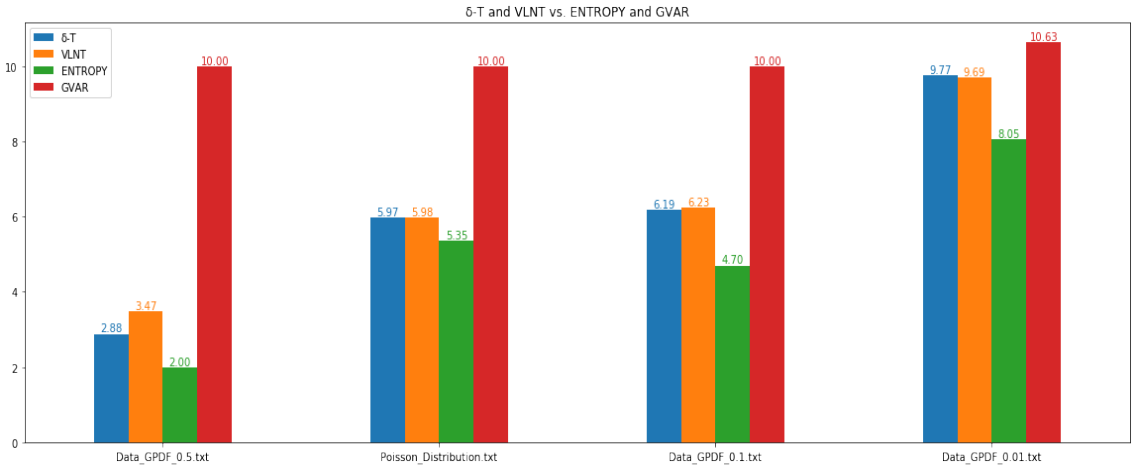

Figure 9: Synthetic results using excess tree growth and parsing with four bytes

Three observations result from this experiment. First, the bit rate gaps between entropy and both VLNT and δ-T are small. Second, the only significant difference between δ-T and VLNT arises when compressing Data GPDF 0.5. Third, GVAR greatly underperforms δ-T and VLNT.

The first observation is that the gaps between VLNT and δ-T and entropy are the lowest out of all the experiments in this study. The average bit rate difference from entropy in experiment 4a is 1.24, while the bit rate differences for 1a and 3a are 10.62 and 4.67, respectively. This difference results from the additional symbols added to the data in experiments 1a and 3a during the parsing phase.

The second observation is that the differences between δ-T and VLNT are highest when compressing Data GPDF 0.5. In experiments 1a and 3a, the number of exceptions required by δ-T and VLNT plays a role in increasing the bit rate. δ-T and VLNT require one and four exceptions. In this example, the number of unique symbols for Data GPDF 0.5 is 14, in contrast to experiments 1 and 2. Compressing using δ-T allows the final tree to have one sequence with one fewer bit in the code length than when using VLNT. Meanwhile, the

final tree's total number of sequences with VLNT is 14 sequences. However, VLNT uses an additional bit for the code.

The third and final observation is that GVAR does not compress files where the majority of values are small as well as δ-T and VLNT. GVAR's average bit rate is 10, meaning that the average value of the input is eight bits long. δ-T and VLNT are able to assign codes fewer than eight bits due to the nature of modified Tunstall code tree.

In conclusion, VLNT and δ-T perform better without parsing the data. Increasing the number of exceptions can hinder performance. GVAR is hindered by its use of fixed length encoding. A greater degree of compression exist in this experiment as the bit rates are much lower than the previous experiments.

5.4.2 Experiment 4b: *Wikipedia Gap Datasets*

Figure 10 summarizes the results of compressing Wikipedia gap datasets in experiment 4b. GVAR and entropy are used for comparisons, and datasets are read using the whole 32-bit integer. The results all represent bit rates per 32-bit integer.

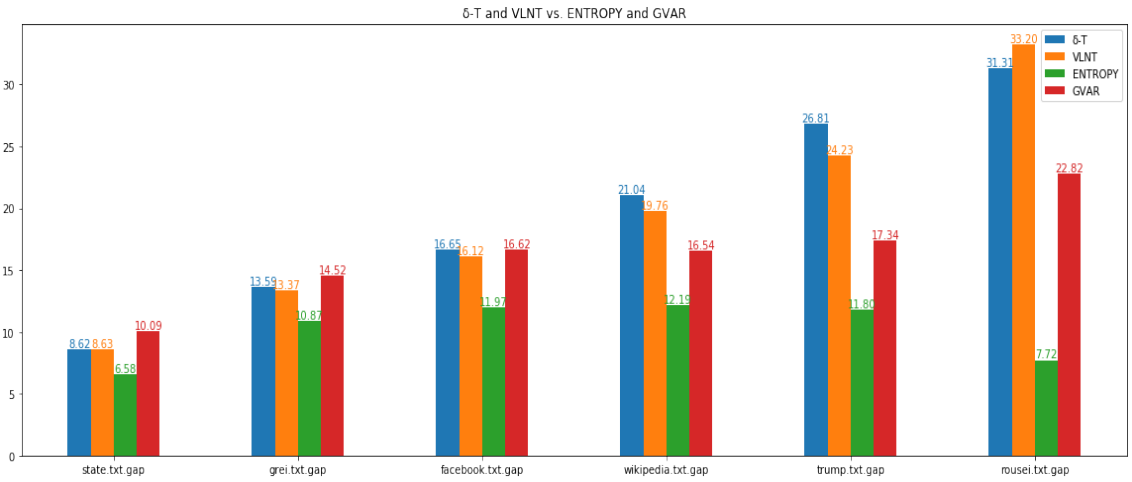

Figure 10: Wikipedia results using excess tree growth and parsing with four bytes

Three observations can be discussed for experiment 4b. First, entropy is not correlated with bit rate. Second, the performance when compressing rousei is troubling, and third, VLNT results in a smaller bit rate than δ-T for the majority of the datasets.

The correlation between entropy and bit rate is .09. Alternatively, the correlation between dispersion and bit rate is .98. Removing rousei's results increases the correlation with entropy and thus the bit rate to .78.

Figure 11 shows the normalized values for the dispersion, entropy, and average bit rates between δ-T and VLNT for each of the Wikipedia gap data. Visually it is clear that bit rate and entropy are poorly correlated.

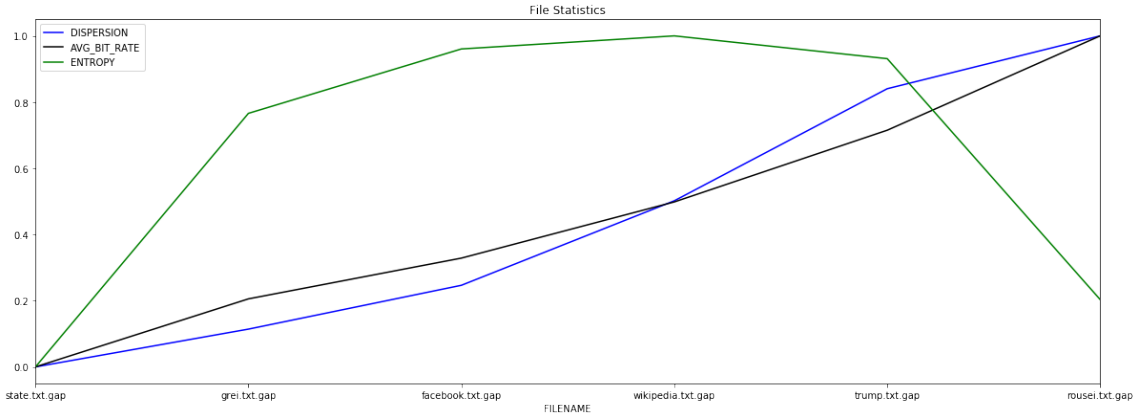

Figure 11: File statistics for the Wikipedia gap dataset

The troubling performance of rousei for both of the compression algorithms is rooted in the dispersion of the data, which is exactly 1. Consequently, no repeated numbers exist for rousei and every number is coded with either VLN or Elias-Delta.

The reason that δ-T has a lower bit rate than VLNT is the larger value sizes of independent symbols in rousei, whereby the average independent symbol value is 232,295, which falls in the 3-byte range. Because VLNT compression stops at two bytes, everything

larger is not compressed. In fact, the encoding actually inflates the data by adding an exception flag to the front of the code.

The most surprising observation is the performance of VLNT over δ-T. A unique side-by-side comparison is made from compressing the trump data. The same number of compressed codes is used to compress trump for both VLNT and δ-T, and the final code length is the same for trump. The trump data compresses in the same order regardless of whether VLNT or δ-T is used. Three sequences are found and represent the same symbol combinations, which implies that the difference between the two compression algorithms results from exceptions.

3,857 exceptions are used in both datasets. The number of bits for exceptions in VLNT and δ-T are 103,346 and 115,162, respectively; these values have a difference of 11,816. Alternatively, the number of bits for known symbols in VLNT and δ-T are 7,824 and 7,822, respectively – a difference of only two bits.

Compressing the trump data source results in 3,375 cases of δ-T exceptions being greater than VLNT exceptions. The number of VLNT exceptions greater than δ-T is 217, while the percentage of cases where δ-T exceptions are greater than VLNT exceptions is .87. In the case of the state data source, that percentage drops to .37.

VLNT produces shorter exceptions over δ-T for certain numbers. For example, when compressing trump, the exception for 12 using VLNT is 111100. The first two bits are used for the exception code, and the last four bits are a bit representation of 12. Using δ-T, the full exception code is 100100101, where the first bit is used for the exception code and the last eight bits for the Elias-Delta code 13.

In conclusion, a number of new observations arise from using this larger dataset. Dispersion is a better correlated with bit rate than entropy, and in the case of the rousei data, δ-T and VLNT perform worse only using exceptions. The bit rate and dispersion correlation for the Wikipedia datasets results from the fact that datasets such as rousei and trump are relatively small but have a larger ratio between unique symbols and size. Entropy does not take the data stream size into account. Finally, compression exists in all but one input. However, the one input with no compression is rousei and as described above is a special case.

5.5 Experiment 5: Compressing synthetic data using probability-limited tree growth (parsing one byte)

Figure 12 summarizes the results of experiment 5 using the synthetic datasets, with entropy used for comparison. The bit rates for δ-T and VLNT are multiplied by four to obtain the per 32-bit representation. The graphs represent a single dataset, with the x-axis displaying each of the probability limits and the y-axis displaying the bit rates.

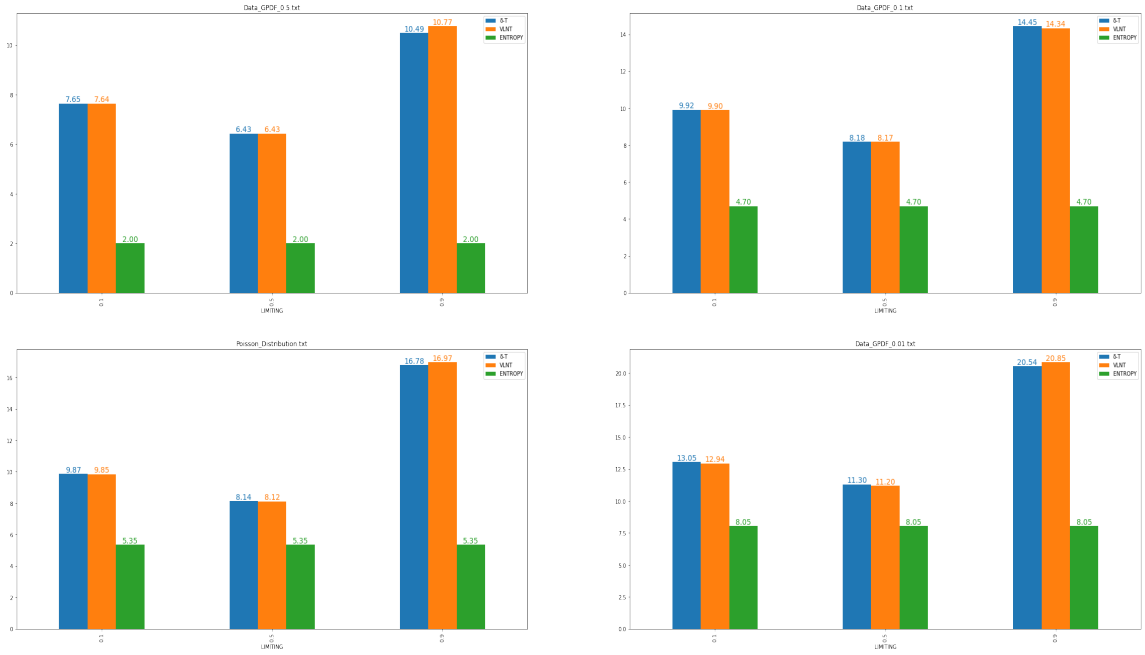

Figure 12: Synthetic results using probability-limited tree growth and parsing with one byte

Three observations can be collected from experiment 5a. First, the performance of δ-T and VLNT is erratic. Second, a probability of 0.5 produces the best results across all datasets. Finally, the performance of δ-T and VLNT is poor relative to entropy.

The performance of δ-T and VLNT changes with data and probability limits. For example, in the case of the Data GPDF 0.01, Data GPDF 0.5, and Poisson distribution datasets, δ-T and VLNT bit rates are similar unless using a .90 limiting probability, which results in a lower bit rate for δ-T than it does for VLNT. Compressing Data GPDF 0.1 results in the opposite using a limiting probability of .90.

Each of the datasets compressed using the .90 limiting probability produces the following percentages of symbol sequence codes to total codes for Data GPDF 0.5, Data GPDF 0.1, Data GPDF 0.01, and Poisson distribution: .52, .86, .65, and .42, respectively, for VLNT. For δ-T the corresponding percentages are .52, .84, .68, and .43. The percentage of

δ-T exceptions being greater than VLNT exceptions for Data GPDF 0.5, Data GPDF 0.1, Data GPDF 0.01, and Poisson distribution are .80, .92, .99, and .98, respectively.

This difference in compressing Data GPDF 0.1 using the .90 limiting probability results from the fact that δ-T has a lower use of symbols representing sequences of symbols than VLNT. 86% of the codes used by VLNT represent symbol sequences, while the codes used by δ-T represent symbol sequences in only 84% of cases. This decrease in symbol sequences cannot offset the fact that .92 of the exceptions for δ-T are larger than VLNT exceptions.

The experiment using a limiting probability of .50 produces the best results. In order to explain this phenomenon, an examination of the same statistics as above is performed on the .50 and .10 experiments. In addition, the final tree size is also examined.

Using a limiting probability of .50, the data statistics are as follows. The percentage of symbol sequence codes to total encoded codes for Data GPDF 0.5, Data GPDF 0.1, Data GPDF 0.01, and Poisson distribution are .899, .975, .901, and .987, respectively, for VLNT. For δ-T, the corresponding percentages are .891, .977, .902, and .987. The exceptions of δ-T are greater than those of VLNT for Data GPDF 0.5, Data GPDF 0.1, Data GPDF 0.01, and Poisson distribution in .80, .95, .99, and .98 percent of cases, respectively.

Using a limiting probability of .10, the examination of the results is as follows. The percentage of symbol sequence codes to total encoded codes for Data GPDF 0.5, Data GPDF 0.1, Data GPDF 0.01, and Poisson distribution are .99, .99, .908, and .99, respectively, for VLNT. For δ-T the corresponding percentages are .99, .99, .908, and .99. The exceptions of δ-T are greater than those of VLNT for Data GPDF 0.5, Data GPDF

0.1, Data GPDF 0.01, and Poisson distribution in .80, .946, .99, and .98 percent of cases, respectively.

The experiment using a limiting probability of .10 takes advantage of more sequences than any of the other experiments. However, this comes at the cost of a larger tree and thus a longer bit string code. This longer code offsets any gains from using more symbol sequences. Meanwhile, using the .90 limiting factor utilizes a smaller tree but fewer symbol sequences. The average final tree size for each of the limiting factors is 10, 8, and 6.5 for probabilities .10, .50, and .90, respectively.

An observation that stands out is the performance of δ-T and VLNT relative to entropy, the cause of which is discussed in experiment 1a. Parsing the data on the 1-byte boundary increases the total number of symbols in need of encoding from 10,000 to 40,000. This increase is not fully compressed and thus has a higher bit rate.

In conclusion, a number of observations result from experiment 5a. The best choice of probability limit takes tree size and the number of symbol sequences into account. Finally, parsing on the 1-byte boundary produces poor results relative to entropy. Finally, compression exists for each input because no bit rate is larger than 32.

5.6 Experiment 6: Compressing Silesia data using probability-limited tree growth (parsing one byte)

The probability-limited tree growth method is used to compare δ-T and VLNT in this experiment, which also uses the 1-byte parsing method to parse the Silesia dataset. The results are compared to entropy.

Figures 13 and 14 illustrate the results of experiment 6. Because the input dataset is of an unknown format, no manipulation is needed to represent the per 32-bit results. The results are compared to entropy.

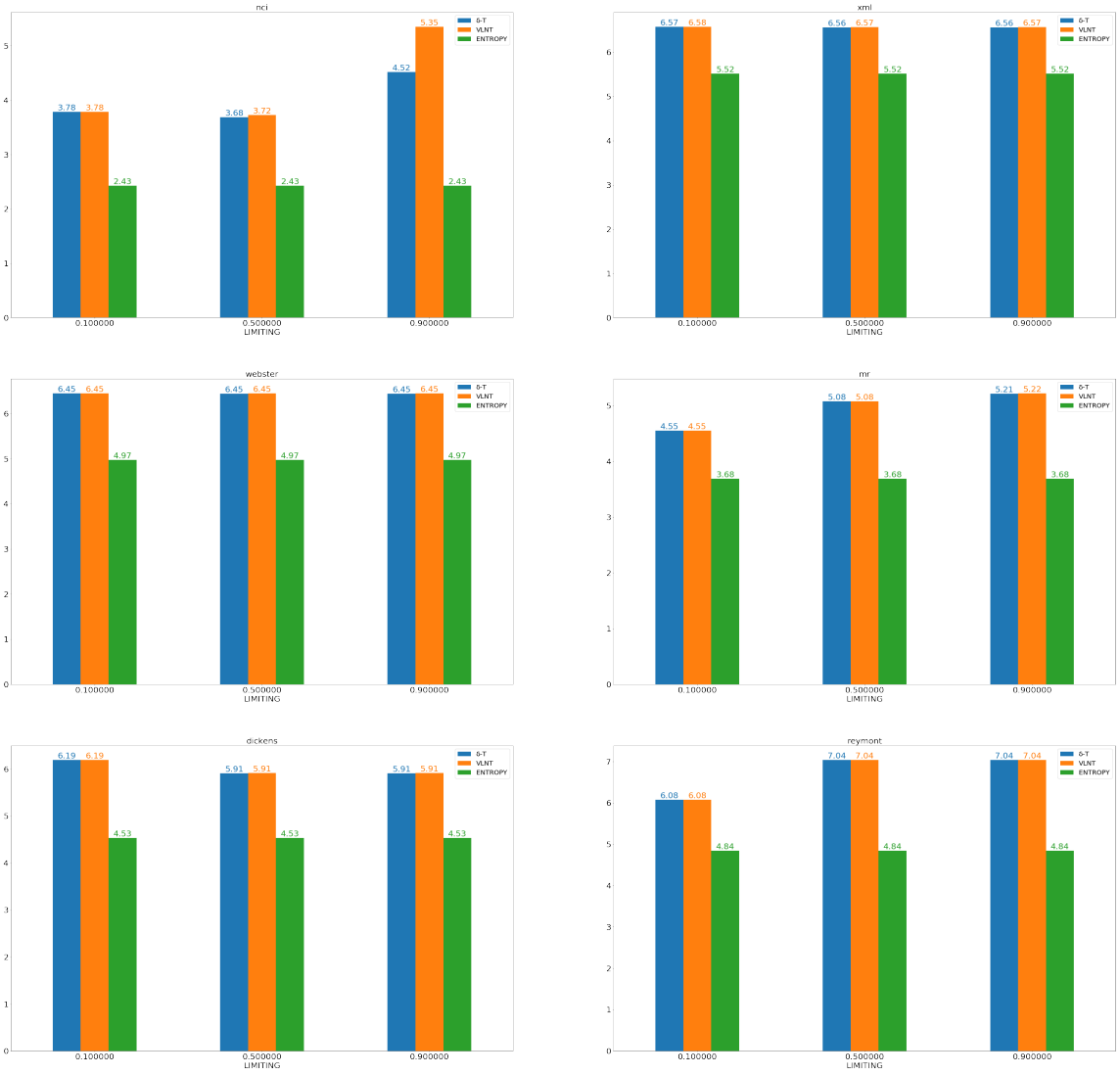

Figure 13: Silesia using the probability-limited tree growth method part A

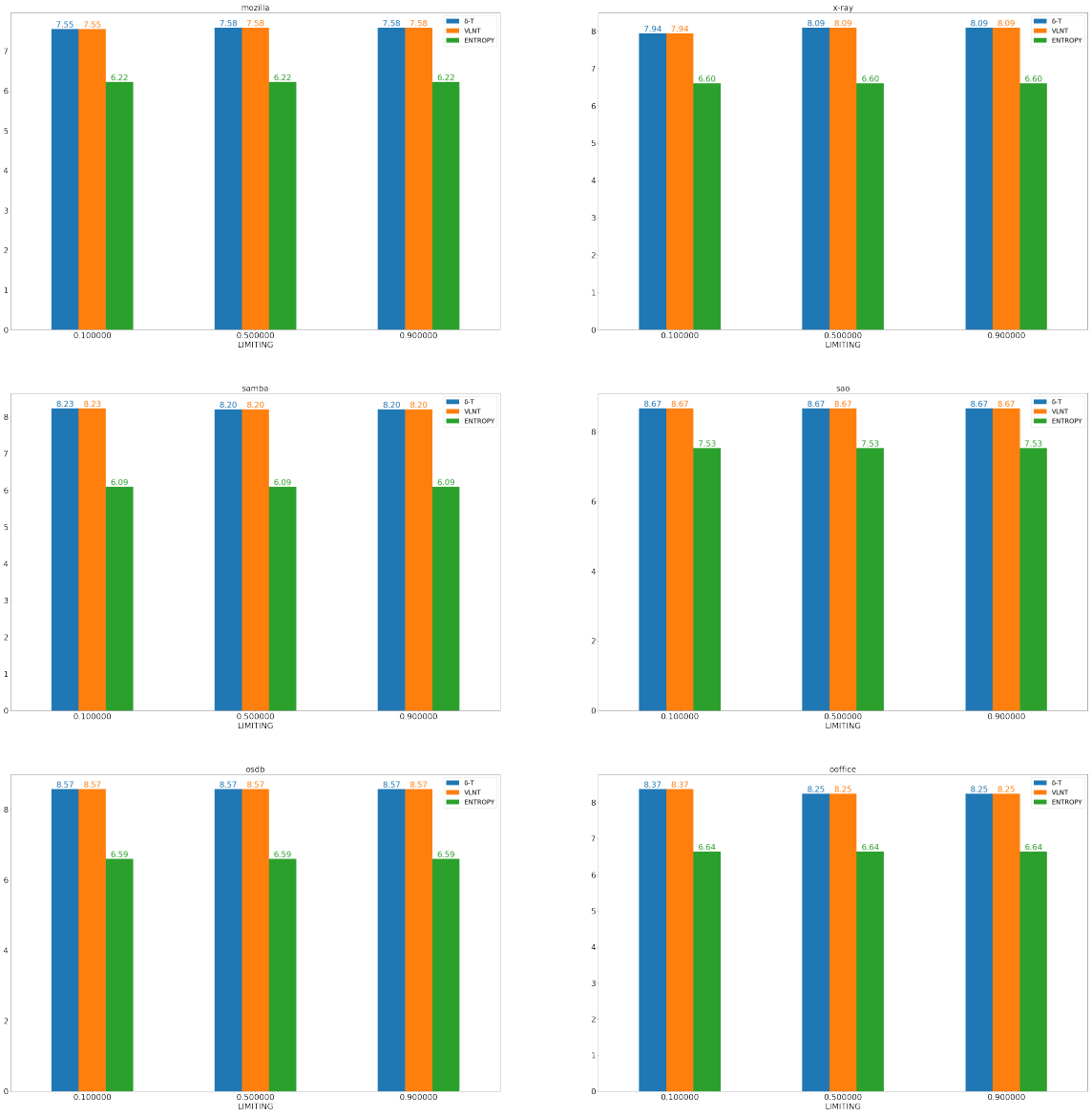

Figure 14: Silesia using the probability-limited tree growth method part B

The results indicate several patterns. First, the difference between δ-T and VLNT is minimal. Second, for the majority of the datasets there is no evident benefit of changing the limiting probability.

In experiment 6 there is only one instance of a significant difference between δ-T and VLNT. This difference exists only for the NCI dataset, because of the number of unique symbols in the data; it contains only 62 unique symbols. This allows δ-T one code to

represent a sequential set of symbols. The number of unique symbols in the other datasets is either equal to a power of 2, like 256, or between powers of 2, in such a way that it makes little difference which algorithm is used.

The experiments involving Samba, XML, and osdb datasets show little, or in the case of osdb no, difference between bit rates using the different limiting probabilities. Alternatively, the NCI dataset results show greater differences using the different limiting probabilities.

These differences, or lack thereof, stem from the probabilities of the symbols in the datasets. The highest probability for a single symbol for Samba, XML, and osdb is .10, .08, and .06, respectively. Meanwhile, NCI contains symbols with probabilities as high as .52. These higher probabilities take advantage of the limiting factors in this algorithm.

In conclusion, several observations can be made from this experiment. Unless the number of unique symbols falls in the perfect spot in relation to a power of 2, the differences between δ-T and VLNT are small. In order to benefit from using the limiting probability threshold, the thresholds need to be low enough to capture the probabilities of the symbols contained in the data. Entropy is a good number to use to gauge the resulting bit rates for a set of data. Finally, compression exists only for a select number of inputs where the bit rates are less than eight.

5.7 Experiment 7: Compressing synthetic data using probability-limited tree growth (parsing two byte)

Figure 15 summarizes the results of experiment 7 using the synthetic datasets, and comparisons are made with entropy. The bit rates for δ-T, VLNT, and GVAR have been multiplied by two to obtain the per 32-bit representation. The graphs represent a single

dataset, with the x-axis displaying each of the probability limits and the y-axis displaying the bit rates.

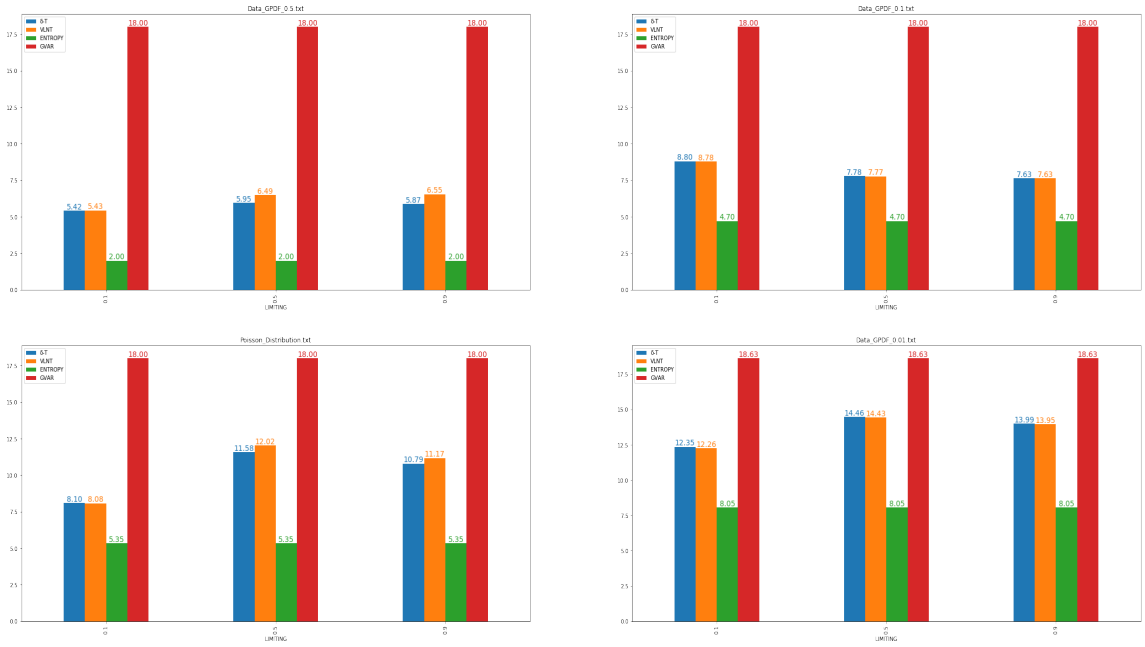

Figure 15: Synthetic results using probability-limited tree growth and parsing with two bytes

Experiment 7 provides the following observations. δ-T performs better in some cases, and Data GPDF 0.1 performs best using the highest limiting probability. Parsing at the 2-byte boundary produces poor performance relative to entropy, but a better performance than GVAR.

δ-T performs significantly better compressing Poisson distribution and Data GPDF 0.5 using the limiting probabilities .50 and .90. The reason for this is related to the exception and the fact that no probability is greater than .50 in either of these datasets. The tree is not grown due to probability. Therefore, the only way that the tree includes symbol sequences is due to excess space. δ-T uses less excess space on exceptions, leaving more room for symbol sequences.

The compression of Data GPDF_0.1 benefits from the smaller tree grown using a higher threshold. The bit string lengths for the final tree are the same for the thresholds .50 and .90. However, the bit string lengths used at the beginning of the compression process are longer for .50 than for .90.

Relative to entropy, parsing on the 2-byte boundary provides no benefit in compression. However, δ-T and VLNT perform better than GVAR. The poor performance is a result of the increase in symbols that are not fully compressed.

In conclusion, a number of observations can be made from experiment 7a. First, δ-T performs better than VLNT when exceptions cut out a majority of the symbol sequences. Second, larger tree sizes are counterproductive if increases in bit string lengths cannot be offset by the use of symbol sequences. Furthermore, 2-byte parsing is ineffective in compressing integers. Finally, compression exists for each of the inputs because all bit rates are less than 32.

5.8 Experiment 8: Compressing data using probability-limited tree growth (parsing whole 32-bits)

The final experiment uses the synthetic and Wikipedia gap datasets to compare δ-T and VLNT. As in experiment 4, the datasets are parsed using the 4-byte method. In contrast, experiment 8 uses the probability-limited tree growth method. As in experiment 7, experiment 8 uses entropy and GVAR to make overall comparisons.

5.8.1 Experiment 8a: *Synthetic Datasets*

Figure 16 presents the compression results for experiment 8 using the synthetic datasets, where GVAR and entropy are used to compare the bit rates for each of the experimental setups. No multiplication is required because experiment 8 parses the data using the entire 32-bit integers.

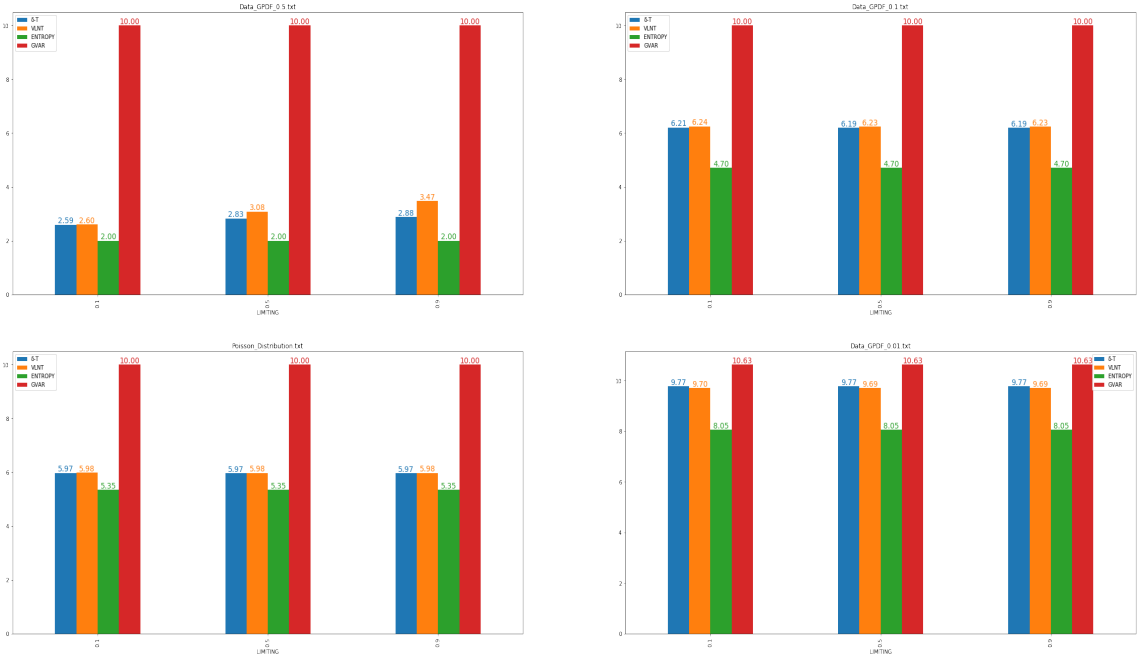

Figure 16: Synthetic results using probability-limited tree growth and parsing with four bytes

Three observations are made on experiment 8a. First, there is little difference in bit rates between the different thresholds. Second, δ-T performs better than VLNT with any significance at compressing Data GPDF 0.5. Third, GVAR underperforms relative to δ-T and VLNT unless compressing Data GPDF 0.01.

The bit rates for each of the datasets are altered very little by changes to the limiting probability threshold, due to the probabilities of the symbols in the datasets themselves. The highest symbol probability for each of the datasets is .501, .098, .011, and .058 for Data GPDF 0.5, Data GPDF 0.1, Data GPDF 0.01, and Poisson distribution, respectively. Data GPDF 0.5 is the only data that shows a significant difference, because it can utilize the threshold and grow an additional level of the tree.

δ-T performs better at compressing Data GPDF 0.5 for the same reason discussed in experiment 1a. δ-T takes advantage of only having to use one exception and is thus able to produce more symbol sequences. For example, when compressing Data GPDF 0.5 using

a limiting probability of 0.5, δ-T's final tree contains three codes representing three symbols in sequence. In contrast, VLNT reserves those codes for exceptions.

The superior compression of δ-T and VLNT over GVAR comes from their ability to represent all the possible symbols with less than the ten bits which is the minimum required by GVAR. For example, all the symbols in Data GPDF 0.5 are represented with four bits. Additionally, GVAR is forced to use ten bits to represent every symbol in the Data GPDF 0.5 because compressing values less than one byte is impossible with less than ten bits using GVAR.

In conclusion, three observations are made in this experiment. First, datasets with higher probability distributions of symbols take advantage of the probability threshold algorithm. Second, by using this method of tree growth, more than two symbols are represented using δ-T. Finally, the algorithms perform relatively well at compressing the synthetic datasets compared to GVAR. Finally, compression exists in all inputs for this experiment.

5.8.2 Experiment 8b: *Wikipedia Gap Datasets*

Figure 17 summarizes the compression results for experiment 8 using the Wikipedia gap datasets, where GVAR and entropy are used to compare the bit rates for each of the experimental setups. No multiplication is required because experiment 8 parses using the entire 32-bit integers.

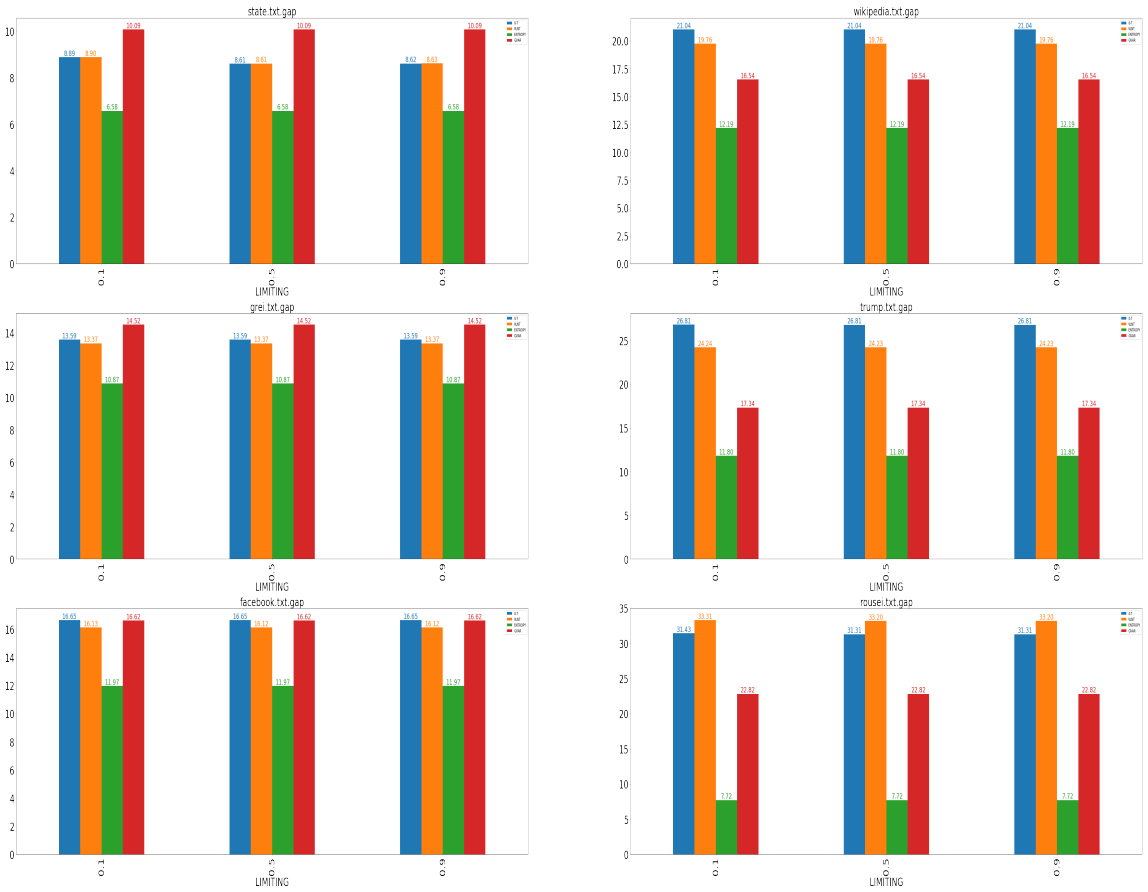

Figure 17: Synthetic results using probability-limited tree growth and parsing with four bytes

The major observations resulting from this experiment are as follows. First, no significant changes exist in bit rates from one limiting probability to another. Second, δ-T and VLNT perform worse than GVAR for a number of the datasets. Finally, bit rates do not correlate with entropy.

Other than state, no data exhibits more than a .10 change in bit rate between the different limiting probabilities, for a similar reason to that discussed in experiment 8a. The datasets in this set do not contain a symbol with a sufficiently high probability distribution to make use of the threshold algorithm.

Considering the results relative to GVAR, there is a surprising observation. For datasets such as trump and Wikipedia, δ-T and VLNT underperform in relation to GVAR.

This is because the symbols are distributed in a manner that prevents the use of sequential symbols that are encoded in the tree.

The correlation of entropy and bit rates is .08 in this experiment. However, dispersion correlates with bit rates at .98, but entropy fails to effectively relate how the symbols are dispersed within the data source.

Figure 18 shows the normalized values for dispersion, entropy, and the average bit rates for both δ-T and VLNT for each of the datasets. It is visually apparent that entropy does not correlate with bit rate and that dispersion correlates better for these types of files.

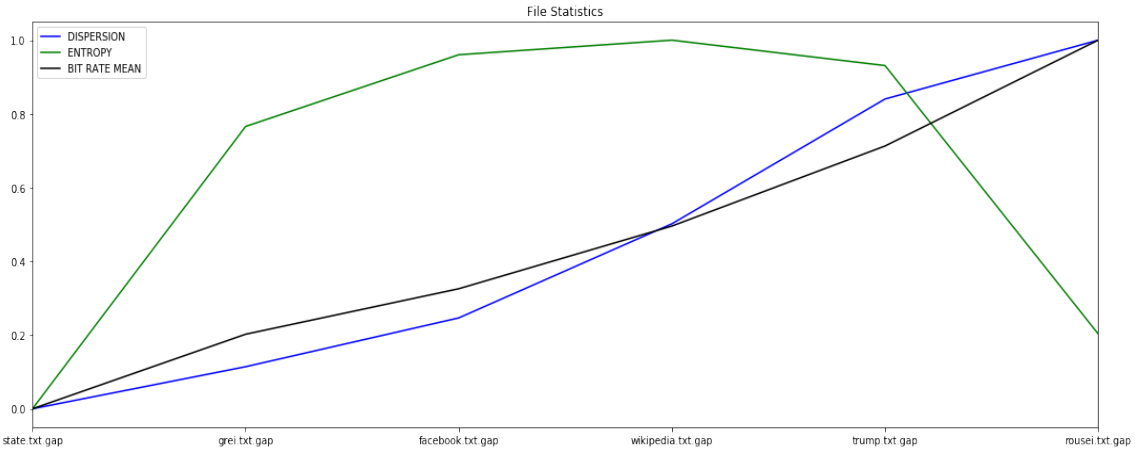

Figure 18: Statistics for Wikipedia gap data

In conclusion, a number of issues arise in this experiment. First, the symbol distribution in this dataset fails to take advantage of the tree growth algorithm. Second, if the algorithm cannot use codes that represent multiple symbols, then GVAR exhibits better performance. Finally, dispersion is a suitable metric to use when entropy is no longer useful. Finally, compression exists for all inputs except rousei for this experiment.

CHAPTER 6

RESULTS EVALUATION

This chapter evaluates the results and discusses our impressions of the compression algorithms.

In this thesis research, we have conducted eight experiments in an effort to improve compression performance. Experiments 1, 3, 5, and 7 parse the datasets on different byte boundaries in an attempt to increase the sequences of symbols. Experiments 1 to 4 use an excess tree growth method, while experiments 5 to 8 use a probability-limited tree growth method. Significantly, the experiments reveal that the compression results obtained by δ-T and VLNT vary depending on the input datasets.

For the integer datasets, the new compression algorithms show that they are effective in compressing in all but one special case. For the Silesia dataset, the algorithms are not as effective. However, the nature of the Silesia dataset shows that the δ-T and VLNT are able to compress without knowing anything about the dataset before starting the compression process.

Parsing integer datasets on different byte boundaries hinders the performance of the compression; parsing the datasets adds symbols that the algorithm is unable to fully compress. This is evident in the decreasing bit rates when progressing from parsing on the 1-byte boundary to the 4-byte boundary.

The excess tree growth method used in experiments 1 through 4 demonstrates that the number of exceptions plays a key role in compression quality. The one exception of δ-T provides a benefit over VLNT's multiple exception scheme. Benefits in smaller exception

schemes can be seen in all experiments when compressing the Data GPDF 0.5 dataset. The number of independent symbols in this dataset falls by around a power of two. In experiments 1a and 3a, the number of independent symbols is 15, and in experiment 4a the number is 14. The use of one exception flag by δ-T allows it to represent all symbols without growing the tree to fit additional exceptions. The compression is improved in experiment 4a by the additional space for a sequence. In experiment 2, the same effect is seen with the compression of the 62 independent symbols in NCI.

In experiments 5 through 8, the probability-limited tree growth method is used to grow the tree. In these experiments, an equilibrium point emerges whereby the tree is allowed to grow to the point where the additional code size does not limit performance. This can be best seen in experiment 5a, where a probability limit of .50 was the most effective.

However, finding the correct balance between these sets of probabilities is important. Trying to compress a dataset with symbol probabilities which never meet the threshold provides no benefit. An example of this phenomenon is seen in the results of experiment 4b, which is almost identical to the results of experiment 8b. Additionally, compressing the larger Silesia datasets in experiment 6 results in a similar outcome. All but two datasets show no difference between the three probability thresholds.

Furthermore, each of these experiments uses both of the algorithms. Of particular interest are those experiments that create the largest gaps between compression rates. The previous differences in exceptions explain the results of the NCI and Data GPDF 0.5 datasets. However, symbol values also exhibit an interesting effect on performance; δ-T and VLNT show very few differences when compressing datasets containing small symbol values. For example, experiments 1, 2, 3, 5, 6, and 7 all limit the value to below a 2-byte number, and their compression results are similar. Interestingly, experiments 4b and 8b show

a growing difference between δ-T and VLNT. This difference exists because VLNT better parses larger integer values. However, the rousei data demonstrates that the VLN aspect of VLNT fails to efficiently compress integers larger than two bits.

These results reveal no clear winner in terms of performance. Performance is largely based on the situation of both input and algorithm settings. However, δ-T using the excess tree growth algorithm does perform better overall.

δ-T and VLNT are both satisfactory algorithms for compression. Compression is seen in all the experiments with datasets involving integers. Compression is even seen in the experiments involving the Silesia dataset. Although, VLNT is better at compressing numbers in the 2-byte range. However, the difference is small enough to counter that the single exception flag and being bounded by integer values in δ-T are better traits.

δ-T and VLNT aside, both excess and probability-limited tree growth methods are good growth algorithms. The probability-limited method is better at compressing low entropy datasets than excess tree growth; for example, the results produced for Data GPDF 0.5 by experiment 8a are the best results of the entire study. Nonetheless, probability-limited tree growth method provides no benefit in real world cases, as seen with the gap datasets. Because of the former's lack of benefits and additional complexity, excess tree growth is the better option.

This study concludes that δ-T using excess tree growth is the better algorithm. Nevertheless, the probability-limited tree growth method may provide a platform for future study. Two questions are particularly interesting to consider: can the probability threshold be dynamically determined, and can trimming the tree be beneficial?

The advantage of dynamically allocating a tree growth threshold is that it provides the benefits of adding additional sequences, but for all datasets. This extends the benefits seen in experiment 8a for the Data GPDF 0.5 data to all datasets, regardless of their symbol probabilities.

The tree trimming idea comes from a paper by Klein and Shapira [20]. The use of DynC could prove to be effective at preventing the probability-limited tree growth method from adding too many additional bits to the code.

This is a highly informative study with some interesting ideas. At times these algorithms outperform industry standard practices, such as GVAR. With additional study these algorithms could provide a solution to the growing need for more efficient data storage.

An additional study exploring the probability-limited tree growth approach could lead to greater compression of integers with the benefit of increased throughput and reduced latency.

CHAPTER 7

CONCLUSION AND FUTURE RESEARCH

This thesis presents four new lossless compression algorithms that achieve significant compression ratio across several different input data sources. The algorithms are focused on compression rather than computation complexity. We devised two new algorithms δ-T and VLNT and two new modifications to the Tunstall code tree growth method. By combining the two areas, we create four new methods for compression. Combining Elias-Delta and VLN with Tunstall allows for an adaptive single-pass method for

compressing unbounded data. The adaptive single-pass nature of these algorithms improves throughput and latency.

To the best of our knowledge, this is the first attempt to combine Elias-Delta and VLN with Tunstall. Furthermore, to our knowledge, our variations on Tunstall's code tree growth method are innovative. To test the algorithms' ability to compress unbounded integers, a number of different data types are used. These datasets include synthetic data with both GPDF and PD distributions, Wikipedia search indexes for popular search terms, and the Silesia compression benchmark dataset. The research indicates that our compression algorithms perform well with a number of different inputs but compression of integers is superior. Especially, compressing restricted probability integers.

Further research is planned to explore different modifications that might yield better results. Modifications to the probability-limited tree growth method could yield a better result than the ones presented in this paper. We found that the set limits did not effectively grow the tree in some cases. If those fixed limits were allowed to be dynamically chosen then there could be increased compression. Dispersion is a possible metric to guide the dynamically chosen threshold. Additionally, too low of a threshold yields a large tree that reduces compression. However, if DynC were used to trim the tree as discussed in Klein and Shapira, then additional compression could be possible [20]

REFERENCES

[1] J. Ball, *NSA collects millions of text messages daily in 'untargeted' global sweep,* New York: The Guardian, 2014.

[2] M. d. Kunder, "worldwidewebsize.com/," Faculty of Arts of Tilburg University, 28 June 2017. [Online]. Available: http://www.worldwidewebsize.com/.

[3] G. Press, "Top 10 Tech Predictions for 2017 from IDC," 1 November 2016. [Online]. Available: https://www.forbes.com/sites/gilpress/2016/11/01/top-10-tech-predictions-for-2017-from-idc/.

[4] M. Elgan, "Artificial intelligence needs your data, all of it," 22 Febuary 2016. [Online]. Available: http://www.computerworld.com/article/3035595/emerging-technology/artificial-intelligence-needs-your-data-all-of-it.html. [Accessed 28 June 2017].

[5] C. D. Manning, P. Raghavan and H. Schutze, Introduction to Information Retrieval, New York, NY: Cambridge University Press, 2008.

[6] V. N. Anh and A. Moffat, "Index Compression using Fixed Binary Codewords," in *Proceedings of the 15th Australaian Database Conference*, Darlinghurst, Australia, 2004.

[7] V. Glory and S. Domnic, "Inverted Index Compression Using Extended Golomb Code," in *IEEE-Inteational Conference On Advances In Engineering, Science And Management*, 2012.

[8] N. S. G. Mulpuri, "Dynamic Unbounded Integer Compression," Texas State University, San Marcos, TX, 2016.

[9] B. P. Tunstall, "Synthesis of Noiseless Compression Codes," Georgia Institute of Technology, Atlanta, GA, 1967.

[10] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory,* vol. 21, no. 2, pp. 194-203, March 1975.

[11] D. Salomon, Data Compression, 4th Edition ed., London: Springer, 2007.

[12] D. Salomon, "Variable-Length Codes for Data Compression," London, Springer, 2007.

[13] J. Dean, "Linkedin," Linkedin, [Online]. Available: https://www.linkedin.com/in/jeff-dean-8b212555. [Accessed 24 Oct 2017].

[14] J. Dean, *Challenges in building large-scale information retrieval systems,* WSDN, 2009.

[15] G. H. Freeman, "Asymptotic Convergence of Dual-Tree Entropy Codes," in *Pro-ceedings of the Data Compression Conference*, 1991.

[16] G. H. Freeman, "Divergence and the Construction of Variable-to-Variable-Length Lossless Codes by Source-Word Extensions," in *Proceedings of the Data Compression Conference*, 1993.

[17] P. R. Stubley, "Adaptive variable-to-variable length codes," in *Data Compression Conference*, Snowbird, UT, 1994.

[18] F. Fabris, A. Sgarro and R. Pauletti, "Tunstall adaptive coding and miscoding," *IEEE Transactions on Information Theory,* vol. 42, no. 6, pp. 2167-2180, November 1996.

[19] M. B. Baer, "Efficient implementation of the generalized Tunstall code generation algorithm," *2009 IEEE International Symposium on Information Theory,* pp. 199-203, 2009.

[20] S. T. Klein and D. Shapira, "Improved Variable-to-Fixed Length Codes," *Lecture Notes in Computer Science,* vol. 5280, 2008.

[21] S. Deorowicz, "Silesia Compression Corpus," Siliesa University, [Online]. Available: http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia. [Accessed 1 10 2017].