TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

LAURI HAHNE

CHOOSING COMPRESSOR FOR COMPUTING NORMALIZED COMPRESSION DISTANCE

Master of Science Thesis

Examiners: Prof. Olli Yli-Harja
Prof. Matti Nykter
Examiners and topic approved by the Faculty of Computer and Electrical Engineering council meeting on 4th May, 2011.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
HAHNE, LAURI: Pakkaimen valinta normalisoidun pakkausetäisyyden laskemisessa
Diplomityö, 43 sivua
Joulukuu 2013
Pääaine: Signaalinkäsittely
Tarkastajat: Prof. Olli Yli-Harja, Prof. Matti Nykter
Avainsanat: normalisoitu pakkausetäisyys, algoritminen informaatio, pakkainohjelmat

Nykyaikaiset mittausmenetelmät esimerkiksi biologian alalla tuottavat huomattavan suuria datamääriä käsiteltäväksi. Suuret määrät mittausaineistoa kuitenkin aiheuttaa ongelmia datan käsittelyn suhteen, sillä tätä suurta määrää mittausaineista pitäisi pystyä käyttämään avuksi luokittelussa. Perinteiset luokittelumenetelmät kuitenkin perustuvat piirteenirrotukseen, joka voi olla vaikeaa tai mahdotonta, mikäli kiinnostavia piirteitä ei tunneta tai niitä on liikaa.

Normalisoitu informaatioetäisyys (normalized information distance) on mitta, jota voidaan käyttää teoriassa kaiken tyyppisen aineiston luokitteluun niiden algoritmisten piirteiden perusteella. Valitettavasti normalisoidun informaatioetäisyyden käyttö edellyttää algoritmisen Kolmogorov-kompleksisuuden tuntemisen, mikä ei ole mahdollista, sillä täysin teoreettista Kolmogorov-kompleksisuutta ei voida laskea. Normalisoitu pakkausetäisyys (normalized compression distance) on normalisoituun informaatioetäisyyteen perustuva mitta, jossa Kolmogorov-kompleksisuus on korvattu syötteen pakatulla koolla.

Pakattu koko voidaan teoriassa laskea pakkaamalla syöte millä tahansa pakkausohjelmalla, mutta käytännössä tulokset vaihtelevat eri pakkausohjelmien välillä. Tämä työn tarkoitus on luoda viitekehys, jota voidaan käyttää eri pakkausalgoritmien ja -ohjelmien vertailuun normalisoidun pakkausetäisyyden laskemisessa.

Tässä työssä esitettävä vertailuviitekehys kykynee paljastamaan eroavaisuuksia eri pakkainten välillä tapauksissa, joissa erot liittyvät syötteet pituuteen tai syötteen tuottaneen prosessin algoritmiseen luonteeseen. Viitekehystä käytettiin neljän eri pakkaimen vertailuun ja tämä osoitti selviä eroja pakkainten välillä. Tämän lisäksi tulokset mahdollistivat suositusten antamisen siitä, missä tilanteessa eri pakkaimet ovat käyttökelpoisia.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Signal Processing and Communications Engineering
HAHNE, LAURI: Choosing Compressor for Computing Normalized Compression Distance
Master of Science Thesis, 43 pages
December 2013
Major: Signal Processing
Examiners: Prof. Olli Yli-Harja, Prof. Matti Nykter
Keywords: normalized compression distance, algorithmic information, compressor programs

Modern measurement technologies in sciences such as biology produce huge amounts of measurement data. A major problem with this is that the data need to be classified so that it becomes possible to tell, for example, cancerous cells from regular cells. However, there is seldom enough information about the processes producing the data that is getting measured, so traditional feature based classification algorithms might prove useless.

Normalized information distance is a universal metric than can be used to theoretically cluster all kinds of data based on their algorithmic similarity without knowing the interesting features beforehand. However normalized information distance depends on algorithmic Kolmogorov complexity which cannot be computed. Normalized compression distance is a construct based on normalized information distance. Normalized compression distance substitutes the uncomputable Kolmogorov complexity for compressed file size.

Theoretically any good enough data compressor can be used to compute normalized compression distance. In practice different compressors are known to produce dissimilar results. The purpose of this work is to construct a framework for evaluating the performance of different compressors when computing the normalized compression distance.

The evaluation framework that is presented in this work is able to uncover differences in performance of different compressors given different input lengths and input types. Four different compressors were evaluated using the framework and the results show how different features of the compressors affect their performance. In addition, it became possible to give recommendations about which compressors to use for different kinds of inputs.

FOREWORD

This thesis is related to a research project in which I participated during my work as a research assistant at the Department of Signal Processing. This thesis project also lead to a published paper and poster at the 8th International Workshop on Computational Systems Biology held in Zurich, Switzerland on June 2011.

Prof. Matti Nykter and dr. Juha Kesseli acted as my supervisors and advisors during the project. In addition, professor Olli Yli-Harja provided guidance and advice.

Tampere, November 6th 2013

Lauri Hahne

CONTENTS

SYMBOLS AND ABBREVIATIONS

$C(x)$	compressed size of x
$C(xy)$	compressed size of x concatenated with y
E	expected value operator
$H(x)$	entropy of x
$H(Y\|X)$	conditional entropy of Y given X
$H(X,Y)$	joint entropy of X and Y
$I(X;Y)$	mutual information of X and Y
K	Kolmogorov complexity
$K(x)$	Kolmogorov complexity of x
$K(x\|y)$	Kolmogorov complexity of x given y
log	base 2 logarithm
NCD(x,y)	normalized compression distance of x and y
NID(x,y)	normalized information distance of x and y
p	mutation probability in our compressor evaluation framework
$p(X)$	probability mass function of random variable X
$p(x,y)$	joint probability distribution of variables X and Y
LZ77	Lempel-Ziv of algorithm of 1977
LZMA	Lempel-Ziv-Markov chain algorithm
NCD	normalized compression distance
NID	normalized information distance
PPM	prediction by partial matching
RLE	run-length encoding

1. INTRODUCTION

Traditionally classification problems are divided into two categories: supervised learning and unsupervised learning. The difference between these two approaches is that supervised learning expects example data that has been labeled with correct class to be used as a basis for the creation of a classification algorithm. Unsupervised learning, on the other hand, is a process of looking into data, without knowing the classes of given data points beforehand, to generate means of classifying similar data. A close relative of unsupervised learning is clustering that takes some input data and divides it up into different classes.[12]

Modern research methods in biology and other disciplines are characterized by their ability to generate vast amounts raw quantitative data. The sheer amount of the data often makes it impossible to classify parts of it into distinct classes. One reason for this is that a single measurement vector can contain up to millions of different data points for just one sample. Therefore a common approach to classifying this kind of data is to cluster it. However typical clustering methods such as k-means require some a priori knowledge of data, such as the number of different classes. In addition, they might be too computationally expensive for those use cases where the data contains great amounts of sample values. In addition, methods such as k-means rely solely on the numerical representation of the data and do not have a way of looking into the process that actually produced the data that was measured. This sole reliance on numbers is also problematic because the number of spurious correlations is known to grow faster than the number of variables[25].

It has been proposed that normalized information distance (NID) can be used to overcome these problems in clustering. The normalized information distance between two strings x and y is defined as

$$\text{NID}(x, y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}. \tag{1.1}$$

where K is the Kolmogorov complexity and $K(x|y)$ is the conditional Kolmogorov complexity of x given y. Normalized information distance is a universal algorithmic distance metric which means that it tries to exploit the features of an underlying process that generated the data that was measured. Unfortunately normalized information distance depends on the length of the shortest binary program, or Kolmogorov complexity K, which is by definition uncomputable. Therefore an approximation for normalized information distance, normalized compression distance (NCD), has been developed. Normalized com-

pression distance is defined as

$$\text{NCD}(x, y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}, \tag{1.2}$$

where $C(x)$ is the compressed size of x using some general purpose data compression program. This makes it feasible to compute NCD even for large masses of data because general purpose compression programs are known to perform well on current computer hardware.[8]

Normalized compression distance has been showed to cluster successfully as diverse data as sequenced genomes and Russian literary works[8]. In addition, it has been demonstrated that normalized compression distance can be used to classify random boolean networks given data about their structure and dynamics[20]. This effectively demonstrates, that normalized compression distance is able to uncover algorithmic features behind given measurement data which makes it a lot more universal in clustering as traditional methods such as k-means.

The problem with normalized compression distance is that it is dependent on the choice of the compressor. It is well known that different data compression programs are based on different algorithms and they have different kinds of compression efficiency on different kinds of data. Cebrian et al. have researched the effect of compressor choice on normalized compression distance. They found several limitations of the compressors based on the length of the input file.[7] The problem with the work of Cebrian et al. is that their work only involves computing $\text{NCD}(x, x)$, that is only computing the distance of a string against its perfect copy. In addition, they limit their input to the Calgary corpus which mostly consists of natural text.

The aim of this thesis is to test different data compressors in computing the NCD so that we can uncover possible inherent weaknesses and limitations of these compressors and to be able to give generalized recommendations on which compressor to use when computing normalized compression distance. The ultimate goal is to produce a large amount of data points that can be used to analyze different compressors. This is enabled by the use of grid computing that makes it possible to harness large amounts of processing power in short time frames.

The results are obtained by constructing two different parametric randomness models where it is possible to fine-tune the similarity of two strings. Then it becomes possible to compare the computed normalized compression distance against the parametric randomness value that was used to generate the strings. In addition, the model allows to adjust the length of the strings to uncover any possible limitations in the functionality of compressors when it comes to the length of the input.

The evaluation model was run on a grid computing system installed at the Tampere University of Technology. This allowed to gather more data points and granularity than

previous studies of the same subject had been able to use. The results were analyzed with special attention paid to being able to uncover innate limitations of the used general purpose compression programs. This made it possible to distinguish how certain features, such as block size and look-ahead window, had very specific effects on the computed NCD.

This thesis is structured so that previous work that established normalized compression distance as a legitimate research topic and an interesting and powerful tool is introduced first. After that, relevant parts of information theory are reviewed so that the reader should become familiar with the theoretical background of normalized compression distance and general purpose data compressors. Finally an evaluation framework is formed so that it becomes possible to test different data compressors in relation to computing NCD. This is followed by an analysis of the results and recommendations on which compressor to use.

2. BACKGROUND

The purpose of this chapter is to present a cohesive overview of normalized compression distance, or NCD, and related concepts. This begins by highlighting common and uncommon uses of normalized compression distance and then move on into the theoretical background of measuring information and using information to measure likeness and similarity. Finally an overview of common common general-purpose compression algorithms is given so that that information can later be used to discuss the behavior of particular compressors with different kinds of input data.

Normalized compression distance is a universal distance metric in the sense that it can be used to cluster and classify all kinds of data. This is particularly useful because normalized compression distance assumes no knowledge about the features of data. This is further accentuated by the fact that normalized compression distance has been used to cluster and classify data as diverse as genomes[8], Russian authors[8], and random Boolean networks[20].

What also makes normalized compression distance highly applicable is that general-purpose compression algorithms provide good performance on common computer hardware even for large lengths of input. When this is combined with normalized compression distance's universality, result is that we get a framework that can be used to cluster and classify virtually any kinds of data with good performance without knowing the feature space of the data.

One field of research where this has proven to be invaluable is genetics and biology in general. Sequenced genomes in themselves contain vast amounts of data and an unknown feature space. In addition, modern measurement technologies for gene expression and other biological processes produce a lot of data. Bornholdt has discussed[5] that even the theoretical foundations of the functionality of cells is staggering and more levels of abstraction are needed to turn the problem of understanding living organisms into a fathomable level.

One such abstraction is turning genetic information from the space of atomic interactions and dynamics over time to a view of a genetic control network. Even this model can be further simplified by turning the network into a simple Boolean model where each gene is either in the on state or the off state.[5] This kind of method has been used in conjunction with normalized compression distance to show that gene expression dynamics in the macrophage exhibit criticality[19].

It has also been shown that normalized compression distance can be used to uncover information about the behavior of Boolean networks given data about their structure and dynamics[20]. This work also extends to using metabolic network structure data from Kyoto encyclopedia of genes and genomes[14] to build a phylogenetic tree highlighting that normalized compression distance can uncover fundamental structural differences within these metabolic networks[20].

More traditional analysis of the structure of networks is based on concepts such as calculating statistics like connectedness or by finding and identifying typical network motifs such as feed-forward and feed-back. As such, the structural analysis based solely on the functions, or connections, is what define the next state of the network. Analysis based on dynamics, on the other hand, is related to the succeeding states of the network. This type of analysis thus involves setting the networks state to some known one and then computing a number of succeeding states for this. Typically this kind of analysis is performed for many different initial states and possibly even for all possible initial states if sufficient computational resources are available.[18]

Normalized compression distance provides a valuable information theoretic approach to analyzing both structure and dynamics. This frees us from having to choose and guess interesting features by hand because by virtue of normalized compression distance being the every effective distance[8], it should be able to pick up all features that are present in the data. This has been shown to be the case in earlier research[20, 19].

The main focus of discussion here has been biological networks and data sources which is mainly due to the nature of other research made by the research group where I worked during the practical phase of this thesis writing project. However nothing prevents from extending the concepts of analysis to other kinds of networks. Various kinds of real-world complex networks, such as social and citation networks, have been studied by mainly analyzing their structure[18]. Given the diverse nature of real-world networks, having a powerful tool like normalized compression distance provides an easy to use but very effective analysis construct which circumvents the problem of performing analysis on the individual connections and nodes of the networks.

2.1 Measuring Information

To be able to present the functionality of normalized compression distance, it is necessary to first review key concepts of information theory. This also illustrates the fundamental simplicity and elegance of NCD in relation to theoretical constructs and real-world compression programs. Fundamentally normalized compression distance is a parameter-free similarity metric based on information[8]. Thus the discussion should begin by defining what is information and how it relates to NCD. The equations given here are as described in Elements of Information Theory[10] unless otherwise stated.

The most basic and fundamental measure of information is entropy which is a measure

of uncertainty of a random variable [10]. Shannon interprets entropy as an answer to the question, how much choice is involved in producing a given event[24]. As a natural extension this can be interpreted as, how much information about these choices is conveyed in a event drawn from random variable X.

For a discrete random variable X, entropy $H(X)$ is defined by

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x) \tag{2.1}$$

$$= E \log \frac{1}{p(X)} \tag{2.2}$$

$$= -E \log p(X), \tag{2.3}$$

where $p(X)$ is the probability mass function of X and E is the expected value operator. A base two logarithm is traditionally used and therefore the unit of entropy is bits. In addition, it is generally assumed that $0 \log 0 = 0$ so that adding terms with zero probability does not affect entropy [10].

It is notable that entropy is a function of the distribution of random values and is not dependent on the actual realized values of that variable. The entropy of variable X can be understood to be the minimal amount of information that is required to describe the variable. This interpretation is further accentuated and connected to Shannon's interpretation by the fact that it has been shown that the minimum amount of binary questions required to determine X lies between $H(X)$ and $H(X) + 1$ [10].

The notion of entropy can easily be extended to multiple discrete random variables X and Y given that their joint probability distribution $p(x, y)$ is known. In this case the joint entropy $H(X, Y)$ is defined as

$$H(X, Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(x, y) \tag{2.4}$$

$$= -E \log p(X, Y) \tag{2.5}$$

which intuitively follow from the definition of entropy[10]. A natural explanation for joint entropy that follows the explanation of entropy is, how many choices are involved in drawing random variables from both distributions X and Y. A more interesting notion of conditional entropy may be defined using entropy as defined earlier. The conditional entropy of Y given X or $H(Y|X)$ is

$$H(Y|X) = \sum_{x \in \mathcal{X}} p(x) H(Y|X = x) \tag{2.6}$$

$$= -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(y|x) \tag{2.7}$$

$$= -E \log p(Y|X). \tag{2.8}$$

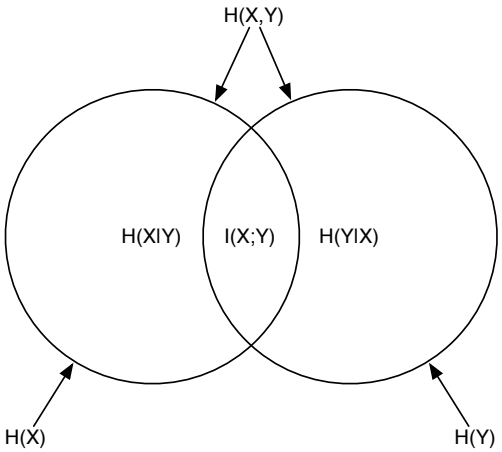

Figure 2.1: The relationship between entropies $H(X)$ and $H(Y)$, conditional entropies $H(X|Y)$ and $H(Y|X)$, and mutual information $I(X;Y)$ visualized as Venn diagram. Based on a drawing in [10].

As such conditional entropy is the expected value of the entropies of the conditional distributions averaged over the random variable that is conditioning.[10] This, in turn, can be interpreted as how much choice is left in a variable drawn from Y once we already know the value of a variable drawn from X.

The concept of conditional entropy can be further extended into information between variables X and Y that is called mutual information. The mutual information $I(X;Y)$ is the reduction of uncertainty about X given the knowledge about Y. Mutual information $I(X;Y)$ can be expressed using entropy and joint entropy as

$$I(X;Y) = H(X) - H(X|Y) \tag{2.9}$$

and it follows that

$$I(X;X) = H(X) - H(X|X) = H(X). \tag{2.10}$$

Therefore entropy is also sometimes called self-information as it expresses the information known about random variable given the variable itself.[10] The relationship between these different measures of information is visualized in Figure 2.1.

All these methods for computing entropy and mutual information require the knowledge about the distribution of the chosen random variable and there appears to be no way to reliably compute entropy or mutual information based only on samples of the random variable. This is unfortunate because when dealing with real world data it is often impossible to determine the real distribution of the measured random variables. Therefore we need more powerful tools than these simple statistical measures to work with real-world data.

In addition, this statistical approach fails to take into account that many real-world information sources such as natural language are combinatorial in nature[15]. There do ex-

ist ways of computing entropy for purely combinatorial sources and for example entropy $H(X)$ for a set of N elements is

$$H(X) = \log N. \tag{2.11}$$

If we fix X we also get a result analogous to Equation 2.10 so that the information contained by a string is its entropy

$$I = \log N \tag{2.12}$$

given that our logarithms are base-two. According to Kolmogorov [15] this combinatorial approach to information also holds on its own without needing assert that the statistical equations hold true. Yet these measures fail to answer the simple question what is the information conveyed by individual object x about object y [15]. The closest answers this far are mutual information and joint entropy. However both of these fall back to the statistical domain.

Kolmogorov complexity provides us with the answer to the question how much information is conveyed by string x. Kolmogorov complexity of string x, or $K_{\mathcal{U}}(x)$, is the length of the shortest binary computer program that outputs x on a universal computer. That is

$$K_{\mathcal{U}}(x) = \min_{p:\mathcal{U}(p)=x} l(p) \tag{2.13}$$

where $l(p)$ is the length of program p [10]. Kolmogorov complexity $K_{\mathcal{U}}(x)$ is usually shortened just as $K(x)$ or K. Kolmogorov complexity is conceptually based on the notion that a universal computer is the most effective data compressor and consequently Kolmogorov complexity is shown to estimate entropy

$$E\frac{1}{N}K(X^n|n) \to H(X) \tag{2.14}$$

given an i.i.d. stochastic process from which the string is drawn[10]. However this connection is seldom used. A more interesting feature is that Kolmogorov complexity can be estimated by using a general purpose data compressor because the decompression program may be considered a computer itself that executes the compressed file as its program[8]. This makes it possible to estimate Kolmogorov complexity for real-life strings of binary data using a computer with a general purpose compression program.

2.2 Measuring Similarity

Traditionally similarity of two objects is measured based on their features[8]. This model depends on identifying the interesting features, extracting the features from data and measuring similarity based on these hand-selected features for multiple objects[12]. It is apparent that this kind of method requires domain knowledge about the objects that are being

measured for similarity and as such this kind of method is inapplicable for data whose major features are unknown.

Cilibrasi and Vitányi propose a concept called every effective distance which encapsulates all known and unknown distance metrics. The intent of the distance metric is to be able to exploit all similarities between objects that all other metrics are able to uncover separately.[8] One such metric is described next.

Every effective distance encapsulates the problem of inherent need of domain knowledge. This raises the question would there be any other way of measuring similarity that does not depend on features but is still able to effectively represent all information contained in features. Cilibrasi and Vitányi propose [8] one such way of measuring similarity by trying to answer Kolmogorov's question what information is conveyed by individual object x about individual object y. For the sake of simplicity it is assumed that x and y are binary strings of bytes.

The question of what information x conveys about y can be answered by computing the Kolmogorov complexity of x, y, and their concatenation xy or $K(x)$, $K(y)$ and $K(xy)$ respectively. If $\max\{K(x), K(y)\}$ is close to $K(xy)$ we know that the concatenation of x and y only contains marginally more information than x or y on their own and that as such x and y are similar. The notion of normalized information distance is based on this concept that the complexity of x given y, or $K(x|y)$ is small when x and y are similar. The normalized information distance is more formally defined as

$$\text{NID}(x, y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}, \tag{2.15}$$

which quantifies the information distance between two inputs x and y symmetrically so that the order of x and y does not matter. In addition, normalized information distance is proven to be a universal metric in the sense that two similar objects produce a small value. Unfortunately, the Kolmogorov complexity is uncomputable so there is little practical use for normalized information distance outside of theoretical context. [8]

An approximation of Kolmogorov complexity $K(x)$, compressed size $C(x)$, can be obtained using general purpose compression algorithms. In the ideal case, $C(x)$ would equal $K(x)$. Using $C(x)$ we can approximate normalized information distance by using the normalized compression distance which is defined as

$$\text{NCD}(x, y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}}, \tag{2.16}$$

where $C(xy)$ denotes the compressed size of the concatenation of x and y. The ideal compressor C is called normal compressor and it satifies the following conditions[8] up to an additive $O(\log n)$ term in relation to the length of the input:

1. Idempotency: $C(xx) = C(x)$, and $C(\lambda) = 0$ for an empty string λ. A reasonable

compressor is expected to be able to convey the duplication of input data with zero overhead and should be able to compress empty input to an empty output.

2. Monotonicity: $C(xy) \geq C(x)$. A compressor should naturally exhibit monotonicity because the concatenation xy is expected to convey more information than just x or y separately.

3. Symmetry: $C(xy) = C(yx)$. An ideal compressor should work ideally irrespective of the order of the input.

4. Distributivity: $C(xy) + C(z) \leq C(xz) + C(yz)$.

When a compressor fulfills the criteria of a normal compressor, normalized compression distance is a similarity metric[8]. This makes normalized compression distance a viable choice to be used as metric when clustering and classifying data. Normalized compression distance has successfully been used to classify and cluster diverse sets of data including genomes of viruses, works of Russian authors and both structural and dynamical features of Boolean networks, an abstract model of biological regulatory networks [8, 19].

Because normalized compression distance does not depend on explicit feature extraction, it becomes evident that the compressor used in calculating normalized compression distance should be able to capture and model the essential underlying properties and processes behind the input data. This makes the choice of compressor a vital part of the clustering and classification process. For real-life purposes, we are interested in cases where C is an existing general purpose compressor program that can be run on a standard PC.

2.3 Compression Algorithms

We can clearly see from the definition of normalized compression distance given in the previous chapter that the distance measure gotten by using normalized compression distance solely depends on the choice of the compression algorithm or program. Previously the choice of compression algorithm, or compressor, has been done by a process of trial and error or by intuition. The fundamental building blocks of several compression algorithms are discussed in this chapter so that the properties can be later used to explain the different kinds of results that were obtained in the course of the analysis that was performed in the context of this work.

A typical general purpose data compression algorithm can be divided into two distinct parts, decorrelation and encoding. Decorrelation transforms the data in a way which minimizes its autocorrelation. This typically results in a transformed signal which has long runs of consecutive symbols. The decorrelated signal is thereafter encoded in a way that minimizes its length given some constraints. A trivial example of encoding would be encoding a string consisting of 500 zeros as “500 zeros”.

General purpose data compression programs typically combine many different algorithms presented here so that they can obtain better data compression than any one algorithm by itself. In addition, compressor programs may feature tweaked more complex versions of the algorithms which may provide superior compression or performance in relation to the basic version of the algorithm.

2.3.1 The Lempel-Ziv Algorithm of 1977

Lempel and Ziv introduced two separate but similar compression algorithms in 1977[29] and 1978[30]. These are generally known as Lempel-Ziv algorithms 77 and 78, or LZ77 and LZ78, for short. The Lempel-Ziv algorithm 77 is a simple dictionary based compressor with adaptive dictionary and a sliding window and has been proven as asymptotically optimal compressor for stationary ergodic sources[10]. The 1978 algorithm, on the other hand, is a tree-structured algorithm similar to the 1977 version but it will not be discussed in any greater detail in this thesis because the 77 version is more commonly used and none of the data compressor that are used as part of this work use the LZ78 variant.

The key idea of the Lempel-Ziv 77 is to parse the input string as a stream and try to encode the current byte and the ones following it as a pointer that tells the location of an identical string further back in the already compressed file. This process is limited by a sliding window which is typically divided into two parts, the look-ahead buffer and the distance that can be referenced backwards. The encoder tries to match the currently encoded byte and a maximum number of its successors from the data that it can point backwards. The encoder then encodes this as a pointer backwards and the length of how many bytes to read from there before moving to the next symbol after the substring that has just been encoded. If no match is found, the current byte is encoded as is. To separate raw bytes from pointers and lengths, a flag is typically used as the first bit or byte of the encoded symbol to signal whether it is a pointer or just the value.

This process is better explained by a decoding example adapted from [10]. Lets assume that our encoded symbols are either pairs in the form (0, byte) or triplets (1, pointer backwards, symbols to read). In this case the first bit that is either one or zero signifies whether the following data is just a raw byte or a pointer backwards with length information. Given that we have symbols (0,A), (0,B), (1,1,1), (1,3,1), (1,4,8) in this notation. When we start decoding from the beginning, the first two symbols decode directly into string AB. The third symbol on the other hand tells the decoder to look backwards one byte in the decoded string and read one byte. After this the decoded string becomes ABB and after that the next symbol tells to go back three bytes and read one and we get ABBA. The fifth and last symbol exhibits a more interesting behavior where the decoder is told to go back 4 bytes and read 8 bytes beginning from there. This might sound paradoxical but after little thinking it becomes intuitive that the next four bytes from current position need to be the same as the four previous so the symbol translates into ABBAABBA and the whole decoded

string is ABBAABBAABBA.

The main problem with the Lempel-Ziv algorithm of 1977 is that it does not define the format of encoded symbols and therefore nothing guarantees that they are the most most compact or otherwise optimal ones[10]. In addition, the length of the look-ahead buffer limits the length of strings that can be efficiently encoded whereas the maximum value of the backwards pointer limits how far back the encoder might look for occurences of the current string.

2.3.2 Huffman Coding

Prefix codes are a set of variable-length codes that can be used to encode symbols. Prefix codes have the feature that no code is contained in the beginning of another code so they can be uniquely decoded. This also makes it possible to transmit or save the encoded string as is because it is uniquely decodable once decoding is started from the beginning.

Huffman code is the optimal prefix code in the sense that no other algorithm can produce a prefix code for a message that would be shorter than the one produced by the Huffman algorithm[13]. Huffman coding works by assigning long binary codes to symbols, or bytes, with the least probability whereas more common symbols get assigned shorted codes. Huffman algorithm is typically used for binary codes but nothing prevents using it for higher-order codes if necessary. Only binary codes are discussed here but the extension to other radices follows straightforwardly.

The Huffman algorithm works by building a binary tree from the symbols starting from the leaves. First the two symbols with lowest probabilities are connected to form a single node which is assigned the combined probability of the two symbols. The combined node is treated as a symbol thereafter and the original two symbols are discarded from the nodes that are being processed in the further rounds of the algorithm. The process is then repeated until all the symbols have been combined into a one single root node. Then all the branches of this tree are assigned binary symbol 1 or 0 deterministically so that the left branch always gets 1 and the right one always gets 0 or vice versa. The actual binary prefix codewords for the original symbols can then be read starting from the root node and reading the symbols of the branches in order.

An example of a Huffman binary tree and the associated codes is presented in Figure 2.2. The original symbols A, B, C, D and E, with probabilities 0.25, 0.2, 0.3, 0.175 and 0.075 respectively, have been combined according to the Huffman algorithm and the binary symbols have been assigned by giving the upper branch always 1. The generated binary codewords are also presented in a table format for clarity. It should be noted that the generated codes are not uniquely optimal as for example the ones and zeros might have been assigned vice versa while the lengths of the codewords would have remained the same.

Using the codewords presented in Figure 2.2 it is possible to encode the string ABBA

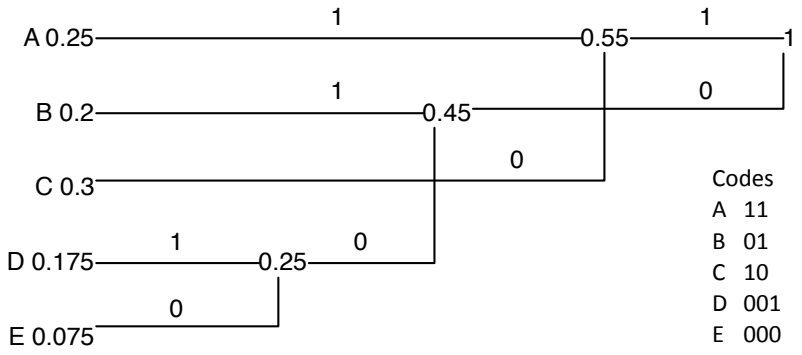

Figure 2.2: A sample of a Huffman tree and the associated probabilities and binary codes. The tree is constructed by always combining the branches with lowest combined probabilities and the resulting prefix codewords are read beginning from the top node.

as 11010111 and ACDC as 111000110. The encoded length of ABBA is 8 bits whereas the encoded length of ACDC is 9 bits whereas a traditional 8-bit text representation would require 32 bits for each word. This demonstrates how a string containing common symbols such as A and B takes less bits to represent than a string containing less frequent symbols like D.

2.3.3 Burrows-Wheeler Transform

The Burrows-Wheeler transform is a lossless transform that reduces string's entropy by reordering its bytes. The Burrows-Wheeler transform does not compress the string by itself but makes the compression easier for encoders that exploit local correlation such as move-to-front transform which is describe in Section 2.3.4.[6]

The Burrows-Wheeler transform encodes the input string so that the encoded version is more likely to have instances of the same byte close to one another than the original one. The power of the Burrows-Wheeler comes from the fact that the encoded string can be decoded back to original by only knowing the encoded string and an index which describes the location of the original string in the sorted list of lexicographically ordered strings[6].

A naive implementation of the Burrows-Wheeler transformation described below is as represented in the original Burrows-Wheeler paper[6]. The algorithm works by generating all possible rotations of the original string and sorting these lexicographically. The index of the original string in this sorted list is stored so that it can be used later when decoding the encoded string. The actual encoding is done by taking the last byte from each one of rotated strings in the lexicographical order. This string is traditionally called L and the index I.

The decoding is done by only knowing the last letters of the sorted strings in the lexicographical order of the whole strings L. If we think of the list of lexicographically sorted

rotations	ordered rotations
ananas	anana**s**
nanasa	anasa**n**
anasan	asana**n**
nasana	nanas**a**
asanan	nasan**a**
sanana	sanan**a**

Table 2.1: The Burrows-Wheeler transform of word ananas. The original word is presented in the upper-left corner of the table and its rotations underneath it. The rotations are presented in lexicographical order on the right side and the Burrows-Wheeler encoded output snnaaa is highlighted in the last column. The index of the original string in ordered strings I is 0 in this case. The index I is required to unambiguously decode the encoded string.

strings as rows of matrix M then the encoded string L is the last column of the matrix M and the first column F can be formed by sorting the column L. Given that we produce a copy of M, M', that is formed from M by rotating its rows once to the right, then L is the first column of M' and F is the second column of M'. As M' is sorted lexicographically in relation to its second column F we known based on the rotations that that instances of any byte b in L occurs in same order as in F. This allows us to create mapping T from F to the the corresponding index for the characters in L. That means

$$F[T[j]] = L[j] \tag{2.17}$$

for all j. In addition we know that $L[i]$ cyclically precedes $F[i]$ in the decoded string S because L and F are adjacent columns in M and M'. We can substitute Equation 2.17 with $i = T[j]$. We get that $L[T[j]]$ cyclically precedes $L[j]$ in S. Knowing that index I gives us the position of original string in M we also know that the last character of S is $L[I]$. Now we can construct the original string S using relation

$$\text{for each } i = 0, \ldots, N-1 : S[N-1-i] = L[T^i[I]], \tag{2.18}$$

where $T^0[x] = x$ and $T^{i+1}[x] = T[T^i[x]]$.

Other versions of the algorithm exist where the use of index I is avoided by appending an end-of-file character to the original string before encoding it. Other versions of the decoding process also exist which are based on generating the matrix M row-by-row by sorting and appending L and F to an empty matrix.[27]

An example of Burrows-Wheeler transforming the word ananas is presented in Table 2.1. The encoded output of ananas is snnaaa with $I = 0$ which demonstrates the power of Burrows-Wheeler transform to move instances of same byte close to each other.

input	output	dictionary
ananas	0	abcdefghijklmnopqrstuvwxyz
a**n**anas	0,13	abcdefghijklmnopqrstuvwxyz
an**a**nas	0,13, 1	nabcdefghijklmopqrstuvwxyz
ana**n**as	0,13, 1, 1	anbcdefghijklmopqrstuvwxyz
anan**a**s	0,13, 1, 1, 1	nabcdefghijklmopqrstuvwxyz
anana**s**	0,13, 1, 1, 1, 18	anbcdefghijklmopqrstuvwxyz

Table 2.2: An example of move-to-front transform for the word ananas. The input is shown left and the encoded output after the bold byte is encoded is displayed on the middle. The adaptive dictionary is shown on the right so that version that was used to encode a byte is presented on the same row. Indexing of the dictionary starts from zero.

2.3.4 Move-to-front Transform

Move-to-front transform is a data compression scheme that exploits local correlation such as symbols that are located close to other instances of the same symbols[4]. The algorithm is based on the simple notion that when using variable length codes, it is possible to exploit the short codes in case that the source string can be presented in a form that minimizes the amount of symbols needed to represent it.

Move-to-front transform achieves this reduction in the required symbols by dynamically altering the codebook while encoding. The encoding begins by forming a list of codewords such as bytes 0-255 in order. When a byte is encoded, its index in the list is output and its item in the list is moved to in front of the list. This new list is used when encoding the next byte and the list is updated on each round. This causes the most frequently used bytes to be in the front of the list and the smallest index values are used in the encoded output the most. In case that the distribution of bytes changes in the input, the encoder will automatically adjust for this without any extra steps.[4]

An example of computing the move-to-front transform for the string ananas is displayed in Table 2.2. It is notable how move-to-front transformation exploits local correlation and turns most of the string into ones. However the output contains four different symbols in this case whereas the input only contains three. This in an example of how move-to-front transform does not guarantee that its encoded output is optimal.

In addition, move-to-front transform produced consecutive identical symbols in its output. This is highlighted by the example in table 2.2 where the symbol 1 is repeated consecutively. This is clearly a local correlation that could be further exploited to provide more compression.

In optimal case move-to-front transform decreases the amount of different symbols needed to encode a string given that the bytes in string correlate locally. However, the length of the encoded output might actually grow in the case that the input string does not locally correlate[4]. In addition, in no case does the move-to-front transform guarantee such an optimality as Huffman coding does[4]. Therefore it makes sense to combine

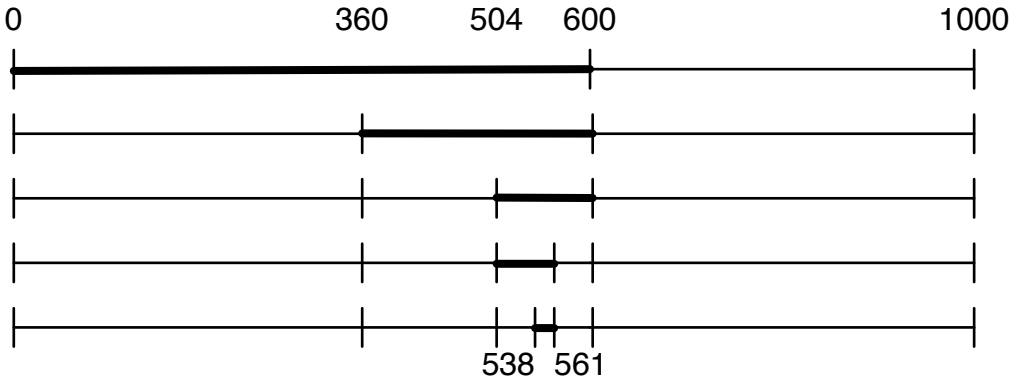

Figure 2.3: An example of range coding process. The coding process that is demonstrated here is explained in the text section.

Burrows-Wheeler transform for local correlation, move-to-front transform for the reduction of symbols and Huffman coding for optimal symbol lengths. One compressor that does all this is bzip2 that is described in greater detail in Section 2.3.9.

2.3.5 Range Coding

Huffman coding is optimal for encoding strings which need to be encoded on a per byte basis. Yet Huffman codes are not optimal when it is possible to encode whole message instead of individual bytes because codeword lengths in Huffman codes are integral whereas the actual entropy is likely to be fractional so Huffman coding is only able to represent entropy within one bit.[10] This problem can be avoided by using either range coding[16] or arithmetic coding[21]. Both of these coding methods encode the whole message as one symbol. Range coding does this by encoding the message into an integer whereas arithmetic coding encodes the message as a rational number between 0 and 1. Instead of giving us just the one integral or fractional number that represents the whole message, both methods actually provide us with a range from which the single representation may be chosen. This makes it possible to pick the encoded symbol so that it is a prefix code that falls completely within the given range and we do not need to save the whole code but just the prefix bits[16, 21]. Only range coding is discussed here in greater detail as it is used by the Lempel-Ziv-Markov chain algorithm described in Section 2.3.10 but the encoding and decoding processes are virtually identical for both methods.

The encoding process for range coding requires in the simple non-adaptive case the knowledge of probabilities for all symbols in the input string beforehand. For the sake example let us assume symbols A and B with probabilities 0.6 and 0.4 respectively and that we want to encode string ABBAB. The encoding process begins by choosing a range into which the message is going to be encoded. In this case we can choose the range to be $[0, 1000)$ but the actual length of the range does not really matter as there are techniques

that allow to extend the range in the middle of the encoding process. The algorithm can be used for any radix but for the sake of easy readability radix 10 is used in the following example. Using radix 2 would produce more optimal results as radix 10 provides more coarse way to describe the information content. The example that is presented here is illustrated in Figure 2.3.

The range is divided into parts that correspond to the probabilities in their size. This gives us range $[0, 600)$ for A and $[600, 1000)$ for B. As our first symbol to be encoded is A we choose the range $[0, 600)$. This range is then further divided into smaller ranges based on the probabilities to encode the next symbol. Now that we want to encode the second symbol B we need to choose the corresponding range out of $[0, 360)$ and $[360, 600)$. This yields us range $[360, 600)$ after the second symbol. To encode the third symbol we yield ranges $[360, 504)$ and $[504, 600)$ and choose $[504, 600)$. The fourth one is chosen from $[504, 561.6)$ and $[561.6, 600)$ to be $[504, 561.6)$. And the final B will be chosen from $[504, 538.56)$ and $[538.56, 561.6)$ so these the whole string ABBAB can be represented by any integer in the range $[538.56, 561.6)$ and we can now exploit the prefix property and just output 54 or 55 as both ranges $[540, 550)$ and $[550, 560)$ are completely within range $[538.56, 561.6)$. Numbers 54 and 55 are completely unambiguous prefix codes within this range as numbers 54 and 55 would be encoded as 055 and 056 respectively because we are only allowed to omit numbers from the end. The fact that we have two different valid prefix codes for the same string is due to the coarseness of our radix 10 representation. If radix 2 would had been used, it would have been more likely that there would had been only one prefix code to represent larger part of the range.

The decoding of the encoded message works in a very similar fashion. The decoder begins by constructing the original range $[0, 600)$ for A and $[600, 1000)$ for B. The prefix code 54 is the taken to be any integer in the range $[540, 550)$ and the decoder sees that this falls into A's range and outputs A. The decoder then divides the range $[0, 600)$ into smaller ranges just like the encoder did. This process of recognizing the correct range and further dividing it into subranges is continued until the whole message ABBAB is output. It should be noted that to stop outputting symbols, the decoder needs to know the amount of symbols that it should output or the uncoded string should contain an end-of-file character than is then also encoded so that the decoder knows to stop once the end-of-file character is output.

2.3.6 Prediction by Partial Matching

Prediction by partial matching, or PPM, is an adaptive data compressor that uses partial string matching and variable length context[9]. At its core, PPM is an adaptive algorithm that encodes symbols based on their context. Context of length N is defined here as a string of N symbols that precede the current symbol.

The core of PPM is based on an arithmetic coder similar to the range coder described in

section 2.3.5. This arithmetic coder is fed by keywords that describe a string of symbols. These keywords are derived using a Markov model of order N which predicts the following symbol based on its context. If a symbol X is not present in the model of order N then a special escape keyword is emitted and the order changed to $N - 1$. If no context contains X then the model escapes all the way to zero-order and the symbol X is transmitted as is. Once the symbol has been output for the first time, its current context is added to the list of known contexts and used in the future. Both encoder and decoder construct the list of contexts in a similar manner so no special knowledge about the contexts need to be transmitted beforehand as the decoder can infer the contexts while decoding.[9]

PPM can achieve great compression rate for sources such as natural language but its compression speed is lacking[17]. Fortunately PPM can be somewhat optimized using methods described in [17] such as keeping the contexts in a tree data format in memory.

PPM has failed to gain any major support or use as a general data compressor program even though it compresses efficiently compression-wise. This is likely mostly due to the fact that PPM is typically much slower than common compression programs and the trade-off between time and compression has not been good enough for PPM to be considered.

2.3.7 Run-length Encoding

Run-length encoding, or RLE, is a somewhat trivial data compression scheme in which consecutively repeated symbols are encoded as the symbol itself and how many times it is repeated. Run-length encoding is useful for compressing data that has long chains of repeated symbols. Such data might be for example black and white images transmitted by fax, that are mostly white, or output sequences produced by move-to-front transform as described in section 2.3.4. As such run-length encoding provides a valuable tool to further compress the output of move-to-front transform which usually still has some exploitable local correlations left.

2.3.8 gzip

Gzip is a compression format that was standardized in 1996 by IETF. The purpose of gzip is to be a data compressor that is CPU and operating system independent, provides compression performance comparable to other algorithms of its time, and be patent free. [11]

The compression functionality of gzip is based on a combination of Lempel-Ziv algorithm of 1977 for decorrelation and Huffman coding for compressing its result optimally. The operation of gzip is not block-based but it is limited by the LZ77 features of the length of the backwards pointer and the size of the look-ahead buffer. As such gzip provides compression efficiency that is typically considered good enough but not part of the cutting edge. The authors of gzip have decided to call their combined algorithm deflate. The

gzip program is universally available on Unix-type computer operating systems and is therefore commonly used for general purpose data compression but is getting replaced by bzip2 and other more modern compression programs.

2.3.9 bzip2

Bzip2 is a popular open-source compression program developed by Julian Seward. Bzip2 is popular most likely because it is patent-free unlike zip and provides better compression ratio in addition to being fully cross-platform.[23] Bzip2 is fundamentally a block-based compressor that divides its input into blocks of 100–900 kilobytes. The block-size can be adjusted when running the compressor but it stays the same during the whole execution. [22]

Bzip2 works by applying multiple different transformations to the blocks of the input file[22]. This is to exploit the fact that different compression methods exploit different kinds of correlations within the file. In addition, the output of one transformation might be more or less of an ideal input to another transformation. While this cascading complexity might add superficial computational complexity to the bzip2 algorithm, in practice it performs remarkably well and its compression efficiency is within a reasonable threshold of fundamentally more computationally complex compressors such as PPM [23].

Bzip2 begins its process by applying the Burrows-Wheeler transformation to its data blocks. This is performed in place in memory so that all the different rotations of the original string do not need to be generated into the computer memory. [22] As was discussed in section 2.3.3, the Burrows-Wheeler transformation does not compress the data in itself but makes it more susceptible for other compression methods. Therefore the Burrows-Wheeler transformation is followed by a move-to-front transformation [22] which allows bzip2 to exploit the local correlations in the output of the Burrows-Wheeler transformation.

After processing data with all these transformations that do not actually compress it but rather exploit local and global correlations to represent the data with less symbols, bzip2 performs the actual compression by utilizing run-length encoding and Huffman coding [22]. Run-length encoding is done first because it allows bzip2 to compress the long occurrences of identical symbols in the output of combined Burrows-Wheeler and move-to-front transformations. Once the run-length encoding has reduced the amount of actual symbols in the block being compressed, Huffman coding is used to represent the output symbols with more optimal lengths.

2.3.10 LZMA

The Lempel-Ziv-Markov chain algorithm, or LZMA, is a compression algorithms originally implemented by 7-zip. There is very little documentation available publicly besides

the original source code. According to Wikipedia[28] LZMA uses a modified LZ77 algorithm with huge dictionary size and some modifications for efficiently coding often repeating match distances. The output of the dictionary coder is then encoded with a range coder to reduce its size. A quick peek through the XZ Utils source code[3] validates that their implementation of LZMA does indeed contain a Lempel-Ziv type encoder and a range encoder among other things.

LZMA has gained popularity during the recent years and the XZ Utils version is the one that often ships with Unix-type systems whereas 7-zip is commonly used in Windows to support LZMA. The surge in popularity is likely to be related to LZMA's great performance in both compression efficiency and the time that it takes to compress and decompress data. In addition LZMA is considered to be patent-free so that it can be used freely.

2.3.11 Effect of the chosen algorithm on computing NCD

The main goal of this thesis is to uncover the effect of the compression algorithm when computing the normalized compression distance. At this point I have covered the theoretical basis and functionality of different general-purpose compression programs that can be run on a modern computer. This makes it possible to compare the different features of the compression programs and how they relate to computing normalized compression distance using these compressors.

Let us recall that the most interesting property of NID and NCD is their universality. That means that the chosen compressor should work efficiently for as many types of input data as possible. All of the compressors presented in this chapter are designed to be universal in the sense that they can compress any kind of file instead of being limited to only certain types of files such as images or audio. This has also affected their design so that the chosen algorithms are ones that work well on many kinds of data.

However, if it was known beforehand which kind of data was to be compared, then a priori knowledge could be used to aid in the selection of the compression algorithm. The usage or design of a specialized compression algorithm has been left out of this work intentionally as the main interest is in universal compression algorithms that work without a priori information about the data and can uncover similarities that might not be apparent.

A limitation that is a problem with most of the compression methods described in this chapter is that they are block-based. That means that the data is first divided into fixed-size pieces that are compressed more or less separately. Cebrián et al. [7] have discussed the effect of bzip2's block size and gzip's look-ahead buffer and backwards pointer on NCD in great detail. Their work clearly shows and explains why these algorithms fail to properly work as universal compressors when the length of the input exceeds their built-in limits.

The mechanism that underlies these block-based compressors is their ability to find repeated symbols. This is a valuable property when dealing with data that shows repeating

properties. Unfortunately these compressors cannot capture similar strings but rather work only on exact sub-string matches. The compressed symbols that represent found matches and their locations can further be compressed by statistical coding, such as Huffman coding, or by range coding.

The nature of Huffman coding and range coding is somewhat different. Huffman coding can only capture data on a per-bit accuracy and it outputs only fixed-length bit strings. This is in stark contrast to range coding that allows sub-bit accuracy and as such provides better approximation of entropy which is typically non-integral.

Another interesting property of some of the compressors is that they model a Markov chain. Markov chains are present almost everywhere in sciences and as such finding a pattern of Markov chain using a compressor when computing NCD might prove valuable. This is a property which cannot be fulfilled by a dictionary encoder or a statistical encoder alone but needs a more advanced approach. PPM provides this capability but is generally considered slow. Fortunately LZMA provides similar functionality while being significantly faster.

As noted in the descriptions of the compressors, general purpose compressors take multiple approaches to compressing the data and stack these on top of each other. This allows the compressors to exploit multiple kinds of similarities within the data to compress it. However, the choice of these algorithms greatly affects the properties of the compressor and make them more suited to certain kinds of input data. However none of the compressors discussed here are specialized to one kind of data. Rather all of them exhibit different kind of compromises made between universality and performance with specialized data.

Cebrian et al. [7] have discussed the effect of the choice of compressor on normalized compression distance. They used files from the Calgary corpus and computed the normalized compression distance between strings and their perfect copies. This was done by taking first N bytes of individual files and computing their NCD. The N was varied and they were able to evaluate the behavior of their chosen compressors on various input lengths. They were able to uncover inflection points in the performance of the compressors. For gzip this was related to its look-ahead buffer and for bzip2 to its block size.

2.4 Processing large amounts of data

The workings of several different compression algorithms have been thoroughly discussed in the previous sections. It becomes apparent that if we are to compute the NCD for a lot of different samples using different compression programs, the task is going to be computationally demanding. In addition, the maturity and speed of compression algorithms vary a lot and for example PPM is very slow to compute when compared to the other general purpose compression programs. In this case the intent is to compute the NCD for a large number of different combinations of compressors, input data types and input data lengths. This defines the problem space and could be considered as an $N \times M \times K$ tensor where

the different combinations produce a result for their corresponding cell of the tensor. The model of what gets computed is presented in the next chapter.

One option of handling plenty of computationally intensive operations is to limit their amount. In practice this means that amount of computation could be limited by accepting less precision and therefore accepting a more coarse set of results. This could be attained by, for example, sampling the problem space in a representative way and only computing the results for those selected combinations of problem space.

In other words, it is apparent that computing all the different NCD values that are interesting is going to take a lot of time and the real problem becomes, how is it possible to compute the results so that there is no need to wait for results so long as it would take for one computer to process all the data. Luckily this is a very common problem in data sciences and there are multiple vendors that offer both commercial and non-commercial software solutions to support dividing up computational work among multiple computers in a network.

Tampere University of Technology operates a grid computing system that is based on a software system provided by Techila Technologies. The system works so that idle computers in classrooms and staff offices run computational jobs that are orchestrated by the Techila system. The basic operation of the system is presented in Figure 2.5 where the Techila system allocates work from its internal work queue to the individual computing nodes over the network. The computing nodes process the data and return the results back to the grid engine where the end-user can download it back to their workstation for further processing and analysis.[26]

The core solution to processing vast amounts of data using this kind of solution is to divide the problem space into single actionable computations. These computations can then be transformed into a queue which now contains all the computations that are required to be performed to attain the desired results. This simple transformation from problem space into a work queue is visualized in Figure 2.4. It is notable how a problem containing discrete variables is almost trivial to process like this as it just requires generating all desired combinations of the variables and enqueuing these these combinations into the work queue.

For example lets take the following Python pseudocode:

```
for i in range(0,100):
        for j in range(0,20):
                for k in range(0,5):
                        results[i][j][k] = i*j*k
```

In this case the individual computation results of the innermost for clause are all independent and can be computed in any order and without knowledge of the results of the other iterations. This also demonstrates how the problem space can be constructed in terms of variables. In this case the axis of the problem space are i, j, and k whereas the individual

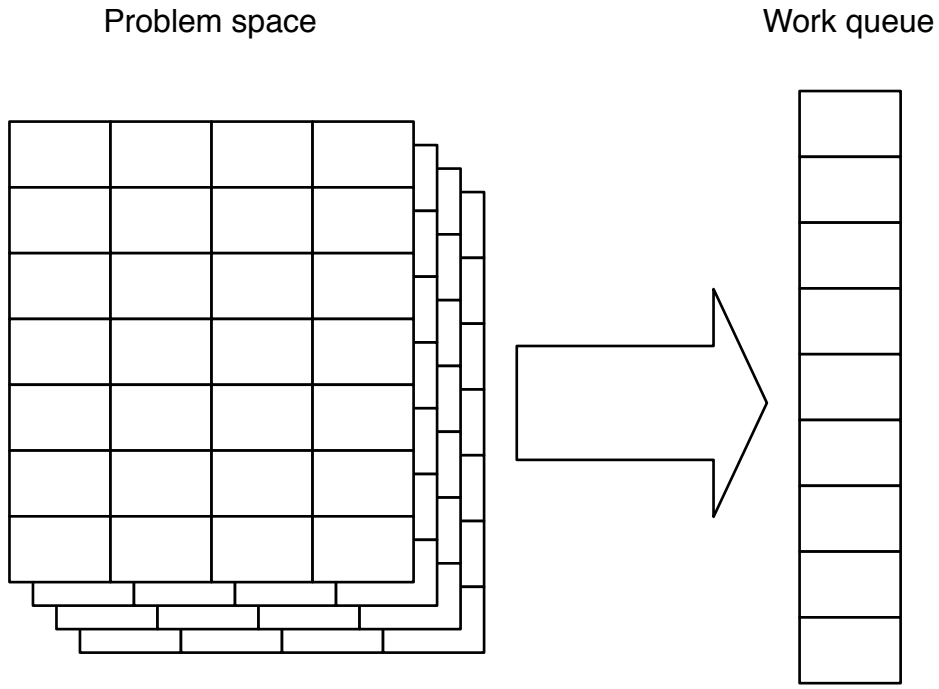

Figure 2.4: The $N \times M \times K$ problems space can be partitioned into discreet tensors that can be mapped to a work queue.

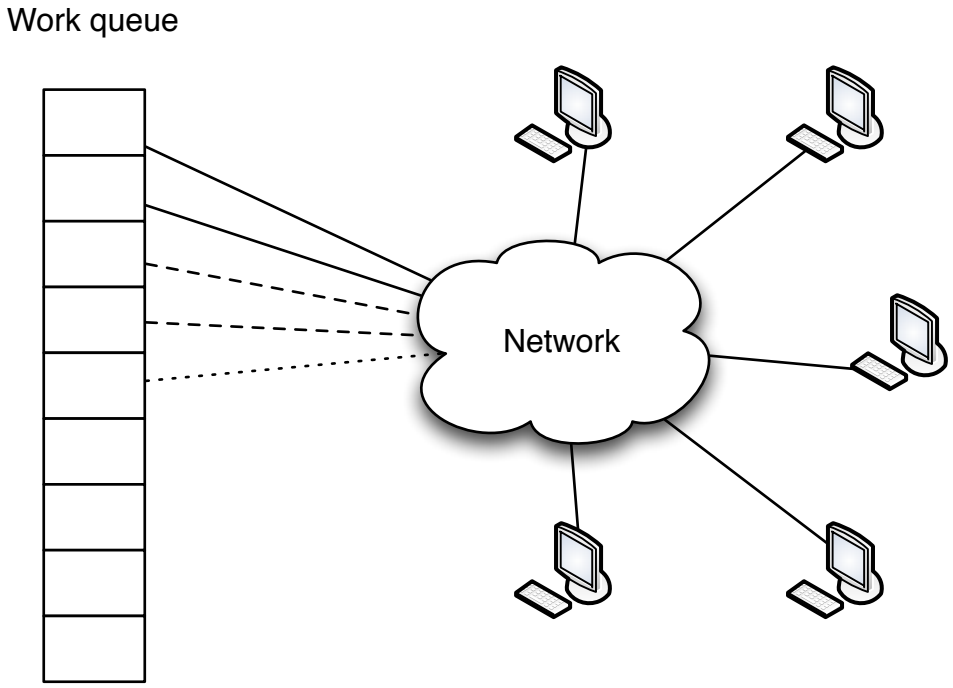

Figure 2.5: In the Techila system, a central work queue is stored on a server which then distributes the work among computers in a network. After processing the individual work items, the computers return results to the central server which then returns the results back to user.

computations of results are the cells of this discrete problem space. These computations can then be enqueued to the work queue as demonstrated next.

```
def compute(i, j, k):
        return i*j*k

for i in range(0, 100):
        for j in range(0, 20):
                for k in range(0, 5):
                        enqueue(compute, i, j, k)
```

This pseudocode listing assumes that the distributed computation framework allows individual functions to be queued up for execution. The actual Techila implementation provides similar but more generic enqueuing implementation in addition to an implementation which allows to directly parallelize for clauses without explicit enqueuing[26].

The work queue is then uploaded to a central server which distributes the computation among the actual computers that perform the computation. This process is illustrated in Figure 2.5. Once the computers finish the individual computations, the results are returned to the central server which in turn allows the original enqueueee to fetch the results.[26]

3. METHODS

The intent of this work is to evaluate the performance of different compressors when computing the normalized compression distance. To achieve this it is required to compute NCD using different compressors with different kinds of inputs. The selection of these inputs will determine how applicable the results will be for different kinds of data.

Therefore the main interest is to generate a cohesive framework for synthetically evaluating the performance of different compressors when computing NCD. The compressors are expected to yield different results for the same input data because of their different inner workings. A cohesive framework should allow us to uncover features with regard to the following features: length of the input, the process generating the input data, and the expected NCD of the data. This is because we want to tweak these parameters and see how different compressors behave under different circumstances.

Length of the input data for NCD is a fundamentally interesting feature of such a comparison. That is because many compressors contain features such as window sizes that might very well affect their performance with different lengths of input. A second interesting feature is the compressibility of concatenation of a random string with itself. This uncovers whether the compressor is able to encode the repeated symbols effectively. An input string generated by a Markov model is another interesting type of input. This is because many biological and stochastic processes generate measurement data that is comparable to the consecutive states of a Markov chain.

As mentioned earlier, one of the most interesting questions is, how is the performance of a given compressor when computing NCD between to inputs whose NCD or NID is somehow already known. This can be achieved by comparing a given input data string against a mutated copy of itself. For random input data, the most natural way mutating it is to introduce random mutations with probability p when p itself becomes an estimate of NCD. For a Markov chain model, a more natural type of mutation is to change its state to some other state with probability p. Again, the estimate for the NCD between the original string and its mutation is p.

The desired features of a comparison framework allow us to construct a simple exhibit of such a framework. It is possible to generate input data using fully random data or using a Markov chain model. This input can then be mutated with a tunable parameter p that is also its expected NCD. In addition, the length of the original input is easily tunable. This model makes it possible to test different compressors and tune the interesting parameters:

length of the input, type of process generating the input, and the expected NCD.

3.1 Modeling randomness and similarity

The previous section briefly discussed the need for an integrative framework for evaluating the performance of different compressors when computing NCD. The purpose of this section is to delve deeper into the details of the framework that was constructed as part of this work. We should now remind ourselves that NCD is a similarity metric. It follows that our evaluation framework should generate strings that are similar by a known amount so that we can compute NCD between those and be able to tell if the computed NCD is close to this a priori similarity estimate. In addition, the evaluation framework should be able to uncover characteristic behavior of the selected compressors and it should be possible to tell from the results how these behaviors map to the features used in the compression algorithms.

Any decent similarity metric should be able to recognize two identical strings as being very similar whereas two completely different random strings should be recognized as very dissimilar. To elaborate on this idea, given a string of bits it should be possible to compute its NCD with itself and get 0 whereas if given to completely random strings, their NCD should be 1.

This simplistic experiment covers the two major corner cases where the input is either two identical strings or two completely different strings. Yet the more interesting question is how does the computation work when given to strings that are not identical but yet not completely dissimilar. In addition, to test the performance of a compressor when computing NCD it should be possible to compare its output to some other estimate of similarity.

3.1.1 Processes generating input strings

A topic that has previously been discussed only briefly is the kinds of processes that might be used to produce the input data for our NCD comparisons. This means that it is expected that different kinds of compressors excel at compressing different kinds of input data because they contain different feature sets as presented earlier. Two different kinds of input were selected to be used in this work: a random input, and a Markov chain process.

Random strings were selected to be the first kind of input to be tested because they represent any random data that might be gotten, for example, by measuring something. In addition, compressing random data instead of repeated 0s or 1s is expected to produce more representative results because a string containing only recurring values might be considered trivial case for any compressor.

The second kind of model that was chosen for producing input data was a Markov chain. In this case the chain was chosen to be very simple. The chain contains discrete states so that the state can be encoded as an 8-bit integer on a computer. The chain runs

deterministically from one state to the next and the number of the state is output to the output string. The length of the chain can be adjusted so that behavior with different chain lengths can also be tested. This allows us to test whether the compressor can capture the essence of a Markov chain process.

3.1.2 Strings varying in length

It has been discussed already that the comparison framework should contain at least three interesting variables, namely: the length of input N, the type of the process generating the input, and a similarity estimate p. The first one of these, N, might be the most obvious one. Once we start computing the NCD of similar strings and let the length of the strings vary, it is expected that compressors based on block or buffer structure will stop noticing the similarities once the content is divided into multiple blocks or exceeds the buffer length. This happens when then compressor compresses the concatenation of the input strings and when processing the second string, it cannot anymore see the first one to use it as an aid in the compression.

Generating strings of varying length can be considered as easy. Most programming languages and frameworks contain a standard library that includes a random number generator that can be used to generate an infinite number of random numbers to fill up the string. If one of these random number generators is not made available by the programming language, operating systems provide similar functionality. As for the Markov chain model, it follows that the model can be run as long as required to produce the desired number of output states to fill the string with.

As discussed earlier, normalized compression distance is a similarity metric and if we are to evaluate compressors, we should have some kind of innate similarity metric against which to compare the calculated NCD values. This allows us to tell whether the result gotten by NCD is a good one or a bad one.

We can define this innate similarity metric as mutation probability p. In this model, p is the probability between 0 and 1 which tells us the likelihood that any single byte of the original input string is mutated when producing the copy that is used to compute the NCD against the original copy. However it follows that defining a natural way to mutate random strings and Markov chain generated strings is not the same.

In the case of random strings the most natural way of mutating the strings is to substitute a byte value for another. Now with p this can be achieved so that the original string is iterated byte by byte and at each step the current byte is replaced with a random one with given probability p. This keeps both the original and the mutated copy completely random and does not introduce any algorithmic features in the string in the sense that all the mutations are completely random.

This model tries to be the simplest possible model that allows to generate multiple strings whose similarity can be approximated so that the results of NCD can be compared

against this estimate. In addition, this model allows to compare the performance of different compressors when there is no algorithmic model behind the mutated data as both the original string and its mutated version are completely random if not accounted for the probability of mutations. In addition, introducing any more sophisticated mutations would be adding a signal to the inherit noise of the random data.

As for the Markov chain model, introducing random mutations in the data like previously described would be like adding noise to a signal. The string generated by the Markov model inherently contains data and not noise like the one generated by taking random numbers. Instead the easiest way to mutate the Markov data is to keep the Markov model but to tweak it so that instead of just running from one state to another, it might also jump to another random state with probability p. This keeps the algorithmic nature of the string generated by the Markov model but introduces variance that is still algorithmic in nature in stead of just being noise.

3.1.3 Markov Chain Model

The random string model that was described in the previous section can be considered a really simple one and it is unlikely that any real world measurement data would exhibit that kind of dissimilarity unless the only source of difference is noise in the measurement data. Therefore it becomes apparent that a more complicated model might prove more usable when trying to simulate real-world measurement conditions. A common pattern that keeps on repeating in natural processes is the Markov chain or the Markov process.

The second model of generating test data for evaluating normalized compression distance is therefore based on Markov chains. A Markov chain can exhibit more complex behavior than a simple random string mutation model. However, in the context of this work, the aim is to keep the model simple and therefore a deterministic Markov chain is used to produce repeating input data. This model just moves from one state to the next and the current state is encoded as an eight-bit integer. A mutated copy is introduced to facilitate the computation of NCD so that a chain of same length as the original is generated with one modification. The modification is that at each step the chain can jump to a random valid state with probability p. Given that also the number of states, that is the length of the chain, can be modified, this gives us a model relatively close to the random string model.

This Markov chain model is considered to be interesting in the sense that many natural processes exhibit Markov chain like properties. For example a dynamic system running around an attractor produces an output similar to this. In addition, the mutation can be considered as an external perturbation affecting the system and moving it into another state.

To sum up, the second model generates a deterministic Markov chain of M states where the state changes in order from state x to state $x + 1$ and from the last state to the first state.

N kilobytes of the output states are encoded as 8-bit integers. A mutation probability p is introduced to this model so that the chain changes its state to a random valid state with uniform probability p at each step.

3.2 Utilizing the model

The two models described in this chapter, namely the random string model and the Markov chain model, can be used to evaluate the performance of compressors when computing the normalized compression distance. This can be done by varying the parameters of all models so that it becomes possible to see how these parameters affect the results of computing the normalized compression distance.

In the case of random string model, the length parameter N allows directly to test how compressors perform at different input lengths. However in the case of the Markov chain model, the length of the chain M does not directly map into the length of the input but rather describes how long matches can be found in the input data. The mutation probability p, on the other hand, measures how robust the compressor is to noise and perturbations.

The concept of normal compressor was introduced earlier to describe which kinds of features make a compressor perform well when computing normalized compression distance. To recap, the features of normal compressor C are[8]:

1. Idempotency: $C(xx) = C(x)$, and $C(\lambda) = 0$ for an empty string λ.
2. Monotonicity: $C(xy) \geq C(x)$.
3. Symmetry: $C(xy) = C(yx)$.
4. Distributivity: $C(xy) + C(z) \leq C(xz) + C(yz)$.

Only the first feature of normal compressor, idempotency, is directly measured by the mutated random strings model. When $p = 0$ the model directly measures the first property of idempotency. However if we interpret the concatenation xy to mean information conveyed by x in addition to some more information y, and the mutation probability p to convey additional information in both models, then both of the models measure monotonicity indirectly.

4. RESULTS

4.1 Running on the grid

The different models for estimating NCD were run using Techila Grid Engine on TUT network. Techila uses idle desktop computers to perform its computations. This section discusses how Techila was used in this context and how the computations were performed.

Techila was used so that a Java application was prepared that both interacts with the Techila grid to queue all the work and actually performs all the distributed computation when run within Techila. After the enqueueing of work, Techila grid performs the actual work without any user input requirements. After all computations have been finished, Techila returns the results back to the original calling program that enqueued the work.

The packaging and enqueueing of the execution is performed on the user's workstation. The Java application implements the actual compression process by using Java libraries for gzip, PPM and bzip2 whereas native executables are called for LZMA. The Java application then calls Techila's APIs to enqueue our work for Techila to execute. This causes Techila's API to package both our Java application and its binary dependencies and upload them to the Techila Grid Engine running on its own server.

When enqueueing, the Java application generates the desired parameter space of length, p and compressor for our calculations as discussed in the previous chapter. The different combinations of the parameters are enumerated and work is enqueued to Techila so that all combinations are run.

Techila Grid Engine performs the actual computation independently using available computing capacity around the campus. At the core of the grid engine is a work queue where individual computation items reside. Techila allocates work from this queue to the computation nodes which perform the actual computation. Finally the result of the computation is returned to the Techila Grid Engine which stores the results for further use.

Finally Techila notifies the calling program that the results are available and that computation has been completed. At this point the client program fetches the computation results back to user's computer. Now that the results are available locally in the memory, the Java client program writes them onto the disk in a text format so that they can be further analyzed. Afterwards the data was loaded into Matlab for further processing and visualization.

4.2 Measurement setup

The parameters for running the comparison framework were selected based on practical limits of computing power and what was deemed as reasonable. This affected the choice of parameters so that the maximum amount for variables such as N were kept reasonably small but the power of the computing grid was harnessed to gain good granularity.

The choice of compressors was done so that Java SE's internal implementation of gzip was used whereas Apache's Commons Compress [1] implementation was used for bzip2. Colloquial.com's arithmetic coding package for Java was used as the PPM compressor [2]. These were chosen mainly because they were readily available for Java and did not require any extra binaries to be bundled with the computation code.The XZ Utils implementation of LZMA was used [3]. This was because while there was a Java implementation available, its compression performance was terrible when compared to the native one and as such it produced virtually unusable results.

All compressors were used with their default settings except for PPM which requires an order parameter. The order for PMM was set to 8 which makes it a PPM(8) compressor. The choices were made because compression programs are expected to come out of the box with reasonable defaults. In addition, the choice of PPM(8) was done because it was the highest order that was possible to compute in a reasonable time.

The random string model was run so that the inputs would capture the buffer lengths of gzip and bzip2 whereas PPM and LZMA do not feature any such limitations. So the input length N was varied from 1 kilobyte to 1024 kilobytes with increments of 1 kilobyte which means that the concatenation of the string with itself, or the mutated copy, varied from 2 kilobytes to 2048 kilobytes. This should make it possible to see the effects of gzip's look-ahead buffer of 32 kilobytes and bzip2's block size which is 900 kilobytes by default.

In the case of the random string model, the mutation probability p was run from 0 to 1 with increments of 0.01. This was done mainly because available processing power permitted such granularity. This should also allow us to see if there is an inclination point at which the NCD performance of given compressors starts to decline.

The parametrization of the Markov chain model was done differently. This is mainly because there is one more free parameter available, namely the length of the Markov chain. The length of the chain was varied from 1 to 256 so that all states could be encoded as 8-bit, or one byte, integers. A string worth of 10 kilobytes was created by running the chain from state to state and encoding the ordinal of the state as integer. The length of 10 kilobytes was chosen mainly because it is long enough allow the output of the chain to repeat multiple times but still fits within gzip's lookahead buffer.

The mutation probability p of the Markov chain model was varied from 0 to 1 with increments of 0.01. This was done to generate strings that range all the way from identical to fully different.

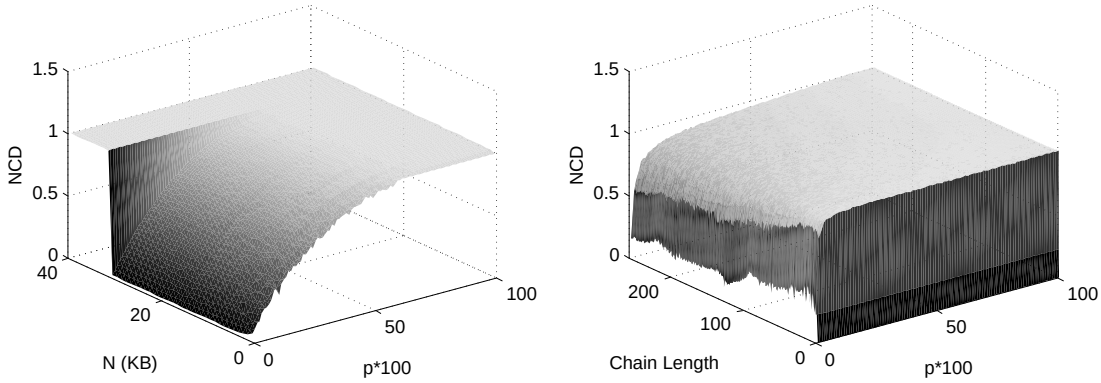

Figure 4.1: The individual test run results for gzip. Picture on the left contains the results from random string model and the picture of the right contains the results for the Markov chain model.

4.3 Results and analysis

The purpose of this section is to present the results that were obtained using the grid computing setup described earlier. The results and analysis are going to be present first on a per compressor basis for both the random string model and the Markov chain model. Afterwards the compressors are compared to each other so that a reasonable recommendation can be done on what compressor to use and on what conditions.

4.3.1 gzip

The results of the test runs for gzip are presented in Figure 4.1. These figures were generated by loading the raw measurement data from the grid application into Matlab.

The results show that with the random input model, gzip performs reasonably well as long as the length of the input remains under the size of the lookahead buffer that is 32 kilobytes. Once the length of the input reaches the aforementioned 32 kilobytes, the NCD between the original and mutated string hits 1 irrespective of p. This result is in line with the results in the earlier work of Cebrian et al.[7]. The usable range of gzip in computing NCD therefore appears to be for content shorter than 32 kilobytes.

When dealing with inputs of shorter length than 32 kilobytes, gzip performs reasonably well. The NCD of identical strings is zero and it grows almost monotonically until p reaches 0.5. This gives a reasonable expectation that gzip works well for computing NCD when the length of input is small and the data is known to be quite similar already.

The Markov chain model appears to break gzip with a lot smaller p values than the random model. It is apparent from Figure 4.1 that with short chain lengths the NCD reaches 1 almost immediately. The performance considerably improves when the length of the chain grows. This is likely because the LZ77 algorithm within gzip can recognize longer repeating runs of the chain when such exist and the overall compression ratio improves.

The relation between p and NCD in the Markov chain case is more complex. Once p

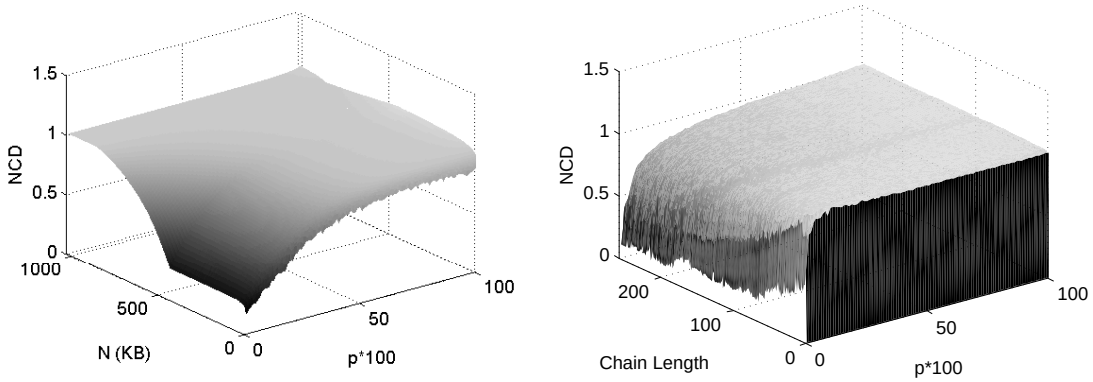

Figure 4.2: The individual test run results for bzip2. Picture on the left contains the results from random string model and the picture of the right contains the results for the Markov chain model.

starts growing, the NCD hits the range of around 0.8 - 0.9 quite rapidly but the growth of NCD seems to stay monotonic even after this. This is interesting because one might expect that there would be some inflection point after which the NCD values would become meaningless. However I would not call the NCD values obtained at the plateau of monotonic growth near one statistically significant.

Therefore it appears that gzip is a usable tool for computing NCD when the length of input is less than 32 kilobytes. In addition, it should be known beforehand that the strings being compared are reasonably similar. This is because in the random model the ability compute usable NCD values stops around $p = 0.5$.

4.3.2 bzip2

The results for bzip2 are presented in Figure 4.2. The figure shows bzip2's performance with the random string model whereas the right one present the performance in the case of the Markov chain model.

Taking a look at the leftmost edge of the the first picture, it is apparent that bzip2's performance here is very different from gzip's performance in the previous section. First it should be noted that the axis have been changed to show data all the way up to 1024 kilobytes. In addition, the profile of the bzip2 curve is very different than gzip's. Where gzip's performance hit a clear and immediate wall at input lengths of 32 kilobytes, bzip2's performance is flat all the way up to 450 kilobytes and after that starts to gradually deteriorate before hitting NCD $= 1$ at 900 kilobytes.

This follows quite clearly from bzip2's default block size of 900 kilobytes. Once the length of the input reaches 450 kilobytes, the length of the concatenation reaches 900 kilobytes. After this part of the concatenation does not fit inside the block anymore and starts overflowing to the next block and once the length of input reaches 900 kilobytes, the whole second part of the concatenation is pushed to the next block and the compressor cannot anymore use information from the first occurance of the string to compress the

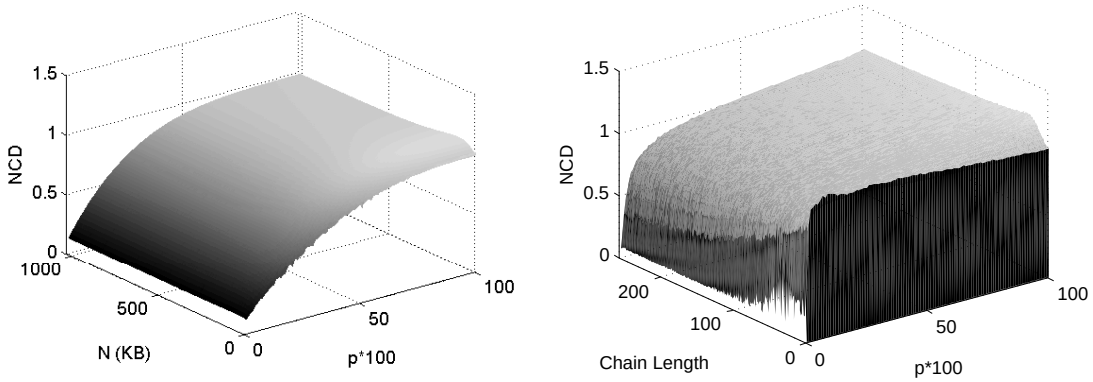

Figure 4.3: The individual test run results for PPM(8). Picture on the left contains the results from random string model and the picture of the right contains the results for the Markov chain model.

second one.

On the p axis bzip2 follows same kind of pattern as gzip. The biggest difference is that bzip2 does not yield NCD of 0 at $p = 0$ but rather the values start around 0.2. The NCD value grows together with p until hitting value around 1 at around $p = 0.5$.

The behavior of bzip2 is also similar to gzip in the case of the Markov chain model. At chain length 0 and $p = 0$ the NCD value is around one and gradually improves to lower values as the chain length increases. The NCD values of bzip2 appear to be uniformly lower than those of gzip's. In addition, with greater chain lengths it appears that usable range of NCD, that is before hitting the plateau, extends further into p as with gzip.

To sum up, bzip2 appears to be useful for content up to the length of 450 kilobytes after which the performance starts to degrade. In addition bzip2 is more powerful than gzip when computing NCD values for the Markov chain model. This demonstrates that bzip2 is able to catch a more broad array of source features for compression purposes.

4.3.3 PPM

The results of the test runs for PPM are presented in Figure 4.3. The figure shows PPM's performance with the random string model whereas the right one presents the performance in the case of the Markov chain model. These results are for the aforementioned PPM(8) compressor.

The random string model results of PPM deviate from the baseline set by gzip and bzip2. PPM has no block size so the length of input has no effect on NCD. Therefore the illustration in Figure 4.3 shows that even strings as long as 1024 kilobytes can be processed and the computed NCD value remains fully usable.

On the p axis PPM shows behavior similar to gzip and bzip2. That is the NCD increases more or less monotonically with increasing p until it reaches a plateau around 0.5. As for the Markov chain model, PPM again shows the same kind of behavior as gzip and bzip2. However it is notable that PPM produces NCD values of over 1 given short chain length

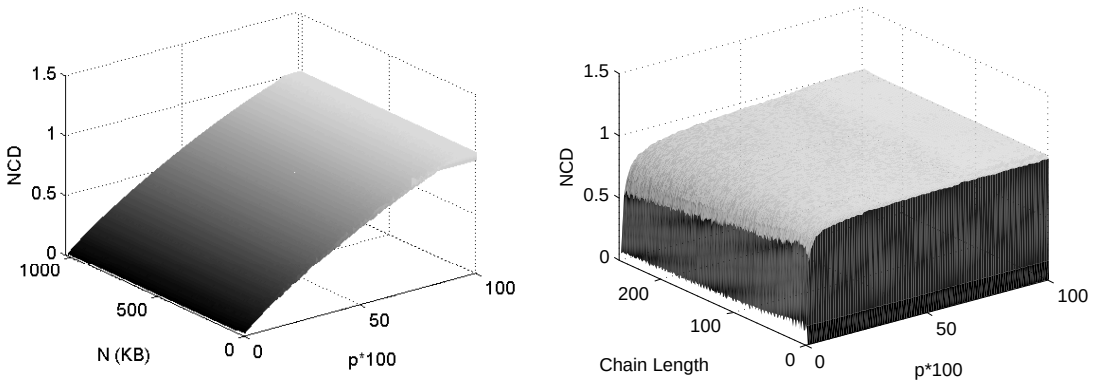

Figure 4.4: The individual test run results for LZMA. Picture on the left contains the results from random string model and the picture of the right contains the results for the Markov chain model.

and large p.

4.3.4 LZMA

LZMA results are presented in Figure 4.4. The random model is presented on the left and the Markov chain model on the right.

At this point, the results shown in the figure do not contain any major surprises when compared to the other compressors. In case of the random string model, the results relating to string length are similar to those of PPM. That is LZMA does not exhibit any behavior which would relate to an algorithm based on block or other buffers. A more interesting behavior is relating to the p axis. NCD computed using LZMA, increases almost linearly along p until around $p = 0.9$. This is a lot more than with the other compressors that basically reach their saturation point of NCD 1 around $p = 0.5$.

In the case of the Markov chain model, LZMA yields results very similar to those of other compressors.

4.3.5 Comparing compressors

Now that the individual results of each compressor have been presented, it becomes easier to cross-compare their performance. The most interesting comparison is naturally, how the different compressors compare when given the same type of input data. In addition, an interesting question is how do the different compressors fare with the different classes of input data such as the fully random input and the Markov chain model input.

The performance of different compressors is pictured in Figure 4.5 in relation to the random input data model. In case of gzip, the effect of its look-ahead buffer is clearly visible and it stop working once N hits 32 kilobytes. In case of bzip2, a similar kind of effect can be seen relating to block size and the quality of the output starts to deteriorate once the length of input hits 450 kilobytes and the length of the concatenation starts overflowing the

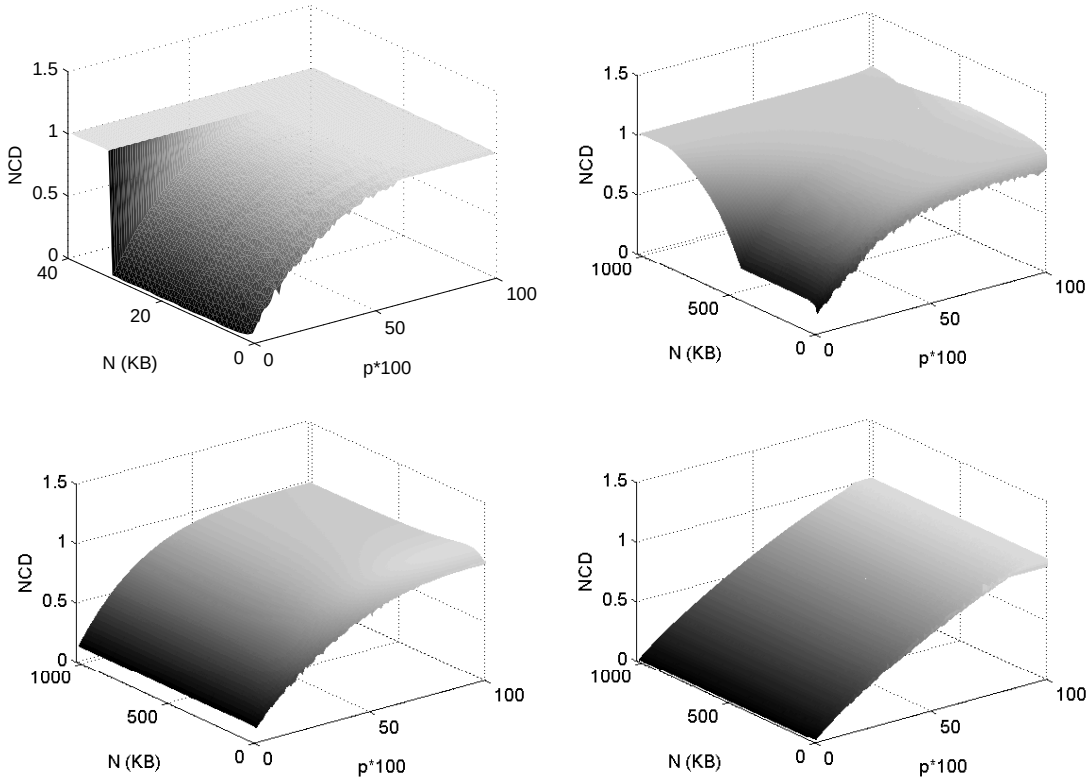

Figure 4.5: The results of the random input model data for gzip, bzip2, PPM, and LZMA presented from left to right and top to bottom. In case of gzip and bzip2, the effect of look-ahead buffer and block size can be seen in the performance related to the N axis. In case of PPM and LZMA, the effect of N is limited and the results are mostly affected by the p axis. Note that gzip is drawn with a different scaling of the Y-axis than the others. Each value in the figure corresponds to a single output value from NCD i.e. no averaging has been done.

default block size of 900 kilobytes. PPM and LZMA do not show a similar dependence on the length of the input at the kinds of lengths used in these tests.

All different compressors show similar kinds of performance in relation to the randomness parameter p. The compressors provide a reasonable NCD estimate when p is small and the computed NCD increases with p on all compressors. The major difference between compressors is that the given NCD value stops being usable at different values of p. LZMA seems to provide most usable values in this case as the increase of NCD is almost linear in relation to p up to values of p around 0.9.

The Markov chain model results for all the compressors are displayed in Figure 4.6. It can be seen that all the compressors produce a similar kind of surface but yet there are small differences. The common characteristics are that with small chain lengths all compressors have difficulties in computing the NCD and the performance improves when the length of the chain increases. In addition, all compressor work better with small values of mutation probability p and the performance decreases quite rapidly once p increases.

Individual cross cuts for the Markov chain model are presented in Figure 4.7. Each one

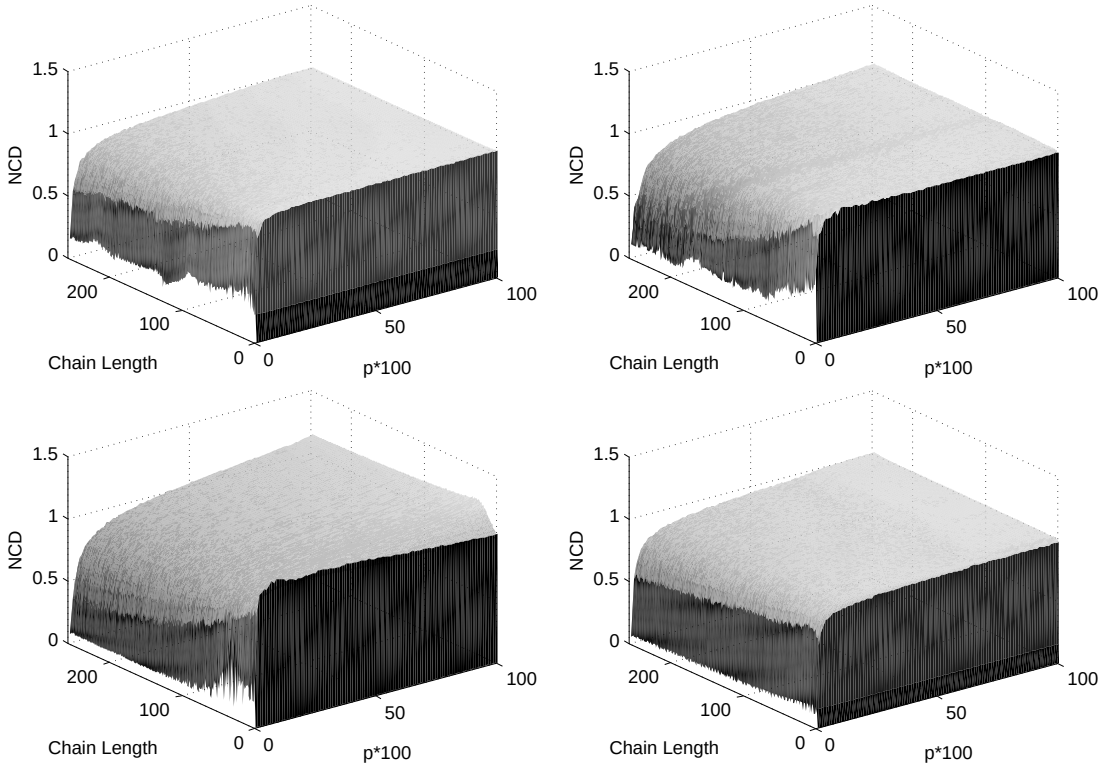

Figure 4.6: The NCD of a deterministic Markov chain compared against a Markov chain with random mutations with probability p as explained in section 3. The compressors are gzip, bzip2, PPM(8) and LZMA (from left to right, top to bottom). The length of the data strings was 10 KB and the state data was encoded as 8-bit integers.

of the cross cuts represents a different chain length so that it can be seen how both the chain length and mutation probability p affect the results. These values have been scaled so that the maximum NCD value of each compressor is normalized to one. This is because some compressors produce NCD values of over one for some inputs. The raw unscaled NCD values corresponding to these results can be seen in Table 4.1.

It could be argued that the compressor that provides the most linear dependency between p and the NCD, and provides the lowest NCD values till the furthest, could be considered the best. However all the compressors produce such results that the p axis of the figure has been scaled to logarithmic scale.

It can be seen in the figure that the order of the compressors changes at different lengths of the chain. For example bzip2 begins as one of the worst at chain length of 32 whereas it keeps on improving together with the increasing chain length and at longer length it becomes almost as good as PPM. PPM, on the other hand, always produces the lowest NCD values irrespective of the chain length. However even PPM's performance improves with the increasing chain length.

To sum up, PPM provides the best and most consistent results in all these cases. Similar strings have a small NCD and as the amount of perturbations in the model increases, the NCD can be seen to increase proportionally more than with other compressors. The per-

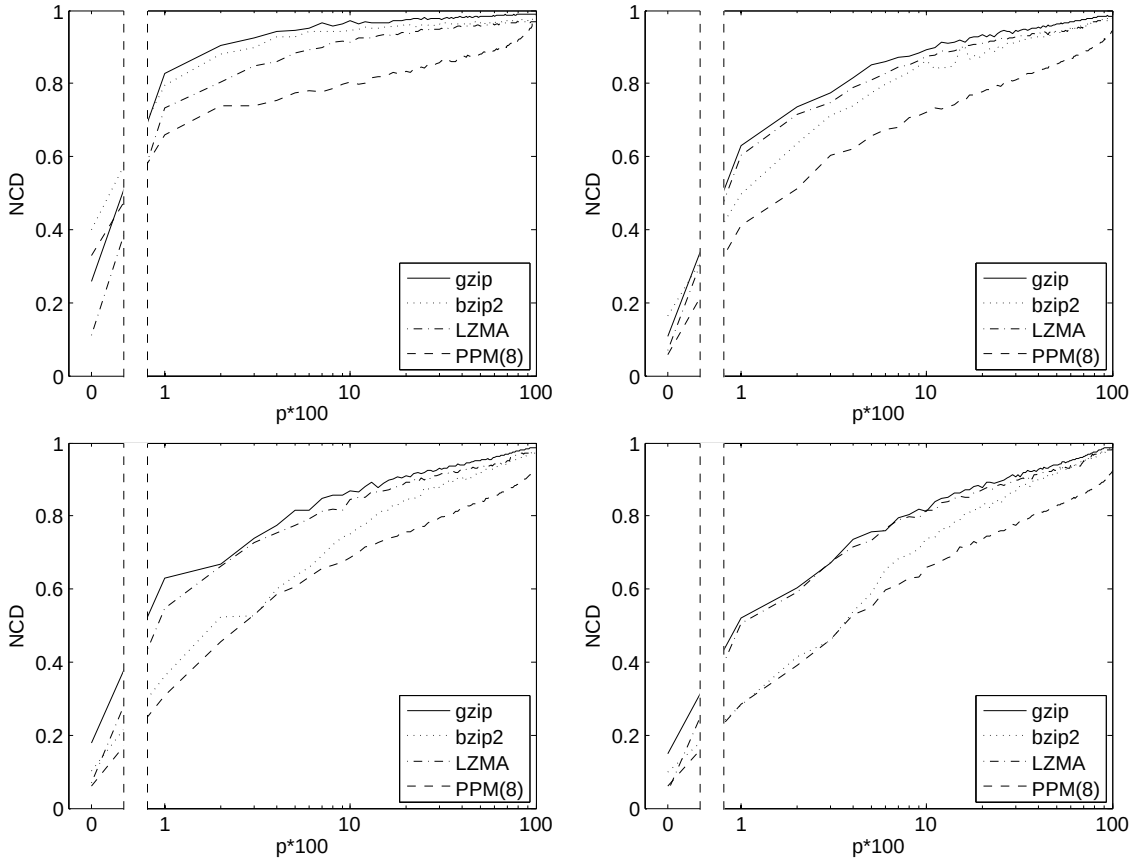

Figure 4.7: The NCD for Markov chains of four different lengths N (32, 128, 192, 256, from left to right, top to bottom). The values have been scaled by dividing with a compressor-dependent constant which was obtained by finding the global maximum of the NCD for each compressor in these calculations.

formance of bzip2 greatly increases with longer chain lengths. Bzip2's good performance for longer chain lengths is also noticeable. LZMA's performance seems to be similar to gzip's in this case even though LZMA fares much better than gzip in the first case. Given that both PPM and LZMA use context modeling, their good performance is not surprising.

A strong connection between bzip2's default block size of 900 kilobytes and the length of the input can be seen. When the length of the input x exceeds 450 kilobytes and thus the length of the concatenated input xy exceeds 900 kilobytes, the performance starts to degrade rapidly. Gzip, in turn, stops working once the length of the input exceeds 32 KB. On the other hand, PPM and LZMA appear to work properly for all input lengths up to 1024 kilobytes which wast the maximum length of input in these tests.

Another way to compare the performance of the compressors is in relation to the normal compressor C that was introduced in Section 2.2. The features of a normal compressor are idempotency, monotonicity, symmetry, and distributivity. The comparison framework that was introduced in Section 3 directly measures idempotency, that is $C(xx) = C(x)$, and $C(\lambda) = 0$ given λ is an empty string, when the mutation probability p is zero for the first case, and the second case is approximated by the case when the length of the input is

Chain length		$\text{NCD}_{p=0}$	$\text{NCD}_{p=1}$
32	gzip	0.26	0.99
	bzip2	0.42	1.02
	PPM(8)	0.41	1.17
	LZMA	0.11	0.99
128	gzip	0.11	0.99
	bzip2	0.17	1.02
	PPM(8)	0.07	1.17
	LZMA	0.07	0.99
192	gzip	0.18	0.99
	bzip2	0.11	1.02
	PPM(8)	0.07	1.15
	LZMA	0.07	0.99
256	gzip	0.15	0.99
	bzip2	0.10	1.03
	PPM(8)	0.07	1.14
	LZMA	0.04	1.00

Table 4.1: Statistics for the NCD that was computed for the Markov chain model of different chain lengths. $\text{NCD}_{p=p^*}$ is the NCD value with $p = p^*$.

one kilobyte. The first case is because when $p = 0$ and $C(xx) = C(x)$ the normalized compression distance becomes

$$\begin{aligned}\text{NCD}_{p=0}(x, x) &= \frac{C(xx) - \min\{C(x), C(x)\}}{\max\{C(x), C(x)\}} \\ &= \frac{C(x) - C(x)}{C(x)} = 0.\end{aligned}$$

Given the numbers in Table 4.1 and the results in Figure 4.5, none of the compressors fully achieves the second feature. This is not surprising given that compressors are generally not optimized for compressing empty strings and the compressed file formats introduce overhead such as dictionaries. The first case is more interesting because it is plausible that real-world compressors achieve $C(xx) = C(x)$ at least under some constraints.

The results for the first case can be deducted from Figure 4.5. All of the evaluated compressors achieved NCD values of well below one for the $p = 0$ case within their limitations of applicability. LZMA and gzip fare the best in this case given that their NCD is very close to zero whereas PPM produces a little bit higher value and bzip2 considerably higher value but still clearly less than one.

The testing setup can also be used to infer properties relating to monotonicity $C(xy) \geq C(x)$ if we interpret the concatenation xy to mean "more information added to x". In this case our mutation model can be interpreted to add some amount of additional information in relation to p to the original string. Computing the NCD in these cases does not directly measure the property of monotonicity but $C(xy)$ should increase with p while $C(x)$ and

$C(y)$ should stay constant so NCD should grow with p. In the experiments, NCD exhibited this kind of behavior for all compressors within their applicable range.

None of the performed tests directly measures symmetry or distributivity. Out of these two, distributivity appears less interesting in computing NCD between two strings. On the other hand symmetry is a more interesting feature since there seldom is a natural order for the kind of measurement data that produces strings that could be used with NCD.

4.4 Discussion

The result that were previously discussed show that there are indeed differences between the different compressor in the context of our models. In addition, the tests uncovered compressor features such as block size and look-ahead buffer which directly affect the usability of some compressors once certain conditions in relation to input are met.

Cebrian et al.[7] have evaluated the performance of gzip and bzip2 when NCD is computed between two identical strings of bytes taken from the Calgary Corpus collection of texts. Their results match the results presented in this work. However the context of this work extends beyond what was studied earlier. Our results for bzip2 and gzip basically duplicate their work which relate to cases where the random string model was used and the mutation probability p was small.

The main research question of this work is whether it would be possible to uncover differences in compressors relating to their performance in computing normalized compression distance and whether it would be possible to give general recommendations for the choice of compressor. It was indeed possible to uncover differences in compressors based on their internal algorithms. This was most dramatically demonstrated by effects of block size and look-ahead buffer in the case of bzip2 and gzip respectively. Smaller differences were found in how large a mutation probability p the compressors tolerate in the random string model and how linearly NCD follows p. However in the case of the Markov chain model, the differences were less significant between different compressors.

This work produced new knowledge especially in relation to the earlier work by Cebrian and others[7] mainly because the random string model can viewed as an extension of their work where the mutation probability p is introduced, more compressors are evaluated, and more granular datapoints are computed. In addition, this appears to be the first work where informed recommendations about the choice of general purpose compressors for computing the normalized computing distance can be given.

However the current evaluation framework does not fully incorporate all features of normal compressor. The model presented in this work covers the first two features of normal compressor, idempotency and monotonicity. The remaining features of normal compressor remain as an elusive research subject as normalized compression distance is a metric only when the used compressor is a normal compressor[8]. Symmetry is the more natural property of the two remaining features of normal compressor, namely symmetry

and distributivity, because measurement data or other kinds of data seldom have a natural ordering. In addition, adaptive compressors may exhibit behavior where their performance is not symmetric due to the adaption. Distributivity, however, is intuitively not such an interesting feature as the others even though it is one of the four properties of normal compressor.

It should be noted that none of the compressors worked especially well with the Markov chain model. This reveals that normalized compression distance is not as universally applicable as could be inferred from its theoretical basis in normalized information distance. So whenever using normalized compression distance in real-world situations, it would be better to cross-check the results with some other method to validate them.

5. CONCLUSIONS

Normalized compression distance is a universal similarity metric. This means that it can be used to cluster and classify many different kinds of measurement data without having knowledge about the feature based similarity of the samples. However the functionality of normalized compression distance depends on a solely theoretical construct called normal compressor which does not exist in real life and therefore normal compressor is substituted for general purpose compressor programs in general usage. This introduces problems in relation to features of the chosen compressor because compressors do have features, such as block size, that hinder the computation of normalized compression distance with certain kinds of inputs.

The purpose of this work was to create a cohesive framework than can be used to evaluate different compressor programs when computing the normalized compression distance. To facilitate this, the framework contains two different data generation models with tunable parameters. The first one generates random strings of given length and normalized compression distance is then calculated against a mutated copy of the original string. In this model it is possible to adjust the length of the input string and the mutation probability within the mutated copy of the string. The second model contains a deterministic Markov chain whose consecutive states are encoded as a string. A second string is generated for comparison so that at each step the chain might get mutated to some other random state with a given probability. The length parameter was used to control the length of the Markov chain in this case. Together these models also make it possible to indirectly observe how the compressors fulfill the first and second property of normal compressor, namely idempotency and monotonicity.

This framework was used to evaluate the performance of gzip, bzip2, PPM, and LZMA compressors with different values of the mutation probability and input length. This was achieved by running the individual computations on a Techila Grid Engine system used at Tampere University of Technology. These results clearly show that gzip can effectively only compute normalized compression distance for input lengths of less than 32 kilobytes. This is due to gzip's internal look-ahead buffer. The bzip2 compressor has a similar similar problem in relation to its block size which is 900 kilobytes by default. PPM and LZMA, on the other hand, seem to be able to handle any input lengths within the tested range. However PPM was slowest of the tested compressors whereas LZMA proved to be way more faster than PPM. In addition, LZMA provided the widest usable range in relation to the

mutation probability and its NCD values were most linear in this range. Therefore LZMA should be the default choice of compressor when computing the normalized compression distance without any knowledge of the input's feature based similarities.

A further extension of this research would be to extend the evaluation framework to contain the other two features of normal compressor, symmetry and distributivity. In addition, it appears that no research has been performed to date on the usage of lossy compressors when computing the normalized compression distance. This might yield interesting new results about when to use a lossy approach and when to use a lossless approach.

REFERENCES

[1] Apache commons compress. http://commons.apache.org/compress/.

[2] Compression via arithmetic coding in java. version 1.1. http://www.colloquial.com/ArithmeticCoding/.

[3] Xz utils. http://tukaani.org/xz/.

[4] J.L. Bentley, D.D. Sleator, R.E. Tarjan, and V.K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[5] S. Bornholdt. Systems biology: less is more in modeling large genetic networks. *Science*, 310(5747):449–451, 2005.

[6] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[7] M. Cebrian, M. Alfonseca, and A. Ortega. Common pitfalls using the normalized compression distance: What to watch out for in a compressor. *Communications in Information & Systems*, 5(4):367–384, 2005.

[8] R. Cilibrasi and P.M.B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[9] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on*, 32(4):396–402, 1984.

[10] T.M. Cover, J.A. Thomas, J. Wiley, et al. *Elements of information theory, 2nd edition*. Wiley-Interscience, 2006.

[11] P. Deutsch. Rfc1952: Gzip file format specification version 4.3. *Internet RFCs*, 1996.

[12] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification*. John Wiley & Sons, 2 edition, 2006.

[13] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[14] M. Kanehisa and S. Goto. Kegg: kyoto encyclopedia of genes and genomes. *Nucleic acids research*, 28(1):27–30, 2000.

[15] A.N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of information transmission*, 1(1):1–7, 1965.

[16] G.N.N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video & Data Recording Conference, Southampton*, 1979.

[17] A. Moffat. Implementing the ppm data compression scheme. *Communications, IEEE Transactions on*, 38(11):1917–1921, 1990.

[18] M.E.J. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[19] M. Nykter, N.D. Price, M. Aldana, S.A. Ramsey, S.A. Kauffman, L.E. Hood, O. Yli-Harja, and I. Shmulevich. Gene expression dynamics in the macrophage exhibit criticality. *Proceedings of the National Academy of Sciences*, 105(6):1897, 2008.

[20] M. Nykter, N.D. Price, A. Larjo, T. Aho, S.A. Kauffman, O. Yli-Harja, and I. Shmulevich. Critical networks exhibit maximal information diversity in structure-dynamics relationships. *Physical Review Letters*, 100(5):058702, Feb 2008.

[21] J. Rissanen and G.G. Langdon. Arithmetic coding. *IBM Journal of research and development*, 23(2):149–162, 1979.

[22] J. Seward. bzip2 source code. http://www.bzip.org/, 1996–2010.

[23] J. Seward. bzip2 web page. http://www.bzip.org/, 1996–2010.

[24] C.E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[25] N.N. Taleb. Beware the big errors of 'big data'. http://www.wired.com/opinion/2013/02/big-data-means-big-errors-people/, 2013. Online; accessed 28-September-2013].

[26] Techila Technologies. Techila fundamentals. http://www.techila.fi/wp-content/uploads/2013/08/Techila-Fundamentals.pdf, 2013.

[27] Wikipedia. Burrows–wheeler transform — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Burrows%E2%80%93Wheeler_transform&oldid=517699361, 2012. [Online; accessed 24-October-2012].

[28] Wikipedia. Lempel–ziv–markov chain algorithm — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Lempel%E2%80%93Ziv%E2%80%93Markov_chain_algorithm&oldid=565248892, 2013. [Online; accessed 28-September-2013].

[29] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.

[30] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.