Università di Pisa
Dipartimento di informatica
Ph.D. Thesis

Compression Techniques for Large Graphs: Theory and Practice

Luca Versari

Supervisor
Roberto Grossi
Università di Pisa

Referee
Travis Gagie
Dalhousie University

Referee
Daniel Lemire
Université TÉLUQ

Referee
Kijung Shin
KAIST

ABSTRACT

In today's world, compression is a fundamental technique to let our computers deal in an efficient manner with the massive amount of available information. Graphs, and in particular web and social graphs, have been growing exponentially larger in the last years, increasing the necessity of having efficient compressed representations for them. In this thesis, we study the compression of graphs from both a theoretical and a practical point of view. We provide a new technique to achieve better compression of sequences of large integers, showing its theoretical and practical properties, as well as new techniques for practical context modeling in large context spaces. We conduct a theoretical analysis of the compression of various models of graphs, showing that theoretically optimal compression for each of those models can be achieved in polynomial time. Finally, we show how to apply the proposed techniques to practical graph compression to obtain a new scheme that achieves significant size savings over the state of the art, while still allowing efficient compression and decompression algorithms.

Acknowledgements

During the years of my PhD, I worked with many fantastic people, both in the University of Pisa and at Google, and I enjoyed the support of my family and many other people whom I did not work with. Thanks to everyone for their friendship, the interesting discussions, and the fun moments! Our interactions helped making my life a bit better every day.

Among those people, special thanks go to those who helped me improve this thesis; in no particular order, Alessio, Roberto, Thomas, Jyrki, Iulia, Robert, Jon.

I also wish to thank the reviewers for their useful comments and feedback.

CONTENTS

Contents 3

1 Introduction 7
1.1 First part: general-purpose compression . . . 8
1.2 Second part: graph compression . . . 9
1.3 Notation and definitions . . . 9
1.4 Published material . . . 11

I General-purpose compression 13

2 Compression concepts and techniques 15
2.1 Entropy of a random variable . . . 15
2.2 Entropy of a text . . . 17
2.3 Integer coding . . . 18
2.3.1 Unary coding . . . 18
2.3.2 Rice coding . . . 19
2.3.3 Elias γ coding . . . 19
2.3.4 Elias δ coding . . . 20
2.3.5 ζ codes . . . 20
2.3.6 π codes . . . 21
2.4 Entropy coding . . . 21
2.4.1 Prefix coding and Huffman coding . . . 22
2.4.2 Arithmetic coding . . . 23
2.4.3 Asymmetric Numeral Systems . . . 24
2.5 Higher order entropy and entropy coding . . . 25

3 Hybrid Integer Encoding 27

3.1 Encoding scheme 28
3.2 Analysis 29
3.3 Experimental results 33

4 Novel techniques for high-order entropy coding **43**
4.1 Context clustering 44
4.1.1 Heuristic algorithm 46
4.2 Decision-tree-based context modeling 47
4.2.1 Optimal algorithm for $n = 1, k > n_t$ 49
4.2.2 Optimal algorithm for $n_t = 1$ 51
4.2.3 Heuristic algorithm in pseudolinear time for $n = 1$ 51
4.2.4 Heuristic for the general case 52

II Graph compression **55**

5 Common techniques for graph compression **57**
5.1 Raw link encoding 58
5.2 Grammar- and dictionary-based 59
5.3 Class-tailored 60
5.3.1 Compression of trees 60
5.4 Tree-based 62
5.5 Copying models 64
5.5.1 Brief summary of WebGraph 64
5.6 Decomposition 65
5.7 Reordering 66

6 Graph compression in theory **69**
6.1 Erdös-Rényi 69
6.2 Stochastic Block Model 70
6.3 Uniform attachment 71
6.4 Copy model 74
6.5 Preferential attachment (Barabási-Albert) 76
6.6 Simplified Copy Model 78

7 Graph compression in practice **81**
7.1 Encoding Integers 82
7.1.1 Negative integers 82

7.2 Graph compression in Zuckerli . . . 83
7.2.1 Context management . . . 84
7.2.2 Choice of reference list and chain . . . 85
7.2.3 Full decompression . . . 86
7.2.4 List decompression . . . 86
7.2.5 Approximation guarantee . . . 87
7.2.6 Details on computing the optimal sub-forest of F . . . 87
7.3 Experimental results . . . 88
7.3.1 Datasets . . . 90
7.3.2 Parameter Choice . . . 90
7.3.3 Effect of Approximation Algorithm and Context Modeling . . . 92
7.3.4 Compression Results and Resource Usage . . . 93
7.3.5 Performance Evaluation . . . 96
7.4 Further improvements on the Zuckerli scheme . . . 100
7.4.1 Tree-based context modeling . . . 100
7.4.2 Reference selection algorithm . . . 101
7.4.3 Experimental results . . . 102

8 Conclusions **105**
8.1 Future work . . . 106

Bibliography **107**

CHAPTER 1

INTRODUCTION

DATA COMPRESSION is a process through which the number of bits used to represent information can be reduced. In today's world, it is an incredibly important topic both for practical and theoretical pursuits.

Compression can be either *lossless*, when it allows to reconstruct the input exactly, or *lossy*, when it only allows to reconstruct an approximation of the input.

From the practical point of view, most of the services that we use today on the Internet would be significantly slower, if not entirely unfeasible, without using compression. Lossless compression, which is used by more than half the websites of the world [47], allows a $2-3x$ reduction of the amount of bytes for transmitting text content, layout and behaviour of webpages. Lossy compression of images achieves typically $10x$ savings over uncompressed data, and lossy compression of video can be even up to $1000x$ more efficient than uncompressed data, even with low information loss. It is thanks to compression, together with the improved processing power and connectivity that we enjoy in more modern times, that the Internet as we know it can exist.

From a theoretical point of view, compression is tightly tied with *information theory*, which tries to quantify the complexity of a given system. It is also for this reason that the capability of achieving good compression ratios on a given source has been connected with our *understanding* of the source, and the ability to compress of a system has been used as a measure of its degree of intelligence [73]. This connection between compression

and understanding has also been exploited in perhaps unexpected ways, such as as part of a measure of level of consciousness of patients in a coma or under the effects of medication [31].

On the other hand, graphs are one of the most versatile structures in mathematics and computer science, finding extremely varied applications, from answering questions on afternoon walks in Prussian towns [45] to building ranking algorithms for search engines [81] and doing community detection [37, 38].

Graph theory is also an important topic in theoretical research, being a source of multiple deep results like a quasipolynomial time algorithm for graph isomorphism [9] and the famous graph-minor theorem [90].

Given the importance of both those topics, the interest in compression of graphs is self-explanatory. In this thesis, we will focus on the compression of *large* graphs:

- From a theoretical point of view, analyses will be conducted keeping in mind the asymptotic behaviour of the systems in analysis, and will be focused on aspects of the theory that are relevant to large real-world networks.

- From a practical point of view, the focus will be mostly on graph classes with hundreds of millions or billions of edges, such as web graphs and social networks [69].

After this chapter, this thesis is divided in two parts. The rest of this chapter contains an overview of the structure of those two parts, as well as notation and definitions that are used throughout the thesis.

1.1 First part: general-purpose compression

The first part of this thesis presents general techniques for data compression. These techniques will then be used in the second part of the thesis, and more specifically in Chapter 7, to obtain a new graph compression method that significantly improves the state of the art.

Chapter 2 provides an overview of some known techniques for general-purpose compression.

Chapter 3 introduces a novel integer coding scheme that uses both entropy coding and raw bits, also giving an analysis of its efficiency on common distributions and a comparison with other known techniques.

Chapter 4 introduces novel techniques to achieve practical higher-order entropy coding.

The techniques of Chapters 3 and 4 have been partially developed by the author during the development of JPEG XL [2] and partially for this thesis.

1.2 Second part: graph compression

This part of the thesis is dedicated to, specifically, compression of graphs. It provides both theoretical and practical novel results in the topic of graph compression.

Chapter 5 provides an overview of the state of the art of encoding schemes for large graphs.

Chapter 6 contains the main theoretical results on graph compression in this thesis, providing a novel theoretical analysis of various graph models. In particular, it gives lower bounds on the compressed size of graphs belonging to a specific model and polynomial-time compression algorithms that match, or almost match, the lower bounds. Special focus is given to models for *sparse* graphs. While the results of this chapter are interesting from a theoretical point of view, they are not necessary for the practical results in Chapter 7.

Chapter 7 applies the techniques in Chapters 3 and 4 to graph compression, developing a new compression scheme that achieves significant density improvements compared to the state of the art. Part of the results in this chapter have been published in [98]; Section 7.4 provides a compression scheme that achieves some further improvements compared to [98], at the cost of an increase in encoding and decoding time.

1.3 Notation and definitions

We denote the base-2 logarithm of a positive number x as $\log x$, and its natural logarithm as $\ln x$, unless otherwise noted. We will assume $0 \log 0 = 0$ and $0^0 = 1$ for convenience of notation.

We will refer multiple times to two families of probability distributions on natural numbers: the *geometric* and the *Zipf* distribution. We remark that the Zipf distribution is usually defined on a range of positive integers, i.e. $\{1, \ldots, n\}$, while the distribution defined on $\mathbb{N}^+$ is usually denoted as the *Zeta* distribution. We will use the two terms interchangeably.

Differently from usual, we define the geometric distribution as the distribution of the number of failures of a sequence of independent trials each with a probability p of *failing* before the first success. It follows that the probability of an integer k is given by $p^k(1-p)$.

A Zipf distribution with parameter α is a distribution where the probability of an integer k is proportional to $k^{-\alpha}$. To ensure that the probabilities sum to 1, we normalize them using the Riemann ζ function, defined for all $\alpha > 1$:

$$\zeta(\alpha) = \sum_{n=1}^{\infty} \frac{1}{n^{\alpha}}$$

The probability of a given integer k under a Zipf distribution with parameter α is then $k^{-\alpha}\zeta(\alpha)^{-1}$.

A *graph* G is a pair (V, E) of *vertices* (or *nodes*) and *edges*, with $E \subseteq V \times V$. We will typically assume V to be of the form $\{1, \ldots, n\}$. The *degree* of a node δ_v is the number of edges that v belongs to. A sequence of distinct nodes $n_1, \ldots, n_k$ is a (simple) *path* if $(n_i, n_{i+1}) \in E$ for all i.

A graph can be *directed* or *undirected*: in an undirected graph, $(u, v) \in E \Leftrightarrow (v, u) \in E$; undirected edges are also denoted as $\{u, v\}$. If a graph is directed, we make the distinction of *in-degree* (δ_v^-) and *out-degree* (δ_v^+), the number of edges of which a node is respectively the first or the second element.

The *neighbours* of a node v in an undirected graph, denoted by $N(v)$, are all the nodes that share an edge with v. It follows that $|N(v)| = \delta_v$; *in-neighbours* ($N^-(v)$) and *out-neighbours* ($N^+(v)$) are defined in a similar way for directed graphs. An *adjacency list* for a node is the list of its (out-)neighbours.

A graph is *connected* if there is a path between any two nodes. A *connected component* of a graph is a maximal subset of nodes that is connected. An undirected graph is called a *tree* if there is a *unique* path between any two nodes. Any tree with n nodes has exactly $n - 1$ edges. Nodes of degree 1 in a tree are called leaves.

A tree may be *rooted* in a node, called the root. In a rooted tree, we implicitly assume edges to be oriented away from the root, and in that case we define, for any edge (u, v), u to be the *parent* of v and v to be a *child* of u.

A *forest* if its connected components are trees; equivalently, if each pair of edges has *at most one* path between them. It follows that the number of edges of a forest of n vertices is at most $n - 1$.

When the graph is clear from context and unless specified otherwise, we shall denote with n the number of nodes of a graph, and with m its number of edges.

When referring to graph representations, we will consider the following scenarios for access patterns.

- **Full decompression**: Decompress the representation entirely, obtaining the standard representation of G.

- **List decompression**: For any given node $u \in [n]$, decompress incrementally the adjacency list of u, while keeping the rest compressed.

- **Edge queries**: For any given pair of nodes (u, v), determine whether $(u, v) \in E$ or $(u, v) \notin E$.

The above scenarios are listed from least to most flexible: indeed, it is always possible to support list decompression using n edge queries per list (although faster implementations might be possible), and it is possible to support full decompression using n list decompressions.

1.4 Published material

During the course of the PhD, more work was done that is not as tightly related to the topic of this thesis. For completeness, a full list of publications is reported here:

- Efficient algorithms for listing k disjoint st-paths in graphs. R. Grossi, A. Marino, L. Versari - Latin American Symposium on Theoretical Informatics, 2018

- Tight Lower Bounds for the Number of Inclusion-Minimal st-Cuts. A. Conte, R. Grossi, A. Marino, R. Rizzi, T. Uno, L. Versari - International Workshop on Graph-Theoretic Concepts in Computer Science, 2018

- Finding maximal common subgraphs via time-space efficient reverse search. A. Conte, R. Grossi, A. Marino, L. Versari - International Computing and Combinatorics Conference, 2018

- D2K: scalable community detection in massive networks via small-diameter k-plexes. A. Conte, T. De Matteis, D. De Sensi, R. Grossi, A. Marino, L. Versari - Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2018

- Listing subgraphs by Cartesian decomposition. A. Conte, R. Grossi, A. Marino, R. Rizzi, L. Versari - International Symposium on Mathematical Foundations of Computer Science, 2018

- Discovering k-Trusses in Large-Scale Networks. A. Conte, D. De Sensi, R. Grossi, A. Marino, L. Versari. IEEE High Performance extreme Computing Conference (HPEC), 2018

- Listing maximal subgraphs satisfying strongly accessible properties. A. Conte, R. Grossi, A. Marino, L. Versari. SIAM Journal on Discrete Mathematics 33, 2019
- JPEG XL next-generation image compression architecture and coding tools. J. Alakuijala, R. van Asseldonk, S. Boukortt, M. Bruse, IM. Comșa, M. Firsching, T. Fischbacher, E. Kliuchnikov, S. Gomez, R. Obryk, K. Potempa, A. Rhatushnyak, J. Sneyers, Z. Szabadka, L. Vandevenne, L. Versari, J. Wassenberg - Applications of Digital Image Processing XLII, 2019
- A fast discovery algorithm for large common connected induced subgraphs. A. Conte, R. Grossi, A. Marino, L. Tattini, L. Versari - Discrete Applied Mathematics 268, 2019
- Sublinear-Space and Bounded-Delay Algorithms for Maximal Clique Enumeration in Graphs A. Conte, R. Grossi, A. Marino, L. Versari - Algorithmica, 2020
- Benchmarking JPEG XL image compression. J. Alakuijala, S. Boukortt, T. Ebrahimi, E. Kliuchnikov, J. Sneyers, E. Upenik, L. Vandevenne, L. Versari, J. Wassenberg - Optics, Photonics and Digital Technologies for Imaging Applications VI, 2020
- Temporal coding in spiking neural networks with alpha synaptic function. IM. Comsa, T. Fischbacher, K. Potempa, A. Gesmundo, L. Versari, J. Alakuijala - IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2020
- Truly Scalable K-Truss and Max-Truss Algorithms for Community Detection in Graphs. A. Conte, D. De Sensi, R. Grossi, A. Marino, L. Versari - IEEE Access, 2020
- Zuckerli: A New Compressed Representation for Graphs. L. Versari, IM. Comsa, A. Conte, R. Grossi, IEEE Access, 2020
- Intelligent Matrix Exponentiation. T. Fischbacher, IM. Comsa, K. Potempa, M. Firsching, L. Versari, J. Alakuijala - to appear.
- ISO/IEC DIS 18181-1: Information technology - JPEG XL Image Coding System - Part 1: Core coding system

Part 1

General-purpose compression

CHAPTER 2

COMPRESSION CONCEPTS AND TECHNIQUES

ENTROPY is a fundamental concept in data compression, giving a lower bound on the amount of bits necessary to represent a given piece of data. In this chapter, we define the concept of *entropy of a random variable*, and its application to data compression in the form of Shannon's Source Coding Theorem [94].

We then discuss some well-known techniques for encoding a sequence of independent, identically distributed random variables (typically positive integers or symbols from a given alphabet), which is also known as *order-0* compression.

Finally, we discuss the concept of *higher-order* compression, which achieves better results when compressing a sequence of non-independent random variables, as is often the case for e.g. text, or graphs.

2.1 Entropy of a random variable

Consider the process of flipping a biased coin that has a probability p of obtaining heads. We define the *information content*, or *surprise*, of obtaining heads on such a coin as

$$I(\text{heads}) = -\log p$$

Note that the information content of "heads" decreases as p increases. This matches intuition, since it is less surprising to obtain heads on a coin with a probability of heads very close to 1.

We then define the information content of obtaining tails as

$$I(\text{tails}) = -\log(1-p)$$

We can now define the *entropy* of our coin flip as *the expected amount of information content*:

$$H(\text{coin flip}) = \sum_{e \in \{\text{heads},\text{tails}\}} Pr(e)I(e) = -p\log p - (1-p)\log(1-p)$$

Under this definition, the entropy of a coin with $p = 0$ or $p = 1$ is 0. This matches intuition as an event that is certain gives no information (or surprise).

We can generalize these definitions to any random event x and any random variable X.

Definition 1. *The* ***information content****, or* ***surprise****, of an event x is defined as*

$$I(x) = -\log Pr(x)$$

The ***entropy*** *of a random variable X is the expected value of the information content of X. For a discrete variable, we have*

$$H(x) = \mathbb{E}[I(x)] = \sum_{x \in X} Pr(x)I(x) = -\sum_{x \in X} Pr(x)\log Pr(x)$$

Following the definition, we can compute the entropy for some common random variables.

- The entropy of a random variable that is uniformly distributed over n values is $H(U_n) = \log n$.
- The entropy of a Bernoulli random variable with probability p is, as computed above, $-p\log p - (1-p)\log(1-p)$. We will denote this value as H^p.
- The entropy of a subset of n elements in an universe of m elements, chosen uniformly at random, is $\log\binom{m}{n} = mH^{\frac{n}{m}} + O(\log m)$

The fundamental application of entropy to data compression is given by Shannon's Source Coding Theorem [94]:

Theorem 2 (Source Coding Theorem)**.** *With high probability, at least $nH(X)$ bits are required to represent a sequence of n i.i.d. random variables with probability distribution X.*

One important property of entropy is that it is *additive*: the entropy of a sequence of independent random variables is equal to the sum of the entropies of each random variable.

In the rest of this chapter, we will refer to the *redundancy* of an encoding scheme:

Definition 3 (Redundancy)**.** *The* ***redundancy*** *of a given encoding scheme for a specific source is given by the* ***ratio*** *between the number of bits used by the encoding scheme and the lower bound given by the Source Coding Theorem, i.e. the entropy of the source.*

It follows that the redundancy is always at least 1. We remark that this definition is somewhat different from the usual one, where the redundancy is the difference between the number of used bits and the entropy.

2.2 Entropy of a text

Here and in the rest of this thesis, when we mention a *text* we refer to a sequence of symbols $T = t_0 t_1 \dots t_n$ of symbols from a given alphabet Σ.

To define the entropy for a text, we need to define a probability distribution for it. The simplest definition is that of *order* 0 entropy, in which we assume every symbol to be chosen independently of the others.

Typically, we assign to each symbol a probability equal to its *frequency*. Thus, if we denote with n_s the number of occurrences of s in T, we have

$$H_0(T) = \sum_{s \in \Sigma} n_s \log \frac{n}{n_s}$$

However, this definition is not entirely satisfactory, as in most natural languages symbol choices are *not* independent: for example, in an English text, the sequence `don'` is almost always followed by `t`.

The concept of *higher order entropy* models this property. In particular, to define the k-th order entropy of a text H_k, we consider every symbol to belong to a different probability distribution depending on *the previous k symbols*.

Let's consider, as an example, the text $T_n = a^n b^n$. As `a` and `b` appear the same number of times in T_n, each of them has a frequency of 0.5, and thus the order 0 entropy of T_n is

$$H_0(T_n) = 0.5n + 0.5n = n$$

For computing the order 1 entropy, we divide the symbols in T_n in 3 groups:

- The first symbol in the text: by definition, there is only one such symbol, which occurs with frequency 1 and thus provides no entropy.
- Symbols preceded by a `a`: there are $n-1$ symbols `a` and one symbol `b`, for a total entropy of $nH^{\frac{1}{n}}$.
- Symbols preceded by a `b`: there are exactly $n-1$ symbols `b`, which occur with frequency 1 and thus provide no entropy.

The total order 1 entropy is thus

$$H_1(T_n) = nH^{\frac{1}{n}} = \log n + (n-1)\log\frac{n}{n-1} = \log n + O(1)$$

This value is significantly smaller than the order 0 entropy: this is because T_n has a very specific structure and symbol choices are very far from being independent.

2.3 Integer coding

It is common to encode integers that are independently produced with a given distribution. Multiple encoding schemes have been produced, suited to various distributions; as a consequence of the Source Coding Theorem, a given encoding scheme that uses $x(i)$ bits for an integer i is optimal for a distribution where i has a probability of $2^{-x(i)}$, if $\sum_{i=1}^{\infty} 2^{-x(i)} = 1$.

All the encodings described in this section will be described in the setting of encoding *positive* integers. If it is necessary to also encode 0, the number to be encoded can be increased by 1.

2.3.1 Unary coding

This is one of the simplest encodings: to encode x, a sequence of $x-1$ binary digits 1 is produced, followed by a single digit 0. It is also common to see the roles of 0 and 1 be swapped, which doesn't change the characteristics of the encoding.

Encoding x thus uses x bits: this encoding is optimal for sources for which x has probability 2^{-x}, i.e. a geometric distribution of parameter $\frac{1}{2}$.

2.3.2 Rice coding

Rice coding can be seen as a generalization of unary coding. It is defined by a parameter $M = 2^k$. When $k = 0$, this procedure defines the same encoding as unary coding.

In general, a number N is encoded by:

- Encoding $\left\lfloor \frac{N}{2^k} \right\rfloor$ in unary, and
- Encoding $N \mod 2^k$ using exactly k bits.

It follows that encoding N requires $\left\lfloor \frac{N}{2^k} \right\rfloor + k$ bits. In general, this encoding can only be optimal for sequences in which each group of consecutive 2^k symbols is equiprobable. However, if we ignore the errors introduced by rounding, we can see that Rice coding for a given k is optimal if $\left\lfloor \frac{N}{2^k} \right\rfloor$ is distributed geometrically with $p = \frac{1}{2}$. This property suggests that Rice coding is a good choice for geometrically distributed sources where each trial has probability of *failure* of $\frac{1}{2^{2^{-k}}}$.

Finally, we remark that it is possible to generalize Rice coding to the case where N is not a power of two. The resulting coding scheme is called Golomb coding.

2.3.3 Elias γ coding

All the codes described so far are most suitable for *geometric* distributions. However, geometric distributions decrease in probability very quickly compared to many real-world data streams. Elias γ coding [42] is a code that addresses this issue, being suitable for numbers that follow a power-law distribution.

The Elias γ code for a positive integer N is defined as follow:

- First, $\lfloor \log N \rfloor + 1$ is encoded in unary.
- Then, the lowest $\lfloor \log N \rfloor$ bits of N are encoded directly.

It follows that this encoding uses $2\lfloor \log N \rfloor + 1$ bits to represent N; it is thus suitable for integers distributed in such a way that the probability of N is close to

$$2^{-2\lfloor \log N \rfloor - 1} \approx \frac{1}{2N^2}$$

Thus, γ coding is suitable for integers following a Zipf distribution with exponent close to 2.

2.3.4 Elias δ coding

The γ code [42] for an integer N is defined as follows:

- First, $\lfloor \log N \rfloor + 1$ is encoded using Elias γ coding.
- Then, the lowest $\lfloor \log N \rfloor$ bits of N are encoded directly.

It follows that this encoding uses $2\lfloor \log \lfloor \log N \rfloor \rfloor + \lfloor \log N \rfloor + 1$ bits to represent N; it is thus suitable for integers distributed in such a way that the probability of N is close to

$$2^{-2\lfloor \log \lfloor \log N \rfloor \rfloor - \lfloor \log N \rfloor - 1} \approx \frac{1}{2N \log^2 N}$$

This probability distribution decreases more slowly than any Zipf distribution (as the sum of N^{-1} over the natural numbers diverges, and hence there is no Zipf distribution of exponent 1). However, Zipf distributions of exponent very close to 1 should be well represented by Elias δ coding.

2.3.5 ζ codes

Elias codes provide a good representation for Zipf distributions with exponents close to 1 and 2. To address the necessity of encoding Zipf distributions with intermediate exponents, such as ones that are common in web graphs that have an exponent of ≈ 1.3, [23] introduces ζ codes.

To define this code, we first need to define the *minimal binary code* of an integer $n \in [0, z)$.

Let $s = \lceil \log z \rceil$. The minimal binary code of n is defined as

- If $n < 2^s - z$, then n is represented as its binary representation using $s - 1$ bits.
- If $n \geq 2^s - z$, then n is represented as the binary representation of $n - z + 2^s$ using s bits.

We remark that, if $z = 2^d$, this representation corresponds to usual binary representation on d digits.

ζ codes are parameterized by a *shrinking factor* k. To represent an integer N, let h be such that $N \in [2^{hk}, 2^{(h+1)k})$. Then, the representation is given by

- $h + 1$ written in unary, and
- A minimal binary code of $N - 2^{hk}$, with $z = 2^{(h+1)k} - 2^{hk}$.

When $k = 1$, then $h = \lfloor \log N \rfloor$ and $2^{(h+1)k} - 2^{hk} = 2^h$, making the minimal binary code correspond to the binary code on h digits. Hence, ζ_1 is the same as Elias γ coding.

In the general case, the ζ_k code uses $\lfloor \log N/k + 1 \rfloor (k+1)$ or $\lfloor \log N/k + 1 \rfloor (k+1) + 1$ bits to encode an integer N, depending on which of the two different representations is used for the minimal binary code. This corresponds to an implied probability of approximately

$$\Theta\left(2^{-(\log N/k+1)(k+1)}\right) = \Theta\left(N^{-(k+1)/k}\right) = \Theta\left(\frac{1}{N^{1+\frac{1}{k}}}\right)$$

which suggests that ζ_k codes are good for integers distributed with a Zipf law with exponent $1 + \frac{1}{k}$. Plugging in $k = 1$ gives again the observation that Elias γ codes are good for Zipf distributions with exponent 2, as expected.

2.3.6 π codes

π codes, introduced in [5], allow efficient encoding of Zipf distributions that have an exponent even closer to 1 compared to ζ codes. Like ζ codes, they are also parameterized by a positive integer k.

The encoding is defined as follows. Let $h = 1 + \lfloor \log N \rfloor$, and let $l = \left\lceil \frac{h}{2^k} \right\rceil$. The code is given by

- l written in unary,
- $2^k l - h$ written using exactly k bits, as it is a number in $[0, 2^k)$, and
- The least significant $\lfloor \log N \rfloor$ bits of N.

The total number of bits used to represent N is thus $k + \left\lceil \frac{1+\lfloor \log N \rfloor}{2^k} \right\rceil + \lfloor \log N \rfloor$. This code is hence optimal for distributions where the probability of N is approximately

$$\Theta\left(2^{-\log N/2^k + \log N}\right) = \Theta\left(\frac{1}{N^{1+2^{-k}}}\right)$$

which is the case for Zipf distributions of exponent $\alpha \approx 1 + 2^{-k}$. Thus, π codes can encode efficiently Zipf distributions with exponents significantly closer to 1 for a similar value of k when compared to ζ codes.

2.4 Entropy coding

Entropy coding is a set of techniques that allows us to achieve compression close to the entropy bound for any probability distribution, as long as the distribution is known in

advance both on the encoding and on the decoding side of the communication. This typically implies that the compressed representation should start with some representation of the probability distributions.

2.4.1 Prefix coding and Huffman coding

Prefix coding represents symbols from a given alphabet Σ as a sequence of bits, with the property that for any two distinct symbols $s, t \in \Sigma$ the sequence of bits that corresponds to s is *not* a prefix of the sequence of bits that corresponds to t. This property is usually referred to as the code being *prefix-free*.

The advantage of a prefix-free code is the simplicity of the decoding procedure: indeed, it is sufficient to read one bit at a time from the encoded stream, extending a sequence of bit values. As soon as the sequence of bit values corresponds to the sequence associated with a symbol, that symbol can be produced, and the sequence cleared: this is because there is no ambiguity between the sequence for a symbol or the prefix of the sequence for another symbol.

However, in practical implementations, usually the decoding procedure uses a table of size 2^k, where k is the longest length of a sequence, that contains the first symbol to be decoded for each of the possible combinations, and decoding then proceeds by reading k bits at a time, looking up the symbol to be decoded, and then advancing the position in the encoded stream by the correct amount (possibly less than k).

Prefix coding is, by definition, *stateless*: it is possible to resume decoding from any symbol boundary in the bitstream, without the need of any extra information. Hence, prefix codes are well suited for streams that require random access to specific positions in the encoded data, as the bit position of the starting bit of the required symbol is sufficient to resume decoding.

There are multiple ways to construct a prefix code for a given distribution of symbols. Among those, the most well-known is certainly *Huffman coding* [58], which produces optimal prefix codes; note that this does not mean that Huffman coding produces an optimal encoding, but just that no prefix code can have lower cost. It can be described as follows:

- At the beginning, $|\Sigma|$ single-node binary trees are created, one per symbol; we define the *size* of each of those trees as the number of occurrences of the corresponding symbol.

- The two trees of smallest size are merged together by creating a new tree that

contains a root with the two trees as left and right children; the size of the new tree is the sum of the sizes of the old trees.

- The process is repeated until there's only one tree.

- The code associated with each symbol is given by the path from the root to the corresponding leaf, where each left branch is represented by a 0 bit and each right branch by a 1 (or vice-versa).

Other algorithms to build prefix codes are known. Among those, of particular note is [67], which provides an algorithm to build optimal *length-limited* prefix codes, i.e. prefix codes for which the code word for any symbol does not exceed a specified length.

2.4.2 Arithmetic coding

While prefix coding is more flexible than fixed integer codes, it uses an integer number of bits per symbol; it follows from the definition of entropy that it cannot be optimal unless $-\log p \in \mathbb{N}$, i.e. the probability of each symbol is a power of 2.

One possible method to overcome this limitation is *arithmetic coding* [84]. The main idea of arithmetic coding is to represent the sequence of symbols as a single number in $[0, 1)$.

To this end, we proceed as follows:

- We initialize the current interval as $[0, 1)$.

- We split the current interval into sub-intervals, with one interval per symbol, and length proportional to the probability p_s of each symbol s.

- We replace the current interval with the interval corresponding to the next symbol, and repeat from the previous step while there's still symbols to encode.

- At the end, we pick any fraction with a denominator of the form 2^k that is contained in the final interval; the encoding of the sequence is given by the numerator of this fraction, represented using k bits.

If we choose $p_s = \frac{n_s}{n}$, where n_s is the number of occurrences of a symbol s and n is the total number of symbols, the size of the final interval is given by

$$\prod_{s \in \Sigma} \left(\frac{n_s}{n} \right)^{n_s}$$

Since any interval of size ϵ contains a fraction with denominator $2^{-\log \epsilon + 1}$, it follows that the total number of bits used by arithmetic coding is at most (ignoring the cost of encoding the number of bit itself, which is at most an additional logarithmic term)

$$1 - \log \prod_{s \in \Sigma} \left(\frac{n_s}{n} \right)^{n_s} = 1 + \sum_{s \in \Sigma} n_s \log \frac{n}{n_s}$$

which matches the entropy bound of Section 2.2 up to one extra bit.

In practical implementations, it is of course not ideal to operate with arbitrary precision floating point numbers. Hence, probabilities are typically rounded to some arbitrary small value like 2^{-12}, and the arithmetic coder is *flushed* periodically: whenever the interval becomes sufficiently small, some bits are written to the output stream and the interval is rescaled by the corresponding power of two. This allows to use arithmetic coding with integer arithmetic, instead of arbitrary precision floating point arithmetic. However, even a small interval does not always allow to fully determine the next bits to be written; it is possible to overcome this issue, but as arithmetic coding is not the focus of this thesis, we refer the reader to [77] for more detail about its practical implementation.

The case in which $|\Sigma| = 2$ is particularly simple and efficient to implement in practice, and is often referred to as *binary arithmetic coding*.

2.4.3 Asymmetric Numeral Systems

Like arithmetic coding, Asymmetric Numeral Systems, or ANS [41], encode a sequence $s_1, \ldots, s_n$ of input symbols in a single number x that can be represented with a number of bits that is close to the entropy of the data stream. However, compared to traditional methods of arithmetic coding it can achieve faster decompression speeds, at the cost of requiring the decoding process to obtain symbols in *reverse* order compared to the encoding process.

The encoding process adds a symbol s to the sequence represented by a number x by producing a new integer

$$C(s, x) = M \left\lfloor \frac{x}{F_s} \right\rfloor + B_s + (x \bmod F_s)$$

where M is the sum of the frequencies of all the symbols, F_s is the frequency of the symbol s and B_s is the cumulative frequency of all the symbols before s. The inverse of this function is

$$D(x) = \left(s, F_s \left\lfloor \frac{x}{M} \right\rfloor + (x \bmod M) - B_s \right)$$

where s is such that $B_s \leq x \bmod M < B_{s+1}$. The decoder can thus reverse the encoding process, producing a sequence of symbols from x. Notably, decoding does not require

any division operation (except by M, which is usually a power of 2); moreover, s can be computed by a lookup in a precomputed table of M elements.

Like all variants of arithmetic coding, practical implementations of ANS do not use arbitrary precision arithmetic, but rather they keep an internal state in a fixed range $[S, 2^b S)$ that is manipulated for each symbol in the stream: when the state overflows, it yields b bits during encoding; when the state underflows, it consumes b bits when decoding. For correct decoding, it is required that S is a multiple of M. A reasonable choice is to set $S = 2^{16}$, $M = 2^{12}$, and $b = 16$. More details about practical ANS implementations can be found in [78].

Note that, since the encoding procedure is just the reverse of the decoding procedure, ANS makes it easy to interleave non-compressed and compressed bits. This is especially useful when encoding integers using both entropy coding and directly coded bits, as done in Chapter 3.

2.5 Higher order entropy and entropy coding

As entropy coding requires communicating the probability distributions in advance, it quickly becomes unpractical for higher order entropy coding. For example, to encode with order-2 entropy coding a typical binary file (which has 256 distinct symbols), it would be necessary to communicate 65 536 distributions.

Two common workarounds for this issue are:

- To use other techniques that can exploit higher order correlations in the source data; for example, the Lempel-Ziv sliding window compression scheme [106] has been shown [103] to be asymptotically optimal even for data with higher order correlations. As another example, the usage of run-length encoding [91], that is, encoding a repetition count together with each encoded symbol, allows to get rid of the simplest correlations such as long sequences of repeated symbols. Finally, it was shown in [61] that applying the Burrows-Wheeler Transform also allows to compress its input within higher-order entropy bounds.

- To use *adaptive* probability models, that is, to transmit a small number of probability distributions that get modified in the same way during the encoding and decoding process. This is what is done for example in CABAC, in the H. 264 video coding standard [65].

CHAPTER 3

HYBRID INTEGER ENCODING

HYBRID INTEGER ENCODING is a novel, general technique to encode integers with potentially very large values using a combination of entropy coding and raw bits. This scheme was initially developed in the context of the JPEG XL image compression format [2].

Directly applying entropy coding to encode integers from unbounded (or, in practice, even bounded with very large values) distributions is unfeasible, because it would require communicating an arbitrary distribution over $\mathbb{N}$.

To circumvent this issue, a common idea is to have a single entropy-coded symbol represent multiple integers, and then use additional bits to disambiguate. This idea dates back to at least the JPEG standard [99].

To the best of our knowledge, Hybrid Integer Encoding is the first encoding scheme that is parameterized to allow different trade-offs between integer range reduction and precision of the encoding. Moreover, it is defined analytically, making it suitable for any range of integers, while previous schemes were defined case-by-case, making them usable only on a limited range. A similar technique is introduced independently in [78]; this scheme proceeds by representing the integer in a fixed base (say, base 16) and representing the first digits using entropy coding; the rest of the digits are then represented directly. Hybrid integer encoding is more flexible in that it allows increasing precision for smaller numbers, as well as specifying a number of least-significant bits

to be represented by entropy coding (for cases where i.e. integers to be encoded are all even). Moreover, the precision of the approximation is more uniform across the space of represented integers; more details and an experimental comparison can be found in Section 3.3.

Hybrid Integer Encoding is defined by three parameters i, j and k, with $k \geq i + j$ and $i, j \geq 0$. The integer encoding scheme initially described in [2] corresponds to the variant of Hybrid Integer Encoding with $k = 4, i = 1, j = 0$.

In the rest of this chapter we describe the proposed scheme and analyze the performance of this method on various distributions of integers, comparing to the specific integer codes that have been described in Section 2.3. For ease of implementation, we will restrict the probability distributions to the $[0, 2^{32})$ range.

3.1 Encoding scheme

The k parameter defines the number of *direct* symbols. Every integer in the range $[0, 2^k)$ is encoded directly as symbol in the alphabet.

Any other integer $x \geq 2^k$ is encoded as follows. First, consider the binary representation of x: $b_p b_{p-1} \cdots b_1$, where $b_p = 1$ is the highest non-zero bit, and p its index (hence $p = \lfloor \log x \rfloor + 1$). Identify x with its corresponding triple (m, t, l), where m is the integer formed by the i bits $b_{p-1} \cdots b_{p-i}$ following b_p, l is the integer formed by the rightmost j bits $b_j \cdots b_1$, and t is the integer encoded by the bits between those of m and l, as illustrated below:

$$1\,\overbrace{b_{p-1} \ldots b_{p-i}}^{m}\,\overbrace{b_{p-i-1} \ldots b_{j+1}}^{t}\,\overbrace{b_j \ldots b_1}^{l}$$

Clearly, given the triple (m, t, l), we can reconstruct x. We conveniently encode that triple by a pair (s, t) where $s = 2^k + (p - k - 1) \cdot 2^{i+j} + m \cdot 2^j + l$ encodes the value of $p \geq k + 1$ by $(p - k - 1) \cdot 2^{i+j}$, the value of m as $m \cdot 2^j$, and the value of l. By adding 2^k to s we guarantee that $s \geq 2^k$, as values of $s < 2^k$ represent values of $x < 2^k$ directly.

For example, for $k = 4$, $i = 1$, and $j = 2$, the integer $x = 211$ has binary representation `1 1 0100 11` (with $p = 8$) and its corresponding triple is $(1, 4, 3)$; it is thus encoded as the pair $(16 + 3 \cdot 8 + 1 \cdot 4 + 3, 4) = (47, 4)$. As another example, when $k = 4$, $i = 1$ and $j = 1$, the integers from 0 to 15 are encoded with their corresponding symbol s in the alphabet, and t is empty; 23 has binary representation `10111` and thus is encoded as symbol `17` (the highest set bit is in position 5, the following bit is 0, and the last bit is 1), followed

Algorithm 1: How to decode an (s, t) pair.

```
if s < 2^k then
    return s;
l ← (s − 2^k) mod 2^j;
h ← (s − l − 2^k) / 2^j;
m ← h mod 2^i;
n ← (h − m) / 2^i        (note n = p − k − 1);
return 2^(n+k) + m · 2^(n+k−i) + t · 2^j + l;
```

by the two remaining bits `11`; 33 is encoded as symbol `21` (highest set bit is in position 6, following bit is 0 and last bit is 1) followed by the three remaining bits `000`.

As for t, it is stored as-is in the encoded stream, just after entropy coding s. Note that it is possible to compute the number of bits of t from s, without knowing x: this allows the decoder to know how many bits to read. The procedure to decode an integer from the (s, t) pair consists of recovering the corresponding triple (m, t, l) and then reconstructing x, and is detailed in Algorithm 1.

Table 3.1 reports more values of s, t and the number of bits n of t for some combinations of k, i, j.

We remark that the number of entropy coded symbols is *logarithmic* in the range of the actual integers to encode; thus, it is necessary to store very few probability values even when encoding integers with a range of billions.

3.2 Analysis

We will now provide an analysis of the compression performance of the given encoding scheme, both theoretically and with experimental results, including comparisons with existing schemes.

Theorem 4. *The redundancy of Hybrid Integer Encoding with parameters $k, i, 0$ on a stream of i.i.d. random variables, where value v has probability p_v, can be bounded from above by*

$$r = \max_{a \geq k} \left\{ 1, \max_{0 \leq b < 2^i} \left\{ \max_{v \in I_{a,b}} \frac{a - i - \log \sum_{v' \in I_{a,b}} p_{v'}}{-\log p_{v'}} \right\} \right\}$$

where $I_{a,b} = [2^a + b \cdot 2^{a-i}, 2^a + (b+1) \cdot 2^{a-i})$.

Proof. By the definition of Hybrid Integer Encoding, for any integer in $I_{a,b}$ with $a \geq k$ and $0 \leq b < 2^i$, we obtain the same value of s, and t has a length of $a - i$ bits.

	4,2,0			4,1,1			2,1,0			2,0,2		
	s	n	t	s	n	t	s	n	t	s	n	t
0	0	0	–	0	0	–	0	0	–	0	0	–
1	1	0	–	1	0	–	1	0	–	1	0	–
2	2	0	–	2	0	–	2	0	–	2	0	–
3	3	0	–	3	0	–	3	0	–	3	0	–
4	4	0	–	4	0	–	4	1	0	4	0	–
5	5	0	–	5	0	–	4	1	1	5	0	–
6	6	0	–	6	0	–	5	1	0	6	0	–
7	7	0	–	7	0	–	5	1	1	7	0	–
8	8	0	–	8	0	–	6	2	00	8	1	0
9	9	0	–	9	0	–	6	2	01	9	1	0
10	10	0	–	10	0	–	6	2	10	10	1	0
11	11	0	–	11	0	–	6	2	11	11	1	0
12	12	0	–	12	0	–	7	2	00	8	1	1
13	13	0	–	13	0	–	7	2	01	9	1	1
14	14	0	–	14	0	–	7	2	10	10	1	1
15	15	0	–	15	0	–	7	2	11	11	1	1
16	16	2	00	16	2	00	8	3	000	12	2	00
17	16	2	01	17	2	00	8	3	001	13	2	00
22	17	2	10	16	2	11	8	3	110	14	2	01
23	17	2	11	17	2	11	8	3	111	15	2	01
24	18	2	00	18	2	00	9	3	000	12	2	10
25	18	2	01	19	2	00	9	3	001	13	2	10
26	18	2	10	18	2	01	9	3	010	14	2	10
27	18	2	11	19	2	01	9	3	011	15	2	10
28	19	2	00	18	2	10	9	3	100	12	2	11
29	19	2	01	19	2	10	9	3	101	13	2	11
30	19	2	10	18	2	11	9	3	110	14	2	11
31	19	2	11	19	2	11	9	3	111	15	2	11
32	20	3	000	20	3	000	10	4	0000	16	3	000
33	20	3	001	21	3	000	10	4	0001	17	3	000
63	23	3	111	23	3	111	11	4	1111	19	3	111
64	24	4	0000	24	4	0000	12	5	00000	20	4	0000
65	24	4	0001	25	4	0000	12	5	00001	21	4	0000
127	27	4	1111	27	4	1111	13	5	11111	23	4	1111
128	28	5	00000	28	5	00000	14	6	000000	24	5	00000

Table 3.1: Token, number of bits and raw bits for some hybrid integer configurations.

Thus, the probability of the value s corresponding to integers in $I_{a,b}$ will be

$$\sum_{v \in I_{a,b}} p_v$$

It follows that encoding an integer in $I_{a,b}$ will use a total number of bits given by

$$a - i - \log \sum_{v \in I_{a,b}} p_v$$

For values below 2^k, s is sufficient for encoding the integer, so the number of used bits will be $-\log p_v$.

Overall, the expected number of bits used by hybrid integer encoding is given by

$$\begin{aligned} B &= \sum_{v<2^k} -p_v \log p_v + \sum_{a \geq k} \sum_{0 \leq b < 2^i} \sum_{v \in I_{a,b}} p_v \left(a - i - \log \sum_{v' \in I_{a,b}} p_{v'} \right) \\ &\leq \sum_{v<2^k} -p_v \log p_v + \sum_{v \geq k} -r p_v \log p_v \\ &\leq r \sum_{v} -p_v \log p_v \end{aligned}$$

which proves the theorem. □

Corollary 5. *For $j > 0$, the redundancy can be bounded by the maximum redundancy of 2^j streams of i.i.d. variables, each coded with a Hybrid Integer Code with parameters $k - j, i, 0$ and containing the integers equal to $\{0, \cdots, 2^j - 1\}$ modulo 2^j.*

Using Theorem 4, we can give an upper bound for the redundancy of Hybrid Integer Encoding on a geometric distribution with parameter $0 < \alpha < 1$. We recall that in a geometric distribution $p_v = \alpha^v(1 - \alpha)$.

We first compute

$$\begin{aligned} \sum_{v \in I_{a,b}} p_v &= \sum_{v \in I_{a,b}} \alpha^v (1 - \alpha) \\ &= (1 - \alpha)\alpha^{2^a + 2^{a-i}b} \sum_{v < 2^{a-i}} \alpha^i \\ &= (1 - \alpha^{2^{a-i}})\alpha^{2^a + 2^{a-i}b} \end{aligned}$$

As p_v is decreasing, the maximum over $I_{a,b}$ is achieved on $v = 2^a + 2^{a-i}b$.

Thus, we can rewrite the formula in Theorem 4 as

$$\frac{a - i - (2^a + 2^{a-i}b) \log \alpha - \log(1 - \alpha^{2^{a-i}})}{-(2^a + 2^{a-i}b) \log \alpha - \log(1 - \alpha)}$$

Observing that $\log(1-\alpha) \le \log(1-\alpha^{2^{a-i}}) \le 0$, and that $\log \alpha < 0$, thus making the denominator minimum when $b = 0$, we can bound the given formula from above by

$$1 + \frac{(a-i)2^{-a}}{\log \frac{1}{\alpha}}$$

To compute the maximum of this value, we first differentiate $(a-i)2^{-a}$, observing that its maximum occurs in $i + \log e$. Thus, the maximum over integers will occur either for $a = i+1$ or $a = i+2$, which both yield $2^{-(i+1)}$; moreover, as a increases beyond $i+2$, the value of $(a-i)2^{-a}$ decreases.

As $a \ge k$, we can state the following

Theorem 6. *The redundancy of Hybrid Integer Encoding with parameters* $k, i, 0$ *when encoding a geometric distribution with parameter* α *is at most*

$$r_{\alpha,k,i} \le \begin{cases} 1 + \frac{2^{-(i+1)}}{\log \frac{1}{\alpha}} & \text{if } k \le i+2 \\ 1 + \frac{(k-i)2^{-k}}{\log \frac{1}{\alpha}} & \text{otherwise} \end{cases}$$

We now consider the case of a generic *decreasing* probability distribution. If we define $\hat{p}_{a,b} = p_{2^a+2^{a-i}b}$ and $\check{p}_{a,b} = p_{2^a+2^{a-i}(b+1)-1}$, it follows that

$$\sum_{v \in I_{a,b}} p_v \ge 2^{a-i} \check{p}_{a,b}$$

Thus, we can state

Theorem 7. *The redundancy of Hybrid Integer Encoding with parameters* $k, i, 0$ *when encoding a decreasing distribution is at most*

$$\max_{\substack{a \ge k \\ 0 \le b < 2^i}} \frac{\log \check{p}_{a,b}}{\log \hat{p}_{a,b}}$$

Applying this theorem, we can also obtain a bound for Zipf distributions:

Theorem 8. *The redundancy of Hybrid Integer Encoding with parameters* $k, i, 0$ *when encoding a Zipf distribution with parameter* α *is at most*

$$r_{\alpha,k,i} \le 1 + \frac{\log(1+2^{-i})}{k + \frac{\log \zeta(\alpha)}{\alpha}}$$

Proof. We have

$$\hat{p}_{a,b} = (2^a + 2^{a-i}b)^{-\alpha}\zeta(\alpha)^{-1}$$

and

$$\check{p}_{a,b} = (2^a + 2^{a-i}(b+1) - 1)^{-\alpha}\zeta(\alpha)^{-1}$$

Thus,

$$\begin{aligned}
\max_{\substack{a\geq k\\ 0\leq b<2^i}} \frac{\log \hat{p}_{a,b}}{\log \check{p}_{a,b}} &= \max_{\substack{a\geq k\\ 0\leq b<2^i}} \frac{\alpha\log(2^a + 2^{a-i}(b+1) - 1) + \log\zeta(\alpha)}{\alpha\log(2^a + 2^{a-i}b) + \log\zeta(\alpha)} = \\
&= \max_{\substack{a\geq k\\ 0\leq b<2^i}} 1 + \frac{\alpha\log(2^a + 2^{a-i}(b+1) - 1) - \alpha\log(2^a + 2^{a-i}b)}{\alpha\log(2^a + 2^{a-i}b) + \log\zeta(\alpha)} = \\
&= 1 + \max_{\substack{a\geq k\\ 0\leq b<2^i}} \frac{\log\frac{2^a+2^{a-i}(b+1)-1}{2^a+2^{a-i}b}}{\log(2^a + 2^{a-i}b) + \frac{\log\zeta(\alpha)}{\alpha}} = \\
&= 1 + \max_{\substack{a\geq k\\ 0\leq b<2^i}} \frac{\log\left(1 + \frac{2^{a-i}-1}{2^a+2^{a-i}b}\right)}{\log(2^a + 2^{a-i}b) + \frac{\log\zeta(\alpha)}{\alpha}} = \\
&= 1 + \max_{a\geq k} \frac{\log\left(1 + \frac{2^{a-i}-1}{2^a}\right)}{\log 2^a + \frac{\log\zeta(\alpha)}{\alpha}} = \\
&= 1 + \max_{a\geq k} \frac{\log\left(1 + 2^{-i} - 2^{-a}\right)}{a + \frac{\log\zeta(\alpha)}{\alpha}} \leq \\
&\leq 1 + \frac{\log(1+2^{-i})}{k + \frac{\log\zeta(\alpha)}{\alpha}}
\end{aligned}$$

□

We remark that the bounds obtained here are not especially tight: for example, for $k = 4, i = 2$ and $\alpha = 2$, they predict a redundancy of about 7.3%, while the results of the experimental evaluation show an actual redundancy of about 0.6%. Nonetheless, these bounds serve the purpose of showing that it is possible to achieve redundancy below any constant greater than 1 with the Hybrid Integer Encoding scheme for these two distributions.

3.3 Experimental results

We now report the results of an experimental comparison of the Hybrid Integer Encoding scheme on a sequence of 10^7 i.i.d. integers with geometric (in Figures 3.1, 3.2 and 3.3)

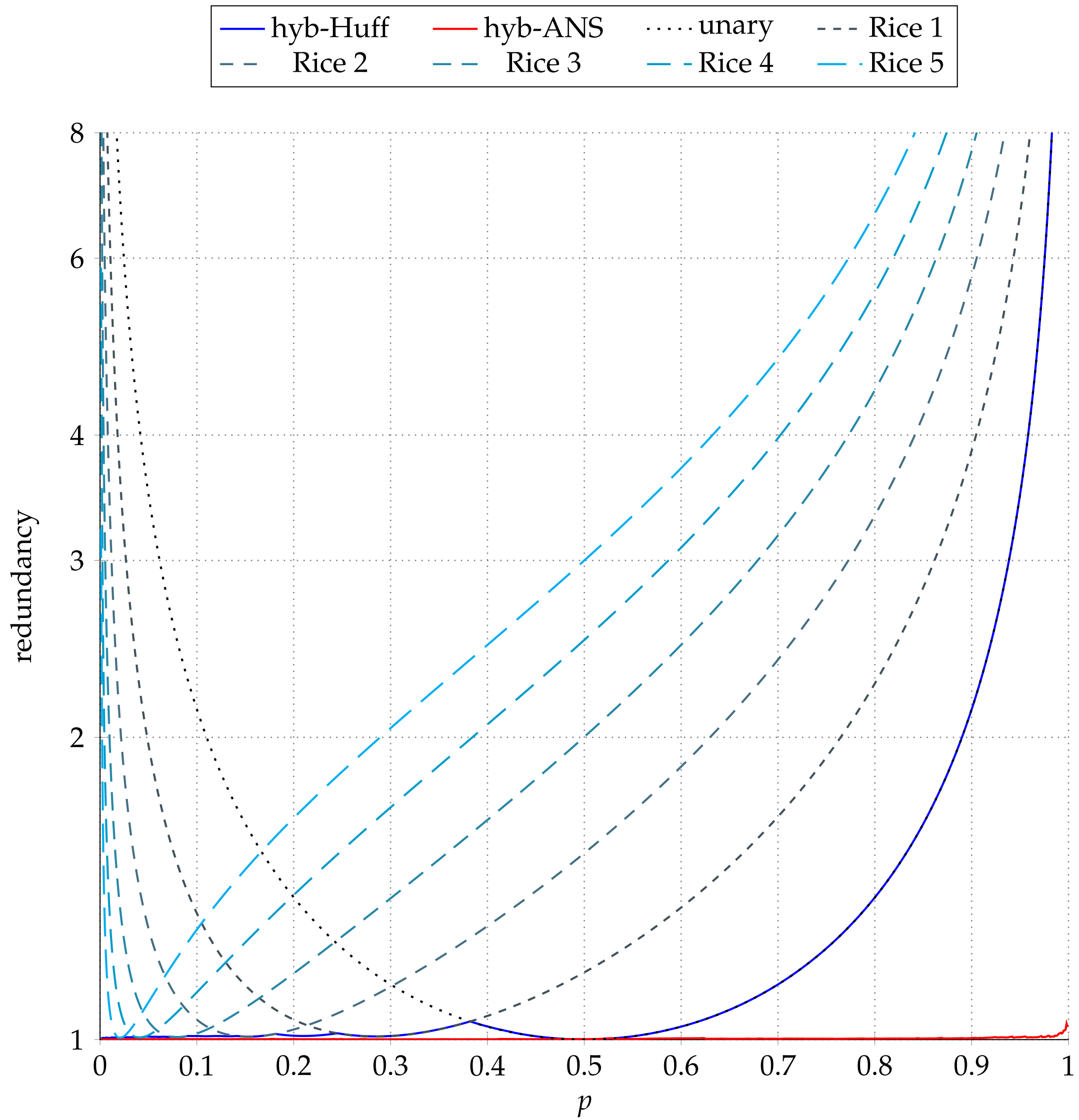


Figure 3.1: Redundancy of Hybrid Integer Encoding, compared to Rice codes, on a geometric distribution for various values of p.

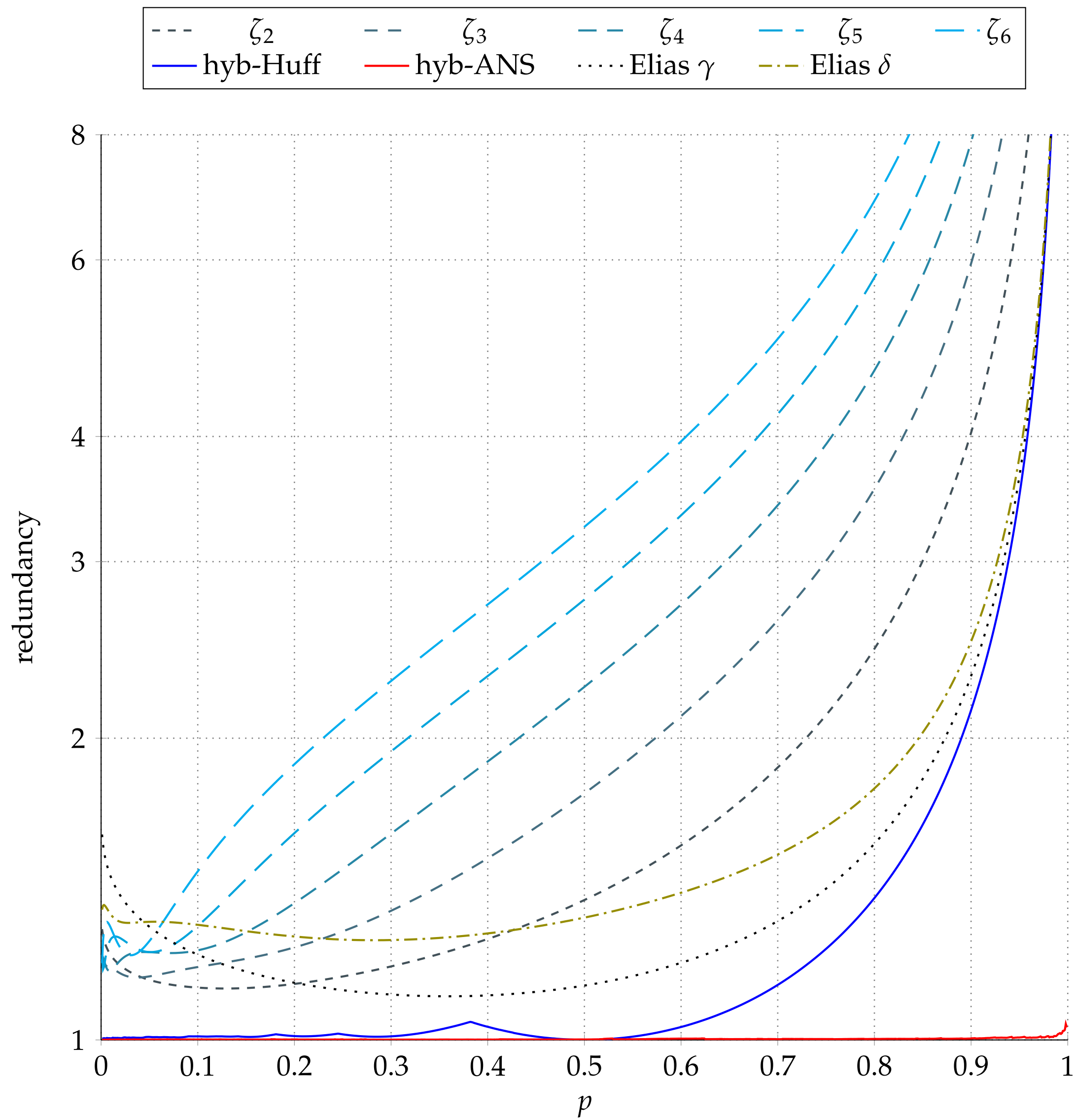


Figure 3.2: Redundancy of Hybrid Integer Encoding, compared to ζ and Elias codes, on a geometric distribution for various values of p.

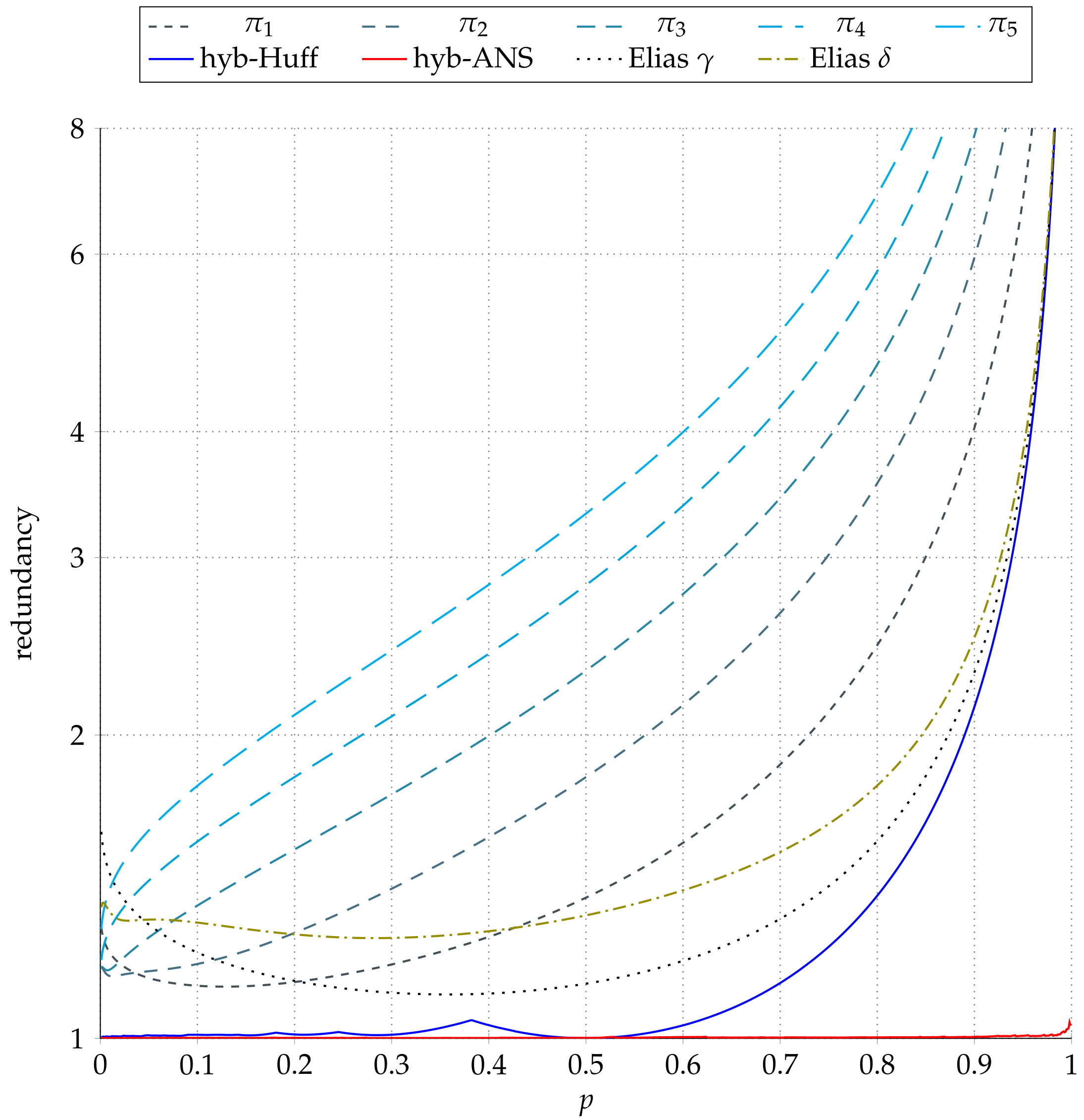


Figure 3.3: Redundancy of Hybrid Integer Encoding, compared to π and Elias codes, on a geometric distribution for various values of p.

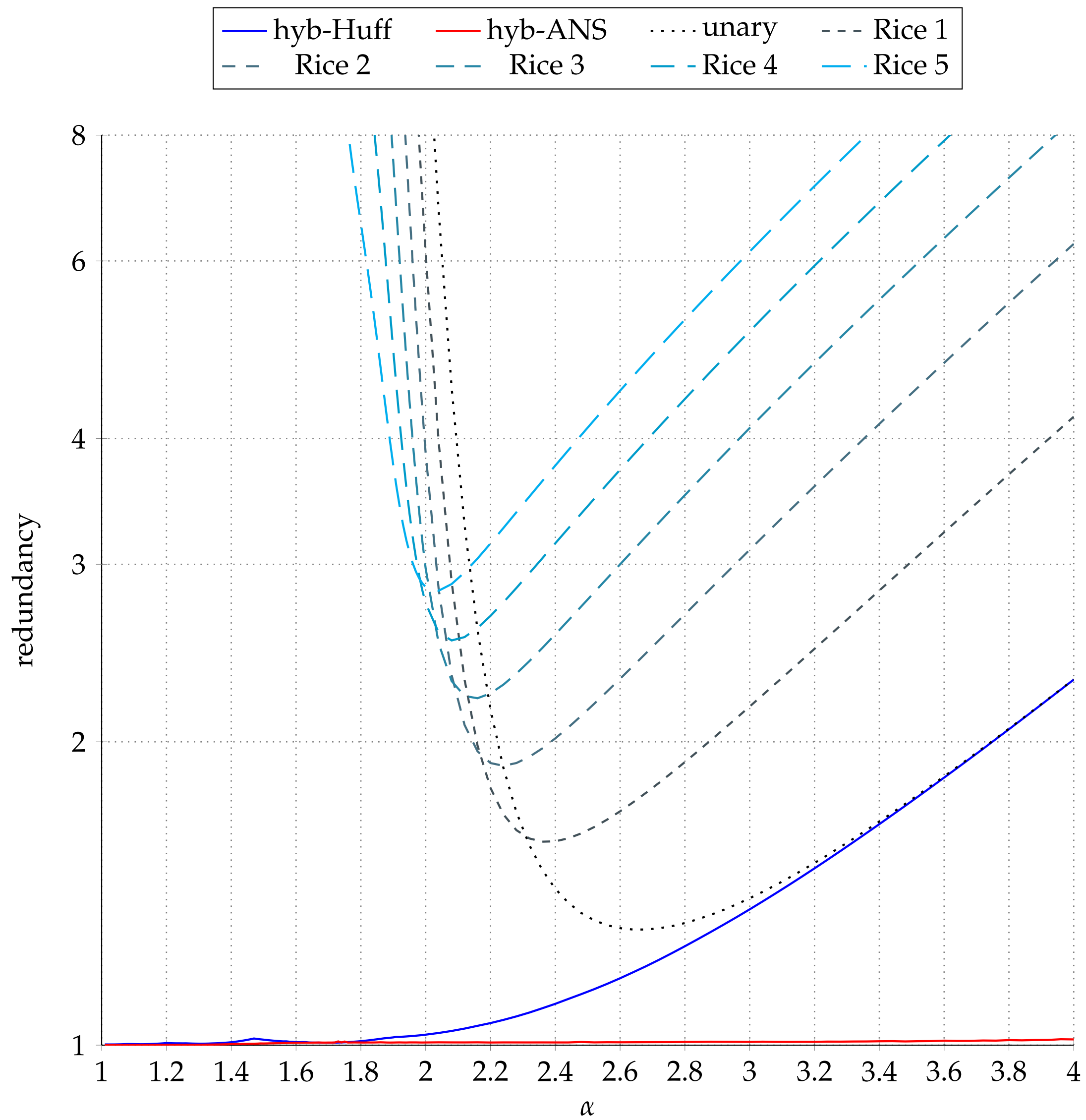


Figure 3.4: Redundancy of Hybrid Integer Encoding, compared to Rice codes, on a Zipf distribution for various values of α.

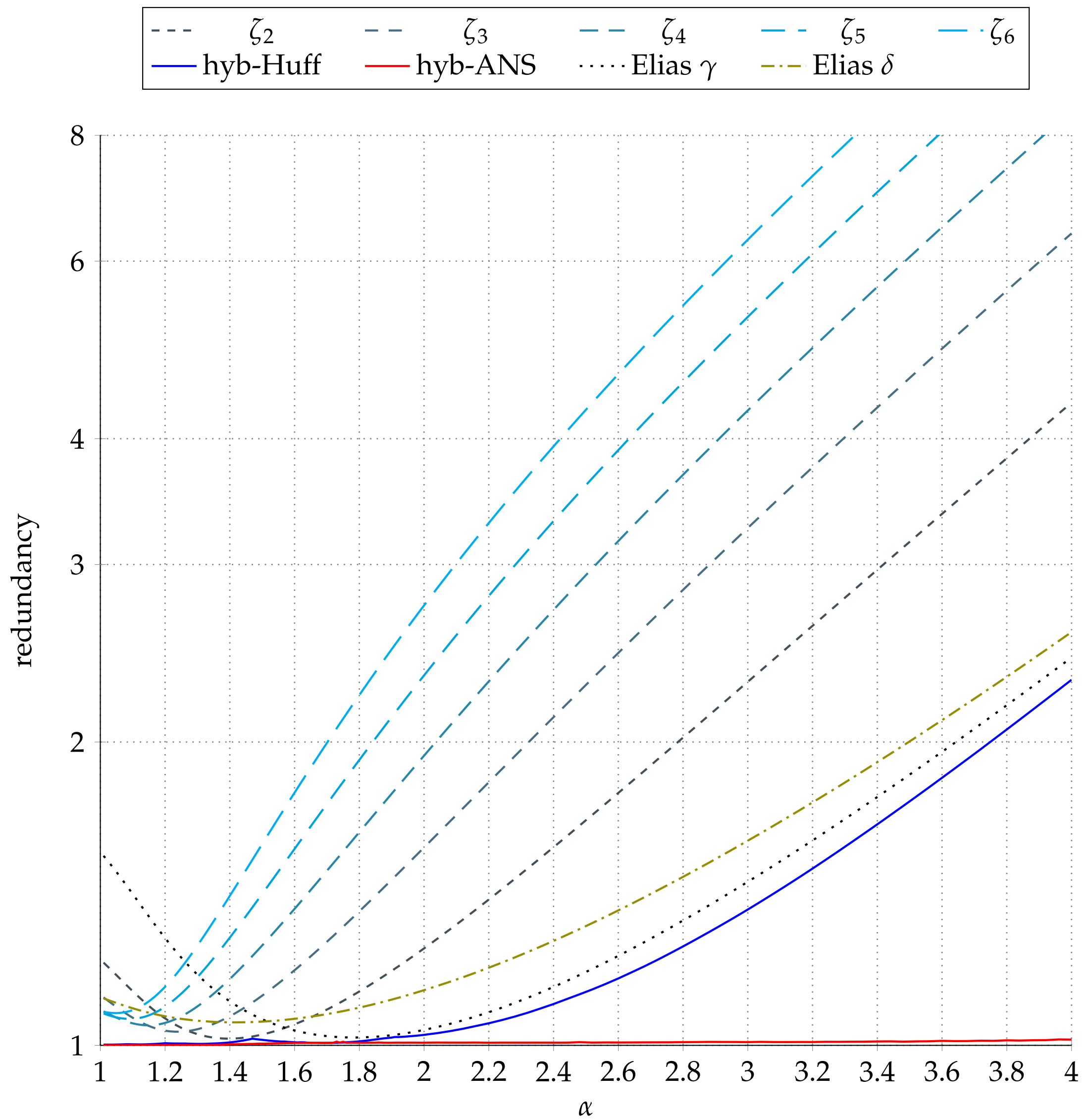


Figure 3.5: Redundancy of Hybrid Integer Encoding, compared to ζ and Elias codes, on a Zipf distribution for various values of α.

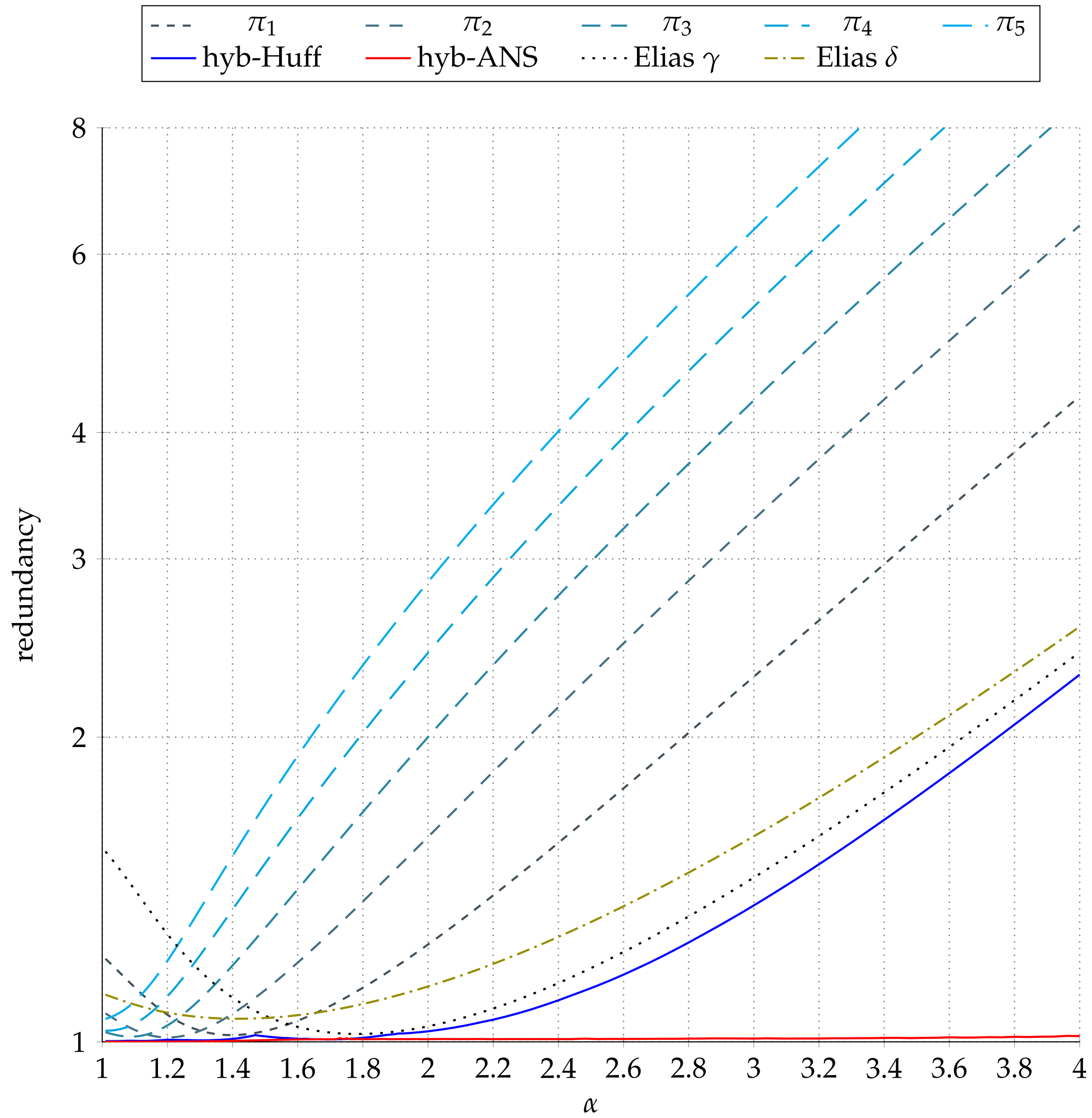


Figure 3.6: Redundancy of Hybrid Integer Encoding, compared to π and Elias codes, on a Zipf distribution for various values of α.

and Zipf (in Figures 3.4, 3.5 and 3.6) distributions respectively, compared with Rice, ζ, π and Elias γ and δ codes. The distributions were cut at the threshold of 2^{32}, i.e. no number higher than 2^{32} was generated.

The Hybrid Integer Encoding scheme was combined with either Huffman coding or with ANS coding for encoding the symbols, using as parameters $k = 4, i = 2, j = 0$.

Hybrid Integer Encoding combined with ANS always has redundancy extremely close to 1, thanks to the possibility of using fractional bits per symbol.

On the other hand, Huffman-based Hybrid Integer Encoding can have higher redundancies for very skewed distributions, but has always the lowest redundancy of all the encoding schemes that use an integer number of bits per value. In particular, it always matches unary and Rice coding those geometric distributions where the codes are optimal.

We also measured the decoding speed of hybrid integer encoding, comparing against the γ and δ coding implementations in `https://github.com/jacobratkiewicz/webgraph`. We found that the extra computation required does not significantly affect decoding speed, as for all the integer coding methods the single-threaded decoding speed is between 6 and 8 nanoseconds per integer on a 32-core AMD 3970X CPU (with SMT enabled) with 256GB of RAM.

A possible implementation of Hybrid Integer Coding is provided in Figure 3.7. This implementation assumes that classes for reading and writing both raw and entropy-coded bits are available.

It can easily be seen that the decoding process just requires simple arithmetic, shifts and bit reading, which are readily available on most modern processors. On more constrained environments, such as low-powered microcontrollers that do not have a barrel shifter, we expect that implementing hybrid integer decoding might be more problematic; nonetheless, such microcontrollers are becoming less and less common.

As for the implementation of the encoder, the situation is similar to the decoder, except for the additional required `FloorLog2` operation; this operation is once again readily available on most modern CPUs (in particular x86 and arm), and can be easily emulated otherwise.

Finally, we compare Hybrid Integer Coding with the method proposed in [78]. For this comparison, we will use base 16 and we will encode just the most significant digit in base 16 using entropy coding, as is done in [78]. This corresponds to, on average, encoding ≈ 2.26 bits below the most significant bit with entropy coding (0 bits when the first digit is 1, 1 bit for $2-3$, 2 bits for $4-7$ and 3 bits for $8-15$); accordingly, the closest corresponding Hybrid Integer Coding setting is $4, 2, 0$, which fully entropy encodes

```
class HybridIntegerCoder {
public:
  void Encode(uint32_t value, BitWriter* writer) const {
    if (value < (1<<k)) {
      writer->EntropyEncode(value);
    } else {
      uint32_t n = FloorLog2(value);
      uint32_t m = value - (1 << n);
      uint32_t high_bits = m >> (n - i);
      uint32_t low_bits = m & ((1 << j) - 1);
      uint32_t token = (1<<k) + ((n - k) << (i + j)) +
                       (high_bits << j) + low_bits;
      uint32_t nbits = n - i - j;
      uint32_t bits = (value >> j) & ((1UL << *nbits) - 1);
      writer->EntropyEncode(token);
      writer->Write(nbits, bits);
    }
  }

  uint32_t Decode(BitReader* br) const {
    uint32_t token = br->EntropyDecode();
    if (token < (1<<k)) return token;
    uint32_t nbits = k - (i + j) +
      ((token - (1<<k)) >> (i + j));
    uint32_t low_bits = token & ((1 << j) - 1);
    token >>= j;
    uint32_t bits = br->ReadBits(nbits);
    uint32_t high_bits = (1 << i) | (token & ((1 << i) - 1));
    return (((high_bits << nbits) | bits) << j) | low_bits;
  }
  uint32_t k = 4;
  uint32_t i = 2;
  uint32_t j = 0;
};
```

Figure 3.7: A possible implementation of Hybrid Integer Coding.

numbers up to 16 and entropy codes the highest two bits after the most significant one for larger numbers.

Results are reported in Figures 3.8 and 3.9. The two schemes result in very similar performance; however, Hybrid Integer Coding has better performance (up to $0.6 - 0.8\%$ better) on geometric distributions with small p, and worse performance (up to 0.3%) for Zipf distributions with very large α, where the effect of the larger average number of entropy coded bits is more pronounced.

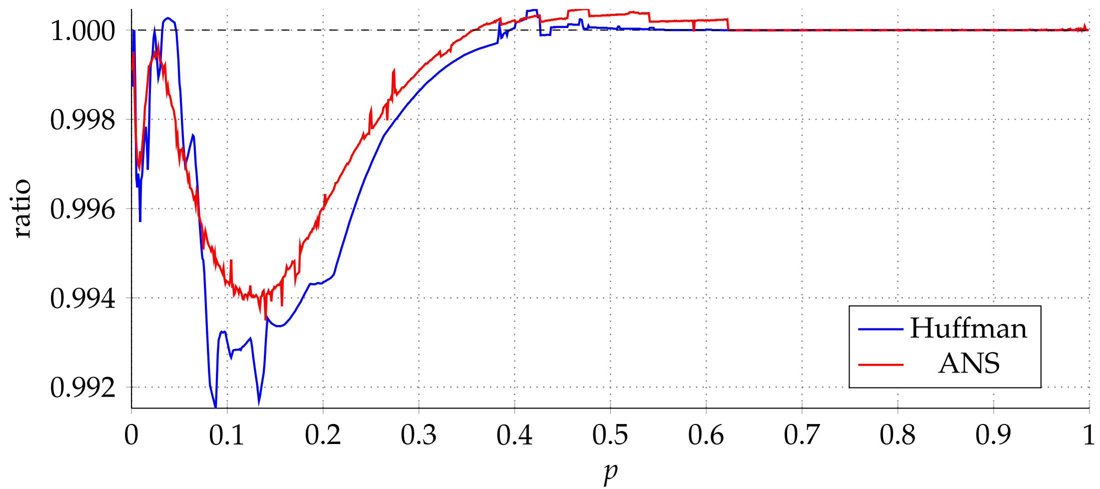


Figure 3.8: Ratio of encoded size between Hybrid Integer Encoding and the method proposed in [78], on a geometric distribution for various values of p. Values smaller than 1 correspond to a smaller size for Hybrid Integer Coding.

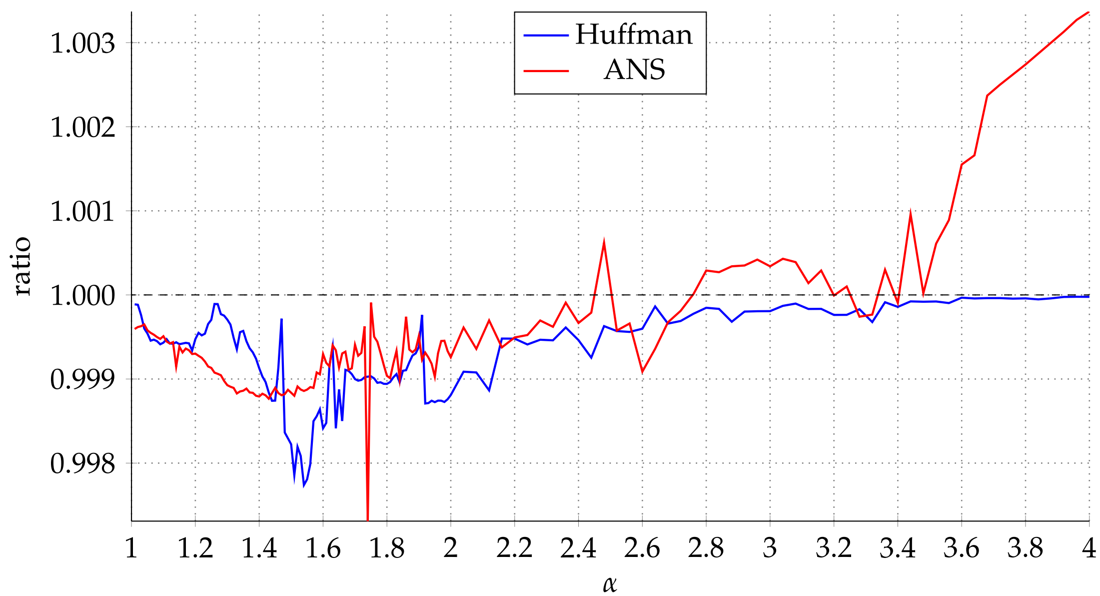


Figure 3.9: Ratio of encoded size between Hybrid Integer Encoding and the method proposed in [78], on a Zipf distribution for various values of α. Values smaller than 1 correspond to a smaller size for Hybrid Integer Coding.

CHAPTER 4

NOVEL TECHNIQUES FOR HIGH-ORDER ENTROPY CODING

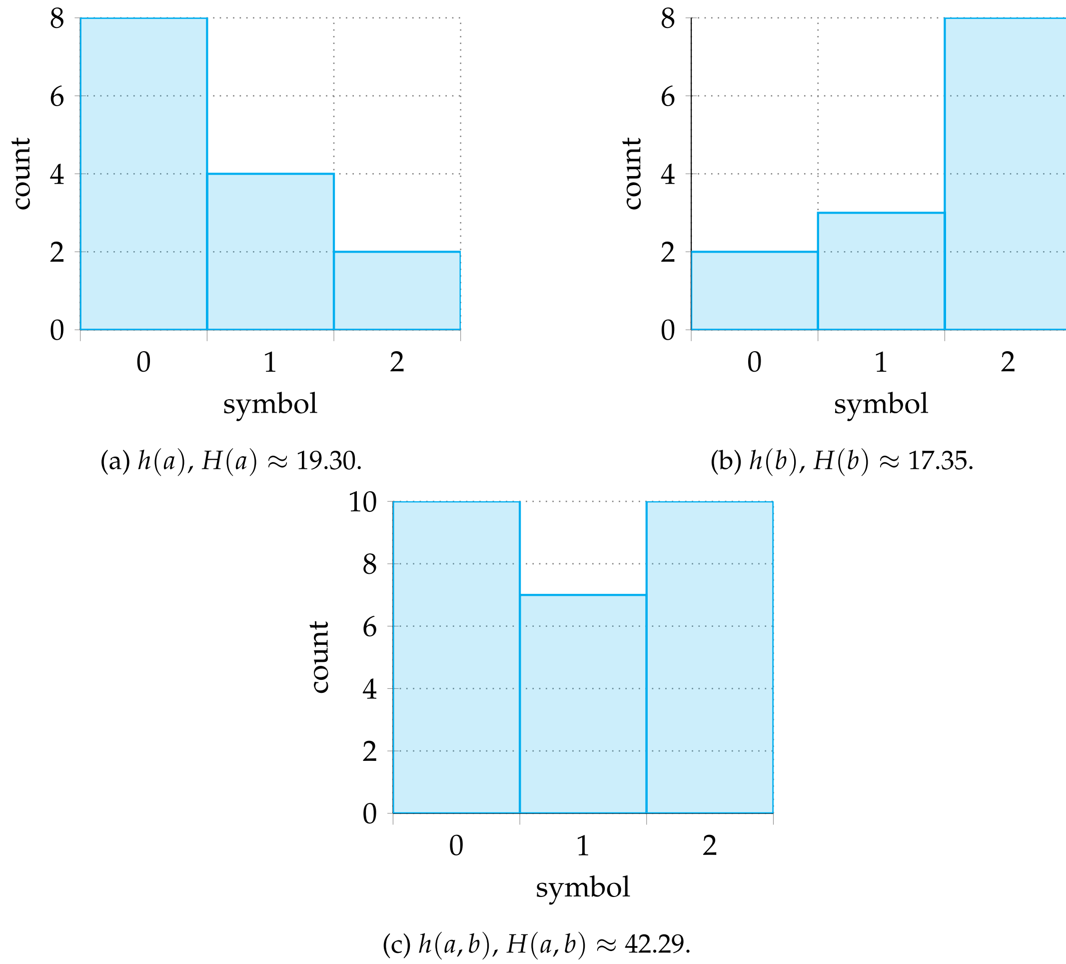
TO MAKE HIGH-ORDER ENTROPY CODING MORE PRACTICAL, we propose two techniques which, to the best of our knowledge, have not been explored in the literature: *context clustering* and *decision-tree-based context modeling*.

In this chapter, we will denote *context space* ($\mathcal{C}$) the space of all the possible contexts for a symbol, Σ the set of all possible symbols, $\mathcal{P}_\Sigma$ the set of all the probability distributions on Σ, and $c(s)$ the context associated with a given symbol s.

We will also make the simplifying assumption that encoding the probability of a given symbol requires a fixed number of bits b, and thus encoding a probability distribution requires $|\Sigma|b$ bits. In most real-world encoding schemes, this is not the case, but the rest of the chapter is substantial unchanged even with more complex distribution cost models; thus, for simplicity of exposition, we will use this trivial cost model.

The problem of communicating the probability distributions of each symbol can be seen as the problem of encoding a function $d : \mathcal{C} \to \mathcal{P}_\Sigma$; in the most general case, corresponding to trivial order-k entropy coding, encoding d requires $O(|\mathcal{C}| \cdot |\Sigma| \cdot b)$ bits.

In the example from Section 2.5, order-2 entropy coding of a binary file, $\Sigma = \{0, \dots, 255\}$, $\mathcal{C} = \Sigma^2$ and $c(s)$ is the pair formed by the two bytes preceding s. Assuming $b = 12$, as is often the case when using arithmetic coding, the cost of transmitting the dis-



(a) $h(a)$, $H(a) \approx 19.30$.

(b) $h(b)$, $H(b) \approx 17.35$.

(c) $h(a,b)$, $H(a,b) \approx 42.29$.

Figure 4.1: An example of histograms for the two context values a and b on the alphabet $\{0,1,2\}$.

tributions in the most trivial way would be approximately $|\Sigma|^2 \cdot |\Sigma| \cdot 12 = 201\,326\,592 \approx$ 25 megabytes, making order-2 entropy coding unfeasible for files that are not extremely large.

4.1 Context clustering

The idea behind context clustering is simple: as distributions are significantly expensive, we can reduce the number of distributions to encode by splitting d in two parts:

$$cl : \mathcal{C} \to \{0, \ldots, k-1\} \qquad dc : \{0, \ldots, k-1\} \to \mathcal{P}_\Sigma$$

$$d = dc \circ cl$$

Algorithm 2: Context clustering algorithm. Returns an array of context clusters.

```
c_m ← c ∈ C such that H(c) is maximum;
S ← {c_m};
while |S| < min(k, |C|) do
    d_x ← min_{c∈S} D(h(c), h(x));
    c_m ← c ∈ C \ S such that d_c is maximum;
    S ← {c_m} ∪ S;
G ← [{c} for c ∈ S];
foreach c ∈ C \ S do
    m ← i such that D(h(c), h(G[i])) is minimum;
    G[i] ← G[i] ∪ {c};
return G;
```

The values $0, \ldots, k-1$ effectively correspond to a *cluster* of context values, and distributions are assigned to clusters instead of specific context values. It is easy to see that the *cl* function can be encoded using $O(|\mathcal{C}| \cdot \log k)$ bits, and the *dc* function requires $O(k \cdot |\Sigma| \cdot b)$ bits. Thus, this scheme achieves a significant reduction of distribution cost when $|\mathcal{C}|$ is large (as typically $\log k \ll |\Sigma| \cdot b$).

We will denote with $h(c)$, for $c \in \mathcal{C}$, the histogram of values x such that $c(x) = c$. We denote by $h(a_1, \ldots, a_n)$ the histogram obtained by pointwise adding the counts of all the symbols in $h(a_1), \ldots, h(a_n)$; finally, we will denote by $H(a_1, \ldots, a_n)$ the cost in bits of encoding all the symbols that have $a_1, \ldots, a_n$ as a context, i.e. the entropy of the distribution defined by $h(a_1, \ldots, a_n)$ multiplied by its total symbol count.

As an example, Figure 4.1 shows histograms for two distinct context values a, b on a 3-symbol alphabet. We have that

$$H(a) = 8\log\frac{14}{8} + 4\log\frac{14}{4} + 2\log\frac{14}{2} \approx 19.30$$

$$H(b) = 2\log\frac{13}{2} + 3\log\frac{13}{3} + 8\log\frac{13}{8} \approx 17.35$$

$$H(a,b) = 10\log\frac{27}{10} + 7\log\frac{27}{7} + 10\log\frac{27}{10} \approx 42.29$$

Once *cl* is known, how to choose *dc* is obvious: it is simply, for each cluster, the histogram obtained by merging the histograms that belong to the cluster.

4.1.1 Heuristic algorithm

We will now present a heuristic algorithm to choose *cl* for a fixed value of k, with running time $O(|\Sigma| \cdot |\mathcal{C}| \cdot k)$.

We first define a distance between two histograms $h(A)$, $h(B)$ as follows:

$$D(h(A), h(B)) = H(A \cup B) - H(A) - H(B) \geq 0$$

Clearly, D can be computed in $O(\Sigma)$ time. From the example above, we can see that $D(h(a), h(b)) = H(a, b) - H(a) - H(b) \approx 5.64$.

The algorithm, detailed in Algorithm 2, proceeds with the following steps:

1. Pick the element c_1 from $\mathcal{C}$ such that $H(c_1)$ is maximum.
2. Pick the element c_i from $\mathcal{C} \setminus \{c_1, \ldots, c_{i-1}\}$ such that $\min_{j<i} D(h(c_i), h(c_j))$ is maximum.
3. Repeat the previous step until k elements have been chosen.
4. Assign each element $c \in \mathcal{C}$ to cluster i if $D(c, c_i)$ is minimum among the choices of i; replace c_i with $c_i \cup \{c\}$.

As described, the second step of the algorithm needs to evaluate $O(k^2|\mathcal{C}|)$ times the function D. However, as the minimum is computed over an increasing set that has maximum size k, it is easy to see that only $O(k|\mathcal{C}|)$ evaluations are necessary. Indeed, we can keep track, for every $c \in \mathcal{C}$, of the minimum of $D(h(c), h(c_i))$ for every selected c_i, and compute the distance of every $c \in \mathcal{C}$ from each newly selected histogram and update the minimum if necessary.

The algorithm described here is reminiscent of a deterministic version of the initialization step of `k-means++` [6]. However, the D function is not one of the common distance metrics ($||x - y||^l$) that are known to guarantee good results for `k-means++`, and the problem being solved is not quite equivalent to *k-means clustering*. Nonetheless, given the similarities of the two problems, it seems reasonable to state the following conjectures:

Conjecture 9. *It is NP-complete to decide whether there exist two functions cl and dc such that*

- *The cost of encoding the input text using* $(dc \circ cl)(c(x))$ *as the distribution for encoding* x *is below a threshold* t.
- *cl only produces* k *distinct values.*

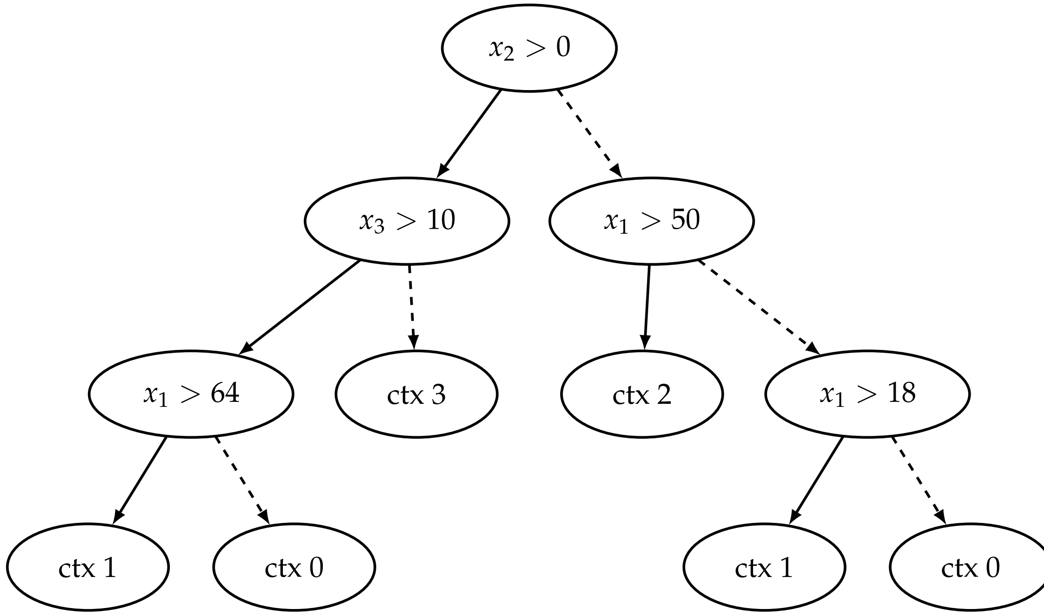


Figure 4.2: An example of a context tree CT. Dashed arrows correspond to branches that are taken if the condition is false; x_n corresponds to the value of the n-th dimension of the context value. Hence, the root of the tree in the figure will choose the left child if dimension 2 of the context value is strictly positive, and the right child otherwise.

Conjecture 10. *There is a randomized scheme for selecting a new element from $\mathcal{C}$ that achieves in expectation a $O(\log k)$ approximation of the true optimum of context clustering.*

The second conjecture follows from the fact that the initialization step of `k-means++` is a $O(\log k)$ approximation, as shown in [6].

4.2 Decision-tree-based context modeling

When $\mathcal{C}$ is very large, encoding the cl function can still be prohibitively expensive. We now present another representation of the cl function that has the interesting characteristic of not requiring space proportional to $|\mathcal{C}|$.

This technique assumes, as is often the case, that $\mathcal{C}$ is a product of intervals of integers $I_1 \times \cdots \times I_n$; this then produces a decomposition of $c \in \mathcal{C}$ given by $c = (x_1, \ldots, x_n)$.

We define a binary tree CT where:

- Each inner node contains an index i and a threshold t.

- Each leaf node contains a clustered context ID, similarly to the output of cl.

An example is given in Figure 4.2.

Then, we can define a function $t_{CT}(c)$ that is computed by traversing the tree; when visiting an inner node, the traversal proceeds left if $x_i > t$, and proceeds right otherwise. When a leaf is reached, its clustered context ID is produced as the output of $t_{CT}(c)$.

Note that more complex decision nodes are possible, for example nodes that contain a weight vector $\mathbf{w} \in \mathbb{R}^n$ and pick the right or left child based on the value of $\mathbf{w} \cdot \mathbf{x}$.

This context modeling technique can be seen as a generalization and modification of the Context Tree Weighting scheme in [102], which defines an adaptive scheme for encoding probabilities of bits. In particular, a binary tree is maintained during the encoding (or decoding) procedure, where level k of the tree is traversed according to the k-th previous bit in the encoded stream. The probability used for encoding the next bit is then given by the product of all the probability along the root-leaf path; moreover, the probabilities are modified during the coding process.

The scheme proposed here differs in multiple ways:

- It is *static* and not adaptive, i.e. probabilities don't change over time; this requires transmitting the tree itself together with the data, but generally allows a faster decoding process.

- It is meant to encode integers and not just single bits, thus requiring full probability distributions; on the other hand, this significantly reduces the number of entropy decoding operations necessary.

- It is more general, allowing arbitrary decision nodes instead of just using the k previous bits. In particular, it allows having nodes on the same level that decide based on different conditions.

Let $|CT|$ denote the number of inner nodes of the tree, k denote the number of possible context IDs, and s denote the maximum size of the intervals that compose $\mathcal{C}$, i.e. $|I_i| \leq s$. Then the function $d = dc \circ t_{CT}$ can be encoded using $O(|CT| \cdot (\log n + \log s) + (|CT| + 1)\log k + k \cdot |\Sigma| \cdot b)$ bits, where the first term is given by the cost of encoding the inner nodes, the second term by the cost of encoding the leaves, and the third term by the cost of encoding the dc function.

It is thus important to consider the problem of constructing the decision tree that minimizes the resulting data encoding cost among those that have at most a given number of nodes and of different context IDs.

Given the multiple hardness results for constructing optimal cost decision trees in the literature [66, 33], we state the following

Conjecture 11. *Determining whether there exists a decision tree CT and a function dc such that*

- *The cost of encoding the input text using $(dc \circ t_{CT})(c(x))$ as the distribution for encoding x is below a threshold t*
- *The number of inner nodes of the tree is at most n_t*
- *The leaves of CT contain at most k distinct values*

is NP-complete, and it is still NP-complete if we take $k = \infty$.

We now give algorithms for the optimization version of the above problem in some interesting special cases. In the rest of this section we will assume that the n_h non-empty histograms for the context space are given in a sparse form, i.e. that it is possible to list all the values of c for which the corresponding histogram of values is non-empty, and to retrieve the histogram corresponding to a given c in constant time. This could be done, for example, by keeping the values of c with non-empty histograms in a cuckoo hash table [82].

4.2.1 Optimal algorithm for $n = 1, k > n_t$

Theorem 12. *The optimal decision tree problem can be solved in $O(s^2(|\Sigma| + n_t))$ time when $n = 1$ and $k > n_t$.*

Proof. We will assume without loss of generality that in this case $\mathcal{C} = \{0, \ldots, s-1\}$. We first compute, for every $0 \leq i < j \leq s$, $H_{i,j} = H(i, \ldots, j-1)$. By computing prefix sums of histograms, this can be done in $O(s^2|\Sigma|)$ time.

We now find a sequence $c_1, \ldots, c_k$ of split points such that $k \leq n_t$ and $H_{0,c_1} + \cdots + H_{c_{k-1},s-1}$ is minimum. It is then straightforward to build an optimal decision tree that satisfies the requirements of the problem from this sequence, by using each c_i as thresholds for the decisions.

We solve this problem via dynamic programming; Algorithm 3 contains a more detailed description. In particular, let $C(i, j)$ be defined as the minimum cost of encoding the symbols corresponding to context values $\{0, \ldots, i-1\}$ using at most j split points. Then

Algorithm 3: Optimal decision tree polynomial-time algorithm for $n = 1$ and $k > n_t$.

```
for i = 0 ... s − 1 do
    hist ← ∅;
    H_{i,i} ← 0;
    for j = i + 1 ... s do
        hist ← hist ∪ {j − 1};
        H_{i,j} ← H(hist);
C ← array with s × n_t entries;
for i = 0 ... s do
    C(i, 0) ← H_{0,i};
for j = 0 ... n_t do
    C(0, n_t) ← 0;
for j = 1 ... n_t do
    for i = 1 ... s do
        C(i, j) ← min_{k<i} (H_{k,i} + C(k, j − 1));
splits ← ∅;
p ← s;
n is such that C(s, n) is minimum;
while n > 0 and s > 0 do
    k is such that k < s and H_{k,p} + C(k, n − 1) is minimum;
    splits ← {k} ∪ splits;
    p ← k;
    n ← n − 1;
return a decision tree using the values in splits as decision nodes;
```

- $C(i, 0) = H_{0,i}$, as this corresponds to encoding all the first i contexts together.
- $C(0, j) = 0$, as this corresponds to encoding an empty sequence.
- $C(i, j) = \min_{k<i} (H_{k,i} + C(k, j-1))$: the function inside the minimum is the cost of having the last split point before position i in position k, which covers all possibilities.

The minimum cost is then given by $\min_{j \le n_t} C(s, j)$, and the set of splits that produces this minimum cost can be easily reconstructed by backtracking through the optimal choices done while computing the dynamic programming table.

The dynamic programming table has $O(s \cdot n_t)$ states, and computing the value of each state can be done in $O(s)$ time. Thus, this part of the algorithm runs in $O(s^2 \cdot n_t)$

time, proving the thesis. □

4.2.2 Optimal algorithm for $n_t = 1$

Theorem 13. *The optimal decision tree problem can be solved in $O(n(s + n_h)|\Sigma|)$ when $n_t = 1$.*

Proof. We first observe that, as the decision tree can have only one node, we can reduce ourselves to the case $n = 1$ without loss of generality. The optimal solution for the general case can be then obtained by running the given algorithm along each dimension of $\mathcal{C}$ separately, and taking the tree of minimum cost among the n optimal trees for each dimension.

Note that the cost of considering a single dimension at a time is $O(n_h|\Sigma|)$, corresponding to the cost of merging together all the histograms that correspond to the same value along that dimension.

Along each dimension, by applying the dynamic programming algorithm we can immediately get an algorithm with cost $O(s^2|\Sigma|)$.

However, by observing that a single split divides the context space in a prefix and a suffix, we can limit ourselves to computing $H_{0,j}$ and $H_{j,s}$ for $0 < j < s$; the optimal solution will then be the j that minimizes $H_{0,j} + H_{j,s}$, for a total time cost of $O(s|\Sigma|)$. □

4.2.3 Heuristic algorithm in pseudolinear time for $n = 1$

In some cases, s may be too large for the proposed algorithm for $n = 1$ to be practical. Thus, we present an heuristic algorithm for the $n = 1$ case that runs in pseudolinear time (a detailed implementation is available in Algorithm 4).

- Create a sequence of intervals $I_i = [i, i + 1)$ for $i = 0, \ldots, s - 1$.
- Keep all pairs of adjacent intervals in a min-heap, sorted by the cost increase of merging the pair together (i.e. by $D(h(I_i), h(I_{i+1}) = H(I_i\ cup I_{i+1}) - H(I_i) - H(I_{i+1}))$.
- Merge together the pair of intervals of lowest cost, and update the cost of merging the new interval with its neighbors.
- Repeat the previous step until at most $n_t + 1$ intervals are left.
- Build a tree using the boundaries of those intervals as thresholds for decision nodes.

Algorithm 4: $s \log s$ heuristic for $n = 1$ and $k > n_t$.

```
H ← empty min-heap;
splits ← [1, ..., s − 1];
for i = 0 ... s − 2 do
    C_{i,i+2} ← H([i, i + 2));
    insert C_{i,i+2} in H;
while |splits| > n_t do
    C_{a,b} ← pop_heap(H);
    erase the only i in splits such that a < i < b;
    if ∃x ∈ splits : x < a then
        x ← max x ∈ splits : x < a;
        erase C_{x,i} from H;
        C_{x,b} ← H([x, b));
        insert C_{x,b} in H;
    if ∃y ∈ splits : y > b then
        y ← min y ∈ splits : y > b;
        erase C_{i,y} from H;
        C_{a,y} ← H([a, y));
        insert C_{a,y} in H;
return a decision tree using the values in splits as decision nodes;
```

It can easily be seen that the computational cost of this algorithm is given by $O(s|\Sigma|)$ for computing the costs of histogram pairs plus $O(s \log s)$ for maintaining the heap, for a total cost of $O(s(\log s + |\Sigma|))$.

4.2.4 Heuristic for the general case

We conclude with a simple greedy algorithm to solve the general case of the optimal decision tree problem; the algorithm proceeds by recursively splitting the context space in two by finding the best single split using the algorithm described in Theorem 13. When a candidate split is found, it is put into a max-heap that uses the gain of the split as a key, and a tree is built by recursively expanding the most promising split. The algorithm stops when the desired number of nodes has been reached.

We remark that, in practical usage, n_t is not an actual problem constraint, but we seek to find the tree that achieves the best balance between compression gains and cost for encoding the tree itself. To this end, we can modify the previous algorithm slightly to make the recursion stop when the gain of performing a split is too small to offset the increased decision tree cost, rather than when a given number of nodes is reached. This

Algorithm 5: Decision tree construction heuristic.

```
Function FindTree(𝒞):
    x_p, v ← best single-node condition from Theorem 13;
    C_t ← cost of encoding with a leaf-only tree;
    C_s ← cost of encoding with a tree with x_p > v condition;
    if C_t − C_s > thres then
        root.dimension ← p;
        root.thres ← v;
        𝒞_l ← 𝒞 ∩ {x_p > v};
        root.left ← FindTree(𝒞_l);
        𝒞_r ← 𝒞 ∩ {x_p ≤ v};
        root.right ← FindTree(𝒞_r);
        return root;
    return leaf;
```

algorithm is presented in Algorithm 5, and used in the experiments in Section 7.4.

Part 2

Graph compression

CHAPTER 5

COMMON TECHNIQUES FOR GRAPH COMPRESSION

COMPRESSION OF LARGE GRAPHS is a well-studied problem, and multiple different techniques have been applied. This chapter gives an overview of the main techniques that have been used in the literature for graph compression.

We will broadly classify compression techniques in *primitive* and *derivative*, where the second class is composed mainly of techniques that apply transformations on the input graph to obtain one or more different graphs that are then compressed with existing methods.

We can identify the following main groups of techniques for primitive graph compression:

- Raw link encoding: schemes that encode adjacency lists directly, without exploiting correlations between separate lists.
- Grammar- and dictionary-based: schemes that exploit recurring patterns by replacing them with a single object, possibly in a recursive way.
- Class-tailored: schemes that are optimal for a specific class of graphs, such as trees or planar graphs.

- Tree-based: schemes that build a tree that represents the adjacency matrix of the graph and encode the tree efficiently.
- Copying models: schemes that use other adjacency lists as a reference for encoding a given list, or encode together multiple lists with their local variations.

In the derivative approaches, we identify two main groups:

- Graph decomposition: schemes that split the input graph into multiple graphs that are encoded separately, but are better compressible.
- Graph permutation: schemes that permute the order of the nodes in the graph to achieve better compressibility.

For an overview of graph compression approaches that are focused on graphs with very specific structures, approaches for compression of labeled graphs where the links and labels are very heavily structured (such as RDF graphs), and application of graph compression in the context of, i.e., graph databases, we refer the reader to [12].

5.1 Raw link encoding

Raw link encoding is one of the simplest ways to represent a graph. The well-known Compressed Sparse Row matrix representation format may be considered one such representation, in which every destination node of outgoing edges of each node is represented using $\lceil \log n \rceil$ bits, all such lists are concatenated in the order in which nodes appear in the graph, and an auxiliary structure with n $\lceil \log n \rceil$-bit entries is used to store the degree of each node. This representation uses a total of $(n+m)\lceil \log n \rceil$ bits, and can be considered a baseline for compressed representations.

The Connectivity Server [14, 27, 100] can be seen as the first substantial improvement over the CSR scheme mentioned above, applying the well-known technique of gap coding to the list of outgoing edges of a given node. More precisely, instead of storing the index of the destination node, the Connectivity Server stores the difference between the index of destination node $i+1$ and the index of destination node i, after sorting the edges in increasing order of destination node. This typically results in a significantly reduced magnitude of numbers to be encoded, which benefits integer coding schemes that use a variable number of bits per edge.

Another technique to reduce the magnitude of the encoded numbers is explored in [54], which observes that often links in large graphs cross a small number of nodes;

a possible encoding scheme would thus encode the difference between the end node and the source node. The scheme proposed in [54] uses Huffman codes for short edges, falling back to fixed-length 16 or 32 bit codes for longer edges.

In [7], an improvement over Connectivity Server was achieved thanks to the observation that many parameters in web graphs follow a Zipf distribution; this observation has been exploited in multiple compression schemes. Moreover, it was observed to be true in other kinds of networks too, like social networks or brain connectivity networks [29].

Another important observation for compression purposes comes from [89]: web graphs often have the properties of *locality* and *similarity*. In other words, if one considers nodes in the web graph sorted by URL, it will often be the case that many links point to nearby pages, and that nearby pages point to the same destinations. The first property in particular explains the success of gap coding, while similarity provides a good theoretical foundation for copying models. As with the power-law distribution characteristics of web graphs, locality and similarity are also present in other classes of large graphs [34].

More recently, LogGraph [13] proposes a couple of variations on the raw link encoding framework, using per-node fixed length integer representations for high performance access, or storing out-edges in a multipart representation that is composed of a shared prefix, common to multiple edges, and a per-edge suffix. This approach is also combined with permutation schemes to maximize the compression savings.

5.2 Grammar- and dictionary-based

Grammar- and dictionary-based approaches exploit the occurrence of common patterns in large-scale graphs.

Inspired by the text compression algorithm Re-Pair [68], [36] proposes to exploit common pattern in the adjacency lists of a graph by considering adjacency lists as a sequence of symbols, and then replacing the most common pair of consecutive symbols with a single new symbol, memorizing in a dictionary the replacement rule. The process is repeated until no pair appears at least twice; the resulting dictionary is then compressed to improve storage requirements.

This approach does not directly exploit the linked structure of graphs. GraphRePair [74] overcomes this limitation by applying the same kind of approach to pairs of *edges*, instead of entries in the adjacency list.

The representations described above are recursive in nature, representing the graph in a tree-like structure that contains the edges as leaves. In contrast, the *virtual nodes* approach of [28] proceeds by doing a structural modification on the graph itself: for

any virtual node v, the presence of a pair of edges (a, v) and (v, b) corresponds to the presence of edge (a, b) in the original graph, so that each virtual node represents as many edges as the product of its in- and out-degrees. The compression algorithm proceeds by finding complete bipartite subgraphs, representing them with virtual nodes, and finally compressing the resulting graph using Huffman coding.

This kind of structural modifications of the graph also have the attractive property of being able to run many algorithms directly on the compressed representation of the graph [62], often with a computational cost proportional to the size of the compressed representation instead of the size of the graph itself.

5.3 Class-tailored

Multiple algorithms have been proposed to compress specific families of graphs down to the corresponding information-theoretical lower bounds.

Algorithms are known to compress a tree of n nodes using $2n + o(n)$ bits, while supporting efficient navigation. More details about tree compression are given in Subsection 5.3.1.

Other algorithms are known for compressing planar graphs, graphs of bounded genus, or k-page graphs in $O(n)$ space. As a generalization, [17] proposes a scheme that uses $O(n)$ bits for any c-separable class of graphs ($c < 1$), i.e. any class of graphs such that there exists a set of $O(n^c)$ nodes that creates two connected components of approximately equal size when removed. This result was then improved to optimal compression in [18].

Other approaches exist that are able to compress *any* graph, but get close to optimality on specific classes. For example, [48] generalizes a compressed tree representation (LOUDS [59]) to be able to compress any graph, but the compression density degrades quickly as the number of edges in the graph increases.

Another interesting approach is the one in [75], which is particularly suited for Eulerian graphs. More precisely, it is based on storing a linearized version of the graph; this representation has the same length as the number of edges of the graph if the graph is Eulerian, and is longer otherwise. Moreover, it allows retrieving both forward and backward edges for the same computational cost.

5.3.1 Compression of trees

We consider two kinds of trees: *labeled* trees, and *unlabeled* trees.

As proven in [32], the number of labeled trees on n nodes is n^{n-2}. Moreover, there is a simple bijection [86] between a tree and its *Prüfer sequence*, a sequence of $n-2$ numbers in $[0, n)$ that is obtained by iteratively removing the lowest-label leaf in the tree and appending the label of its parent to the sequence. It follows that it is possible to compress a labeled tree in $(n-2)\log n$ bits. Other compressed representations of labeled trees exist, such as [85], supporting various queries directly on the compressed representation; however, an in-depth study of this topic is out of the scope of this thesis.

Regarding unlabeled trees, we consider the case of *ordered* trees, which is the class of trees that have a distinguishable *root* node; we also consider two trees to be different if the order of child subtrees differs.

There is a vast amount of literature [59, 79, 88, 11, 60, 93] on the compressed representation of such trees, using $2n + o(n)$ bits (which, as we will shortly see, is asymptotically optimal). For the purposes of this thesis, it is sufficient to see that there is a simple bijection between an ordered tree on n nodes and a sequence of *balanced parentheses* of $2n-2$ parentheses, i.e. sequences where every open parenthesis has a corresponding closed parenthesis according to the usual rules.

Indeed, consider a depth-first traversal of the tree, that visits children of a node in left-to-right order. During this visit, we add an open parenthesis to our sequence whenever we visit a new node, and a closed parenthesis whenever we are done visiting a subtree and move back to the parent node. It is clear that the sequence of parentheses obtained this way is balanced, and that any sequence of balanced parentheses corresponds in turn to an ordered tree.

Hence, ordered trees on n nodes can be compressed in $2n-2$ bits in linear time. Since any unlabeled tree can be transformed in an ordered tree, the same is true for any unlabeled tree.

From the bijection between sequences of balanced parentheses and trees, it follows that the number of ordered trees on n nodes is equal to the n-th Catalan number, hence the entropy of a n-node ordered tree chosen uniformly at random is

$$\log\left(\frac{1}{n+1}\binom{2n}{n}\right) = 2n + O(\log n)$$

The balanced parenthesis representation is thus asymptotically optimal.

$$
\begin{array}{cccc|cccc|cccc|cccc}
0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

Figure 5.1: Example of an adjacency matrix with separation lines that represent the submatrices for k^2-trees. Shaded areas represent submatrices that are filled with 0s and, hence, do not require further divisions. Only the first two levels of subdivisions are shown.

5.4 Tree-based

Another well-known approach to graph compression are k^2-trees [25], which use a succinct representation of a bidimensional k-tree on the adjacency matrix of the graph. More precisely, at each level of the tree the current section of the matrix is split into $k \times k$ axis-aligned submatrices; submatrices that contain no edges are encoded as a 0, while submatrices that contain edges are encoded as a 1, followed by the representation of the submatrix again as a k^2-tree. The resulting tree is then encoded using a succinct tree representation like the one described in Subsection 5.3.1. k^2-trees allow both forward and backward neighbourhood queries, as well as querying for the existence of single edges. This tree-based approach implicitly exploits similarity of adjacency lists by sharing the most significant bits implicitly in the higher levels of the tree. An example of a k^2-tree can be found in Figure 5.2.

In [26], k^2-trees have been improved in multiple ways:

- The value of k is not kept constant, but changed level-by-level to adapt to different characteristics of the graph at different granularities.

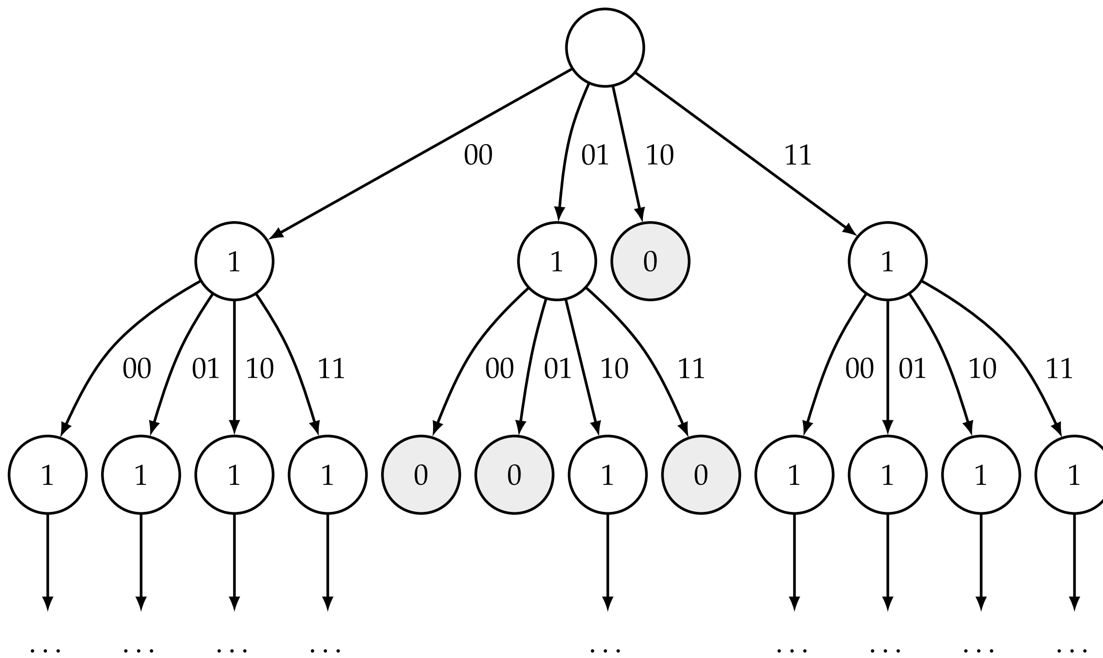


Figure 5.2: First two levels of the k^2-tree for the adjacency matrix in Figure 5.1.

- The matrix is partitioned to allow faster construction and more flexibility in the selection of k.
- Frequent matrices in the bottom layer can be reused of the tree by storing them in a separate data structure and memorizing indices to a specific sub-array; more common patterns get lower indices so that bit usage is reduced.

In [24] a new variation on the concept of k^2-trees is proposed, where a sub-tree can also be *copied* from another region of the adjacency matrix, an approach similar to LZ77 for text compression. This approach, while achieving improved compression ratios, comes at the cost of quadratic-time construction and significantly increased access time, but still supports the same operations of k^2-trees.

We remark that in [16] an approach to compress data structures for point lookups in multidimensional data was proposed that is conceptually similar to [24], but limited to only copying regions that correspond to full nodes. To the best of our knowledge, this has not been yet applied to graph compression, but should provide an intermediate approach between k^2-trees and [24] in terms of compression ratio and construction times.

Finally, in [70] another k^2-tree based approach is proposed, in which the submatrices obtained after a couple of decomposition steps are recombined into a single matrix which is then represented itself with k^2-trees.

5.5 Copying models

Copying models are some of the most successful practical graph compression schemes; they typically offer fast access to single adjacency lists, but not to reverse neighbours or single edge queries.

The first model based on edge copying is proposed in [1]. In particular, it proposes to choose a *reference* node for some nodes of the graph, and to represent the corresponding adjacency lists with a bitmask that defines which edges should be copied from the reference, followed by additional edges.

WebGraph [22, 23] improves the copy-from-reference model by introducing gap coding of non-copied edges, run-length encoding of copy patterns, using ζ codes (described in Subsection 2.3.5), introducing interval coding and introducing a length limit on the maximum reference chain to allow random access during decoding. The WebGraph scheme is described in more detail in Subsection 5.5.1.

A similar scheme to WebGraph is the one used in Graph Compression by BFS [5] (GCBFS). However, instead of copying edges from another adjacency list, the proposed scheme encodes a *repetition count* for some regions of the gap-coded adjacency lists - similar to two-dimensional run-length encoding. GCBFS also proposes a reordering scheme for nodes, discussed in Section 5.7.

Finally, the schemes proposed in [4, 52, 53] split the graph in chunks of consecutive adjacency lists, encoding each chunk independently using, respectively, Re-Pair style compression, list copying combined with Deflate encoding, and merging the lists and encoding the combined list plus a bit-mask encoding which of the lists each edge belongs to.

5.5.1 Brief summary of WebGraph

Let W and L be global parameters representing the "window size", which is limited to speed up compression time, and the "minimum interval length". For each node $u \in V$, WebGraph encodes its *degree* $\deg(u)$ and, if $\deg(u) > 0$, the following information for the adjacency list of u:

1. A *reference number* r, which can be either a number in $[1, W)$, meaning that the list is represented by referencing the adjacency list of node $u - r$ (called *reference list*), or 0, meaning that the list is represented without referencing any other list.

2. If $r > 0$, it is followed by a list of integers indicating the indices where the reference list should be split to obtain contiguous *blocks*. Blocks in even positions represent edges that should be copied to the current list. The format contains, in this order, the number of blocks, the length of the first block, and the length minus 1 of all the following blocks (since no block except the first may be empty). The last block is never stored, as its length can be deduced from the length of the reference list.

3. A list of *intervals* follows; each interval has at least L consecutive nodes that are not copied from the blocks in point 2.

4. Whatever nodes are left from points 2–3 are called *residuals*, and they are *gap-coded*. Their number can be deduced by the degree, the number of copied edges and the number of edges represented by intervals. The first residual is encoded by difference with respect to u (and thus it can be a negative number), and each of the remaining residuals is represented by difference with respect to the previous residual, minus 1.

WebGraph represents the resulting sequence of non-negative integers by using ζ codes [23], a set of universal codes particularly suited to represent integers following a power-law distribution.

Moreover, to guarantee fast access to individual adjacency lists, WebGraph limits the length of the *reference chain* of each node. In particular, a reference chain is a sequence of nodes $u_1, \ldots, u_\ell$ such that node n_{i+1} uses node n_i as a reference r. Every chain has length $\ell \leq R$, where R is a global parameter.

Indeed, to decode a node i that uses r as a reference, we need to first decode r. However, r itself may use a reference. Having a limit on the length of a reference chain ensure that no more than R nodes will ever be required to decode a given adjacency list, giving an upper bound on the decoding time that is proportional to R times the length of an adjacency list.

5.6 Decomposition

Decomposition-based approaches share one recurring theme: splitting the input graph into one or more parts, which get encoded separately (possibly with entirely different

schemes). One of the first instances of this approach appears in [97], which uses different Huffman codes based on whether the destination node belongs to the same web domain or not.

In [87], the graph to be encoded is split into groups, and then for each group edges are encoded separately; for edges between groups, either the edges are represented directly or their complement is, depending on which is smaller.

Similarly, [63] groups together nodes with similar adjacency lists, and then collapses links between different groups, storing the information needed to reconstruct those links in a separate structure.

In [8], links within local groups of nodes are encoded with a special representation that takes into account commonly occurring patterns, such as fully connected subgraphs.

In [35], the Re-Pair approach is combined with k^2-trees: the input graph is divided by domain, then k^2-trees are used to encode edges inside a given domain, and Re-Pair is used for all other edges.

Another approach is to use techniques for *graph summarization*. This is done for example in [96], where the graph is grouped in clusters of nodes with similar adjacency lists; the graph is then encoded as a graph of connections between clusters, together with two graphs of *corrections*, which are edges that should be either added to or removed from the edges defined by the clusters. The graph on the clusters and the correction graphs are encoded using any graph compression scheme, such as WebGraph or GCBFS. As graph summarization techniques are typically lossy, and otherwise decompose the graph into parts to be compressed with other techniques, a more in-depth discussion is out of scope for this thesis; the reader interested in more details can refer to [71].

Finally, another class of approaches [56, 57], building upon and generalizing the ideas of [28], finds *dense subgraphs* in the input graph, defined as pairs (S, C) such that all possible edges between S and C are present. We remark that when $S = C$ this definition corresponds to a clique, and when $C \cap S = \varnothing$ it corresponds to a complete bipartite subgraph. All the remaining edges are then compressed with k^2-trees or with WebGraph.

5.7 Reordering

Permuting a graph to increase compression efficiency is a well-studied problem, with many different variations that are known to be NP-complete; for example, [40] shows that finding the permutation of the graph that minimizes the sum of logarithms of the gaps between adjacent nodes is a NP-complete problem.

Other variations of this problem are also NP-complete. For example, if we denote by $\pi(v)$ the label assigned to vertex v, the problems of finding the permutation π that minimizes either

$$\sum_{(u,v)\in E} |\pi(v) - \pi(u)|$$

or

$$\sum_{(u,v)\in E} \log(|\pi(v) - \pi(u)| + 1)$$

are NP-complete, as seen in [34].

As it is not feasible to find an optimal solution to the graph reordering problem, multiple heuristics have been proposed to produce good orders.

[5] simply orders the graph in BFS order.

In [20, 21], the Layered Label Propagation ordering algorithm is proposed, which produces an ordering of the nodes by assigning labels to each node of the graph and repeatedly updating each label according to local rules.

In [34], an ordering based on *shingles* is proposed: for each node, a low-dimensional locality-sensitive hash of its adjacency list is computed; nodes are then sorted in lexicographical order of their shingles. The effect of the proposed order is to cluster together nodes with similar adjacency lists, as they are likely to share a long prefix of their hash.

The Recursive Bisectioning approach in [40] proceeds by splitting the graph into two parts and moving nodes between the components based on an estimate of the compression cost of the proposed partition. The algorithm then proceeds recursively in the two halves of the partition.

In [95], a DFS order is proposed to improve k^2-tree compression efficiency, with the peculiarity that the next node to be visited is chosen as the neighbour that maximizes the Jaccard similarity between the adjacency lists of the current node and the candidate node.

In [13], ordering nodes according to decreasing in-degree is proposed, since the compression scheme they use requires a number of bits proportional to the logarithm of the label of the destination node. As nodes of high in-degree would receive lower labels, this reduces the total bit expenditure.

CHAPTER 6

GRAPH COMPRESSION IN THEORY

MULTIPLE GRAPH MODELS have been defined in the literature, each of them suited to represent the characteristics of specific classes of graphs. This chapter is dedicated to the study of the entropy of the graphs sampled from these distributions.

We will also provide, for multiple models, an encoding scheme that is optimal, i.e. uses a number of bits equal to the entropy (except for lower-order terms), or almost-optimal.

Many of the models described here rely on a process of *incremental construction*. It is often easy to achieve optimal compression ratio when the construction order is known; however, achieving optimal compression when the order is unknown can be an harder problem. Thus, we will also study the problem of compressing the structure of graphs sampled from these modes, with unknown labels, and provide algorithms or hardness proofs.

6.1 Erdös-Rényi

The Erdös-Rényi random graph model [43] is the oldest and best known model for random graphs. There are two variations of this model, which present similar properties:

- $G(n, m)$ (the Erdös-Rényi variant) is a graph on n vertices with m edges (of course it is necessary that $m \leq \binom{n}{2}$); all the graphs with this number of edges are equiprobable.
- $G(n, p)$ (the Gilbert variant [50]) is a graph on n vertices; every edge of the graph is present with probability p, independently of other edges.

It's easy to see that the entropy of a graph drawn from the $G(n, p)$ random graph model variant is $\binom{n}{2}H^p$. By the formulas in Section 2.1, we can compute the entropy of the $G(n, m)$ model as

$$\log \binom{\binom{n}{2}}{m} = \binom{n}{2} H^{\frac{m}{\binom{n}{2}}} + O(\log n)$$

Thus, by choosing $p = \frac{m}{\binom{n}{2}}$, the two models have the same entropy up to lower-order terms.

A simple compression scheme, that just uses a Binary Arithmetic Coder (see Subsection 2.4.2) to encode whether each edge is present or not, using as a probability the fraction of edges that is present, will utilize a number of bits equal to $\binom{n}{2}H^{\frac{m}{\binom{n}{2}}}$, which is in expectation within $O(\log n)$ of the entropy of a graph sampled from this class.

6.2 Stochastic Block Model

The Stochastic Block Model can be considered a generalization of the $G(n, p)$ random graph model; it is extremely common for modeling graphs for clustering problems.

It is defined by a partition $C_1, \dots, C_k$ of $\{1, \dots n\}$, called the *communities* of the resulting graph, and a symmetric probability matrix P of size $k \times k$; The resulting graph has n nodes, and every edge (u, v) is present independently with probability P_{ij}, where $u \in C_i$ and $v \in C_j$. If $k = 1$, this corresponds to a $G(n, P_{1,1})$ random graph.

If the communities are known, it is easy to see that a compression scheme similar to the one proposed above for Erdös-Rényi random graphs achieves entropy bounds, up to the cost of storing the communities (which is linear in n, hence not the dominant term as n grows, if k is fixed).

Retrieving the community structure of a graph belonging to this family is a well-studied problem in the literature, and a few polynomial-time algorithms are know that compute the community structure successfully w.h.p. under mild assumptions.

For example, the algorithm proposed in [46] succeeds with high probability in the case in which $P_{ii} = p$, $P_{ij} = q$ if $i \neq j$, p is sufficiently large, and q is sufficiently smaller than p.

Finally, we remark that community identification is *not* necessary to compress a graph from the Stochastic Block Model in optimal space using polynomial time: a recent work [15] shows that a suitably-chosen adaptive probability model, used in conjunction with arithmetic coding, achieves optimal compression.

6.3 Uniform attachment

The *uniform attachment* model [10] is one of the simplest models that can be thought of as being constructed incrementally. The construction is based on two parameters, n and d. The process starts with a graph with d nodes and no edges, and proceeds by adding one node at a time until n nodes are present. Whenever a node is added, we select d existing nodes uniformly at random and add links from the new node to those nodes.

The resulting graph is in a bijection with the set of sets of neighbours that were chosen at each step. As each choice of sets is independent, we can write the entropy of a graph from this random family as

$$
\begin{aligned}
H(UA(n,d)) &= \sum_{i=d}^{n} \log \binom{i}{d} = \\
&= \sum_{i=d}^{n} \log i! - \log d! - \log(i-d)! = \\
&= -(n-d)\log d! + \sum_{i=d}^{n} \sum_{j=0}^{d-1} \log(i-j) = \\
&= -(n-d)\log d! + \sum_{j=0}^{d-1} \Big(\log(n-j)! - \log(d-j-1)! \Big) = \\
&= d\log n! - (n-d)\log d! \\
&\qquad - \sum_{j=0}^{d-1} \left(-\log d! + \log(d-j-1)! + \sum_{k=0}^{j-1} \log(n-k) \right) = \\
&= d\log n! - (n-d)\log d! \\
&\qquad - \sum_{j=0}^{d-1} \left(-\log(d-j) - \sum_{k=0}^{j-1} \log(d-k) + \sum_{k=0}^{j-1} \log(n-k) \right) =
\end{aligned}
$$

$$= d \log n! - (n+1) \log d! - \sum_{j=0}^{d-1} \sum_{k=0}^{j-1} \log \frac{n-k}{d-k} =$$
$$= d \log n! - (n+1) \log d! - \sum_{j=0}^{d-1} (d-1-j) \log \frac{n-j}{d-j} =$$
$$= d \log n! - n \log d! + O(\log n)$$

If the construction order is known, it is trivial to reconstruct the sets that were selected at each step, and thus to achieve a compression ratio matching the entropy of this model of graphs.

We now consider the problem of compressing the structure of the graph, ignoring the labels (and thus without knowing the construction order). We will call an unlabeled $UA(n,d)$ graph a $\tilde{UA}(n,d)$ graph. As it is possible to recover a $UA(n,d)$ graph by storing only a permutation of n elements and a $\tilde{UA}(n,d)$ graph, it follows that the entropy of this family of graphs is at least

$$H(\tilde{UA}(n,d)) \geq H(UA(n,d)) - \log n! = (d-1) \log n! - n \log d! + O(\log n)$$

Note that a $UA(n,d)$ graph has $d(n-d)$ edges. Thus, assuming d constant and n sufficiently large, it is not possible to compress a $UA(n,d)$ random graph using a constant number of bits per edge, as the entropy per edge is $\Omega(\log n)$.

We now provide a compression scheme for $\tilde{UA}(n,d)$ graphs that can be computed in polynomial time and prove that it achieves optimality.

Lemma 14. *A $\tilde{UA}(n,d)$ graph has, for a large enough n,* arboricity *d.*

Proof. We recall that the *arboricity* of a graph G is the minimum number k such that there exist k disjoint forests whose union contains all the edges of G.

We can obtain a decomposition in d forests of a $\tilde{UA}(n,d)$ graph as follows.

For a node i not in the initial set D of d nodes, let $n_{i1}, \ldots, n_{id}$ be the nodes (with $n_{ij} < i$) that were chosen as targets of edges during the construction of the graph.

We decompose the edges of our graph in the sets

$$F_j = \{(i, n_{ij}) : i \in N(G) \setminus D\},\ j = 1 \ldots d$$

All the F_j are clearly acyclic, as every node has a single edge towards a node with lower label. Thus, $\{F_j : j = 1 \ldots d\}$ is a valid decomposition in d forests, and the arboricity of a $\tilde{UA}(n,d)$ graph is at most d.

Given that a forest on n nodes has at most $n-1$ edges, and given that a $\tilde{UA}(n,d)$ graph has $d(n-d)$ edges, it follows that its arboricity is at least $\frac{d(n-d)}{n-1}$, which is strictly

greater than $d-1$ if $n > d^2$. Thus, for sufficiently large n, the arboricity of a $\tilde{UA}(n,d)$ graph is exactly d. □

Lemma 15. *An unlabeled graph with arboricity d, with a known decomposition in forests, can be compressed in* $\lceil (d-1)(n-1)\log(n+1) \rceil + 2n$ *bits.*

Proof. It is sufficient to store the set of forests that form our graph. We do so by adding a single node $n+1$ to every forest, and connecting it to a single node for each connected component of the forest, thus obtaining a set of d trees.

As we are only interested in storing the edges of the graph, and not the indices of each node, we can store one of the trees using the Balanced Parenthesis scheme described in Subsection 5.3.1, using in total $2n$ bits.

For the remaining $d-1$ trees, it is necessary to also store the indices of their nodes with respect to the first tree, to be able to properly reconstruct the graph.

We can represent each tree using its Prüfer sequence: the end result is a sequence of $(d-1)(n-1)$ numbers in the $[1, n+1]$ range, that can thus be stored by an Arithmetic Coder (using uniform probabilities) in $\lceil (d-1)(n-1)\log(n+1) \rceil$ bits. □

We recall the following theorem:

Theorem 16 ([49])**.** *The arboricity of a graph and the corresponding decomposition in forests can be computed in polynomial time in the size of the graph.*

We remark that in graphs belonging to the Uniform Attachment model, the *degeneracy*, or *coloring number* [44] of the graph is equal to its arboricity if n is large enough. We recall that the degeneracy is described as the smallest value k such that every subgraph contains a node of degree at most k. Equivalently, a graph has degeneracy k if it can be permuted in such a way that the number of edges towards nodes of higher index is at most k. It is clear from the construction process of $UA(n,d)$ graphs that they have degeneracy d. Moreover, we can construct d forests that cover the graph by selecting one forward edge per node in each forest. Since the permutation can be computed in $O(m)$ time [76], it follows that the optimal decomposition in forests for this class of graphs can be found in *linear* time.

Thus, we obtain the following theorem:

Theorem 17. *An unlabeled graph G generated with the Uniform Attachment process can be compressed optimally (up to lower order terms) in polynomial (linear) time.*

Proof. We begin by applying Theorem 16 to obtain in polynomial time an optimal decomposition of G in d forests. Equivalently, we compute the forward-degree-minimizing permutation of G to compute an optimal decomposition in $O(nd)$ time.

By Lemma 15, such a graph can be compressed in $(d-1)(n-1)\log(n+1) + O(n) = (d-1)n\log n + O(n)$ bits. By Lemma 14, d is also the second parameter that defines the distribution that G is drawn from, thus its entropy is $(d-1)\log n! + O(n) = (d-1)n\log n + O(n)$. Thus, the proposed scheme is optimal up to lower order terms. □

6.4 Copy model

We now consider another model that proceeds by incrementally constructing a network. This model is parameterized by n, the total number of nodes, d, the out-degree of every node, and α, a *copy probability*.

The model is constructed starting with a complete graph on $d+1$ nodes. Then, as is done for the Uniform Attachment model, nodes are added one by one, each of them with d edges, going from the new node to an existing one. However, the edges are picked differently:

- First, a *reference node* r is chosen among the existing nodes, uniformly at random.
- For each of the d edges, with probability α the corresponding edge is copied from the corresponding outgoing edge of r
- Otherwise, a destination node is chosen uniformly at random among the existing nodes.

We denote a graph from this model as $C(n, d, \alpha)$. We remark that, when $\alpha = 0$, this model corresponds exactly to the Uniform Attachment model described above.

In general, during the generation of a graph according to this model, $(1-\alpha)d$ edges per node (on average) are chosen using the same procedure used for $UA(n, (1-\alpha)d)$. Thus, we can bound from below (up to lower order terms) the entropy of a copy model graph as:

$$H(C(n, d, \alpha)) \geq d(1-\alpha)\log n! + O(n)$$

We remark that when α is very close to 1, the model becomes somewhat uninteresting: indeed, in the extreme case of $\alpha = 1$, each node has the same list of out-edges as one of the initial $d+1$ nodes. Thus, we will focus on the case where α is not very close to 1.

We call $\tilde{C}(n,d,\alpha)$ a graph taken from the copy model without node labels. As before, we trivially have

$$H(\tilde{C}(n,d,\alpha) \geq (d(1-\alpha)-1)\log n! + O(n)$$

We now present a compression scheme that achieves close to optimality on this class of graphs (up to lower-order terms). A variant of this compression scheme can be found in [1].

The compression scheme uses an auxiliary forest F to encode the reference used by each node, where a parent-child relationship between p and c means that node p is chosen as a reference for node c. A node that is not a child of any parent is encoded with no reference.

The encoding proceeds as follows:

- The forest is stored using, like for the first forest in Uniform Attachment, $2n$ bits.
- For each node, for each of its d out-edges, a bit is stored to represent whether the corresponding edge was copied from the reference or not. In practice, the number of bits can be reduced by using e.g. a Binary Arithmetic Coder.
- For every edge that is not copied, $\lceil \log n \rceil$ bits are used to store the destination of the non-copied edge. An arithmetic coder could be used to reduce this number to $\approx \log n$, but this is not needed for our purposes here.

We now need to determine a good choice of a forest F. This can be done in polynomial time, giving a total compressed size close to the entropy bound:

Theorem 18. *There exists a polynomial-time algorithm that determines a forest F such that the approach described above uses in expectation at most $d(1-\alpha)n\log n + O(n)$ bits.*

Proof. We construct an auxiliary complete, directed, weighted graph as follows:

- Its nodes are the same nodes as the original graph, plus one.
- There are two edges between each pair of distinct vertices, in both directions.
- The weight of each (u,v) edge is given by the number of edges that can be copied from node u to node v. If either of u and v is the auxiliary node, the weight of (u,v) is 0.

We then find a maximum spanning tree of this graph [64] in $O(n^2 \log n)$ time.

This tree corresponds to a forest (obtained by removing the auxiliary node) that minimizes the number of edges that are not copied. Note that the copy choices made during the construction of the $\tilde{C}(n,d,\alpha)$ graph also correspond to a valid forest on this auxiliary graph.

As the forest corresponding to the original copy choices has a total weight (i.e. number of copied edges) equal to $d\alpha n$ in expectation, the total number of non-copied edges corresponding to the *maximum* weight forest will be in expectation at most $d(1-\alpha)n$, which proves the theorem. □

6.5 Preferential attachment (Barabási-Albert)

We now describe the Barabási-Albert random graph model [10], one of the most well-known models for generating graphs with Zipf degree distributions.

Similarly to the Uniform Attachment, graphs from the $BA(n,d)$ family are also generated incrementally, starting with a complete graph on $d+1$ nodes. At every time step, a node is added to the graph, and d edges are added to the new node. However, differently from the $UA(n,d)$ model, the choice of the new edges is *not uniform*. In particular, if we denote $\delta_i(j)$ the degree of the j-th node of the graph when i nodes are present in total, the probability of connecting node $i+1$ to node j is given by

$$\frac{\delta_i(j)}{\sum_{k=1}^{i}\delta_i(k)} = \frac{\delta_i(j)}{2id}$$

It follows that it is likely for nodes that already have a large number of edges to gain even more edges. This is the "rich get richer" effect, which is often observed in real-world networks.

We will now prove that $H(BA(n,d)) \geq dn\log n + o(n\log n)$; as a $BA(n,d)$ also has arboricity d like a $UA(n,d)$ graph, it follows that the scheme described in Section 6.3 is also optimal for the unlabeled version of $BA(n,d)$ graphs, and thus Theorem 17 has an equivalent version:

Theorem 19. *An unlabeled graph G generated with the Preferential Attachment process can be compressed optimally (up to lower order terms) in polynomial (linear) time.*

We remark that [72] proves a lower bound on the entropy of a $\tilde{BA}(n,d)$ graph of $n(d-1)\log n + O(n\log\log n)$ for the case $d \geq 3$. As discussed in Section 6.3, a lower bound of $nd\log n + O(n\log\log n)$ on a $BA(n,d)$ graph is equivalent; hence, the

following provides an alternative proof of the result in [72] as well as an extension of the proof to the $d = 2$ case (the case $d = 1$ is trivial).

To prove this result, we follow a similar procedure to [34]. In particular, we identify a subset $\mathcal{G}(n,d)$ of all the Preferential Attachment graphs with the property that

- $P(G \in \mathcal{G}(n,d) | G \in BA(n,d)) = 1 - O(n^{-3})$, i.e. a $BA(n,d)$ graph is in $\mathcal{G}(n,d)$ with high probability.
- $P(BA(n,d) = G) \leq \mathcal{P}\ \forall G \in \mathcal{G}(n,d)$, i.e. there is an upper bound on the probability of obtaining each specific graph in this subset.

It then follows from the definition of entropy that

$$H(BA(n,d)) = \sum_{G \in BA(n,d)} P(G) \log \frac{1}{P(G)} \geq \sum_{G \in \mathcal{G}(n,d)} P(G) \log \frac{1}{\mathcal{P}} = (1 - O(n^{-3})) \log \frac{1}{\mathcal{P}}$$

We choose our family $\mathcal{G}(n,d)$ as the set of all the $GA(n,d)$ graphs such that

$$\delta_i(j) \leq d\sqrt{\frac{i}{j}} \ln^3 n \qquad \forall 1 \leq j \leq i \leq n$$

It is shown in [39] that this indeed happens with probability $1 - O(n^{-3})$.

In this family of graphs, let S be the set of nodes that are neither connected to any of, nor one of, the first $T = \frac{n}{4d^2 \ln^8 n}$ nodes.

By our choice of family of graphs, we have that

$$\sum_{j=1}^{T} deg(j) = \delta_n(j) \leq d\sqrt{n} \ln^3 n \sum_{j=1}^{T} \frac{1}{\sqrt{j}}$$

Applying the inequality

$$2\sqrt{T} - 2 \leq \sum_{j=1}^{T} \frac{1}{\sqrt{j}} \leq 2\sqrt{T}$$

which can be easily proven by induction [55], we have that

$$\sum_{j=1}^{T} \frac{1}{\sqrt{j}} = 2\sqrt{T} + O(1)$$

Thus,

$$\sum_{j=1}^{T} deg(j) \leq d\sqrt{n} \ln^3 n \left(2\sqrt{T} + O(1)\right) =$$

$$\leq 2d\sqrt{n}\ln^3 n \frac{\sqrt{n}}{2d\ln^4 n} + O(\sqrt{n}\ln^3 n) =$$
$$\leq n\ln^{-1} n + O(\sqrt{n}\ln^3 n)$$

It follows that $|S| \geq (1 - 2\ln^{-1} n)n$ if n is sufficiently large, as the set of nodes that have an edge towards the first T nodes is of size $n\ln^{-1} n + o(n\ln^{-1} n)$, and $T = O(n\ln^{-8} n)$.

Let us now fix $G \in \mathcal{G}(n,p)$, and give an upper bound on $P(BA(n,d) = G)$. To do so, we shall consider an edge (i,j) with $i > j$ and $i \in S$; there are $d(1 - 2\ln^{-1} n)n$ such edges.

As $i \in S$, it then follows that $i \geq j > T$ by definition of S. The probability of this edge to be selected is thus

$$\frac{\delta_i(j)}{2id} \leq \frac{d\sqrt{ij^{-1}}\ln^3 n}{2id} = \frac{\ln^3 n}{2\sqrt{ij}} \leq \frac{\ln^3 n}{2T} = \frac{2d^2\ln^{11} n}{n}$$

By the product rule, it follows that

$$P(BA(n,d) = G) \leq \left(\frac{2d^2\ln^{11} n}{n}\right)^{nd(1-2\ln^{-1} n)} = \mathcal{P}$$

We now just need to compute $\log\frac{1}{\mathcal{P}}$:

$$\log\frac{1}{\mathcal{P}} = (1 - 2\log^{-1} n)(nd\log n - nd(1 + 2\log d + 11\log\ln n)) =$$
$$= nd\log n + O(nd\log\log n)$$

which proves the thesis.

6.6 Simplified Copy Model

Finally, we turn our attention to the model proposed in [34], which has constant entropy per edge while still displaying a Zipf degree distribution.

The model starts from a *seed graph* G_s, with t_0 nodes, each of out-degree d. At each time step, a node x is chosen uniformly at random, and a new node y is added in position $x + 1$ (shifting all other nodes accordingly); an edge from y to x is then added to the graph, and $d - 1$ edges are copied (choosing uniformly at random without replacement - or equivalently, choosing exactly one non-copied edge) from x to y. The process is repeated until n nodes are present in the graph.

There is a simple bijection between the set of graphs generated by this procedure and random recursive forests on n nodes with t_0 roots, and a $[0, d)$ label on each node. Indeed, we can construct such a forest as we build the graph, adding a new leaf to a node whenever the corresponding graph node is chosen as a copy source, and using as its label the index of the edge that is *not* copied. Different graphs will also result in different forests.

From this bijection, it follows immediately that the entropy of this model is given by the log of the number of rooted recursive forests with t_0 roots (which behaves like the number of rooted trees for large n), plus the entropy of the labels, for a total entropy of $\log n! + n \log d$. As was noted in [34], the forest can be reconstructed in linear time from the graph, thus this model (with labels) can be compressed within entropy bounds in polynomial time.

We now consider the case in which labels are removed from the graph. As this corresponds to removing node labels from the corresponding labeled forest, by using a tree representation such as the ones described in Subsection 5.3.1 we can compress these structures in $(2 + \log d)n$ bits, resulting in $(2 + \log d)/d$ bits per edge.

However, this representation is not optimal, as the number of unlabeled rooted trees is asymptotic to (as shown in [80]) $D\alpha^n n^{\frac{3}{2}}$, where $D \approx 0.4399$ and $\alpha \approx 2.955$. Moreover, the process of producing a random rooted forest and removing the labels does not result in an uniform distribution over unlabeled trees; as such, $(\log \alpha + \log d)n + o(n)$ is just an upper bound on the entropy of this model.

As a final remark, we note that there exist *ranking* and *unranking* polynomial-time algorithms for unlabeled rooted trees on n vertices [101, 104]. These algorithms provide a bijection between the set of all unlabeled rooted trees on n node and an appropriate interval $[0, t(n))$ that can be computed (and inverted) in polynomial time. Hence, there is a polynomial-time algorithm to compress this family of graphs down to $(\log \alpha + \log d)n + o(n)$ bits, which matches the obtained upper bound on the entropy of this model of graphs.

CHAPTER 7

GRAPH COMPRESSION IN PRACTICE

THE ZUCKERLI COMPRESSION SCHEME is based on the WebGraph representation [22, 23] together with the novel integer entropy coding techniques presented in Chapter 2. In this chapter, we present the scheme and the results of an experimental evaluation.

This chapter is divided in two parts; the first part is dedicated to presenting the Zuckerli scheme as it was published in [98], which supports fast decompression in either the full decompression or the list decompression cases.

List decompression can allow us to run some graph algorithms directly on the compressed representation on the graph: several fundamental algorithms, such as a graph traversal, are based on partially scanning adjacency lists that are decompressed during the scan.

On the other hand, we do not want to support edge queries for two reasons: it degrades the performance of scanning an adjacency list, and many of the well-known graph algorithms hardly require to access few random items of an adjacency list without accessing the list from the beginning.

Moreover, scanning a list is so fast in our implementation that any attempt to jump parts of it would just degrade the performance due to the extra machinery required.

We also do not consider the problem of querying reverse neighbours, since many algorithms only require scanning either out-neighbours or in-neighbours, but not both.

Section 7.2 describes the Zuckerli high-level encoding scheme, which, in brief, consists in *block-copying*, that is re-using parts of the adjacency lists of previous nodes to encode the adjacency list of current nodes, *delta-coding* of values that are not copied and *context-modeling* of all the values to improve compression. This section also describes *heuristics* to improve the encoding choices made by the encoder. We then report results of the experimental study in Section 7.3.

Finally, the second part of this chapter, specifically Section 7.4, is dedicated to exploring how much the full decompression scheme can be improved by the techniques in Chapter 4, at the cost of sacrificing the high decoding speed of the "baseline" Zuckerli scheme. We also propose a novel algorithm for reference selection, inspired by LZ77 match finding algorithms and the shingle ordering heuristic from [34].

7.1 Encoding Integers

Zuckerli modifies the adjacency lists of the graph, which are sequences of integers, to produce other sequences of integers that can be encoded more succinctly.

Zuckerli uses the hybrid encoding for arbitrary integers described in Chapter 3.

When list decompression is required, Zuckerli combines the Hybrid Integer Encoding with Huffman coding (see Subsection 2.4.1).

When only full decompression is required, Zuckerli uses Asymmetric Numeral Systems (ANS) (see Subsection 2.4.3) instead of Huffman coding.

When list decompression is supported, one disadvantage of ANS (as well of as other encoding schemes that can use a non-integer number of bits per encoded symbol) is that it requires keeping track of its internal state. For decoding to successfully be able to resume from a given position in the stream, it is also necessary to be able to recover the state of the entropy coder at that point of the stream, which would cause significant per-node overhead if using ANS. Thus, in this case, Zuckerli switches to using Huffman coding.

7.1.1 Negative integers

We encode an integer s that is not known to be positive using the following, easy to reverse bijection between integers and natural numbers, introduced in [22]:

$$x \to \begin{cases} 2 \cdot x & \text{if } x \geq 0 \\ -2 \cdot x - 1 & \text{if } x < 0 \end{cases} \quad (7.1)$$

7.2 Graph compression in Zuckerli

In this section, we summarize the novel aspects introduced by Zuckerli in relation to WebGraph.

Firstly, Zuckerli entropy-encodes the integers, as described in Section 7.1. This is in contrast with WebGraph's ζ coding [23].

Secondly, Zuckerli splits the nodes of G into *chunks* of size C, where the first chunk contains the first C nodes in G, the second chunk contains the following C nodes in G, and so on. When list decompression is not required, we set $C = \infty$. The encoding of the *degrees* of the nodes operates independently in each chunk. By doing so, it is sufficient to decode the degrees of the chunk of a given list to be able to decode its degree; this is a strict requirement to support list decompression.

It can be observed experimentally (see Section 7.3) that the representations of node degrees requires a significant amount of bits. To improve compression, Zuckerli represents degrees via delta encoding, i.e. as the difference between the current degree and the previous one. As this procedure may produce negative numbers, deltas are represented using the transformation described in Equation (7.1).

Thirdly, while Zuckerli uses *reference* lists and blocks in the same way as WebGraph (points 1 and 2 in Subsection 5.5.1), the choice of the reference list and reference chain is more sophisticated. We defer its description to Subsection 7.2.2.

Fourthly, Zuckerli does not use *intervals*, in contrast with WebGraph (point 3 in Subsection 5.5.1). As a form of simplification, the special representation for intervals is replaced with run-length encoding [91] of zero gaps. When reading residuals, as soon as a sequence of exactly L' zero gaps is read, for a global parameter L', another integer is read to represent the subsequent number of zero gaps, which are not otherwise represented in the compressed representation. Since ANS does not require an integer number of bits per symbol, and allows for very efficient representations of sequences of zeros, we set $L' = \infty$ if list decompression is not supported.

Finally, Zuckerli modifies the representation of the *residuals*, which are stored via delta encoding. The representation chosen by WebGraph (point 4) does not exploit the fact that an edge might already be represented by block copies (or intervals). For example, consider the case in which an adjacency list contains edges $\{1, 2, 3, 4, 8, 9\}$, and edges $\{1, 2, 4\}$ are already represented by block copies. Residuals would then be $\{3, 8, 9\}$ and the second residual would be represented by WebGraph using a delta of $4 = 8 - 3 - 1$. However, this representation does not take into account the fact that not all possible delta values smaller or equal to 4 are useful. In this example, reading a

reference node (6)	1	2	4	5	7	10	11	12		
current node (7)	1	2	3	4	8	9	10	11	12	13
block lengths	3	2	3							
block encoding	3	1								
residuals	3	8	9	13						
residuals delta	−4	3	0	0						
list repr.	2	1	2	3 1	−4 3 0 0					

Figure 7.1: Example encoding of an adjacency list. We are encoding the adjacency list of node 7 using the adjacency list of node 6 as a reference. Highlighted in blue are the edges that the two nodes have in common, i.e. the blocks to be copied from the reference node adjacency list. The block encoding is performed as described in Subsection 5.5.1 (point 2). Highlighted in red are the residual values, which are stored as follows: the first residual is encoded as the delta between the current node and the actual residual, while the next values are encoded as $d - 1$, where d is the value to add to the previous residual, implicitly skipping any possible edges that have already been added though blocks. The boxes in the final list representation show, in order, the data that gets encoded: the delta of the degree of the current node with respect to the previous node, the delta (in absolute value) of the reference node with respect to the current node, the number of blocks, the block encoding, the residual deltas.

delta of 0 from the compressed file would result in an edge value of 4, which would be either invalid or superfluous, as this edge is already represented through blocks. Thus, Zuckerli modifies the delta encoding of residuals by subtracting the number of edges that are between the previous and the current residual edge and that are already known to be encoded using blocks. In this case, residual edge 8 would be represented as 3 instead of 4, as there are only 3 possible edges between 3 and 8 that are not already represented in the block copies.

A full example description of how Zuckerli would represent an adjacency list is shown in Figure 7.1.

7.2.1 Context management

As mentioned in Section 7.1, Zuckerli uses Huffman coding and ANS with multiple contexts, i.e. distinct probability distributions. To the best of our knowledge, while this is a well-known encoding technique, its application to graph compression is new. Here we detail how symbols are split among the different contexts. We remark that the

scheme described here produces a small number of contexts (below a thousand), hence the context management techniques of Chapter 4 are not used.

Inside each chunk, the symbol that represents the delta-coded degree with respect to the previous node is used to choose the distribution for the current node. Similarly, inside a chunk, the reference number used for the last list is used to choose a distribution for the current one.

When compressing blocks, a separate distribution is used for the first block, all the even blocks, and all the odd blocks. This is because the first block is the only one that does not have its length reduced by 1, and we expect the number of edges to be copied (odd blocks) to have a different distribution from the number of edges to be skipped (even blocks), depending on the graph.

For delta-encoding the first residual with respect to the current node, the symbol that would be used to represent the number of residuals defines which distribution to use. This is because a list with a high number of residuals will likely be harder to predict.

Finally, for all other residual deltas, the symbol that was used to encode the previous one is used to choose the corresponding probability distribution for the current delta.

We remark that each probability distribution used by Zuckerli is stored in the compressed file, and is not changed as edges are decoded.

7.2.2 Choice of reference list and chain

We explain how Zuckerli selects reference lists to be used during compression. As previously discussed, we may either represent a node's list explicitly or, if we use a reference, we represent the difference from the list of its reference.

To make an effective choice, we need to estimate the amount of bits that the algorithm will use to compress an adjacency list using a given reference. Since we use entropy coding, this is not a simple task, as choices for one list might affect probabilities for all other ones.

We choose to use an iterative approach previously used by Zopfli [3], a general-purpose compression algorithm. We initialize symbol probabilities with a simple fixed model (all symbols have equal probability), and then choose reference lists assuming these will be the final costs. We then update the symbol probabilities given by the chosen reference lists and repeat the procedure with the new probability distribution. This process is then repeated a constant number of times.

We now consider the two types of compression separately.

7.2.3 Full decompression

In this case, there is no limitation on the length of the reference chain used by a single node, i.e., a reference node may itself have a reference node, and so on; we obtain an optimal solution with the greedy strategy, choosing the reference node that gives the best compression out of all the ones available in the window of the current node, i.e., the W preceding nodes.

7.2.4 List decompression

To decompress a single list, we must also decompress its reference chain: when access to single lists is requested, more care is required to select good references while avoiding reference chains longer than a given threshold R.[1]

For example, imagine these are the lists of nodes 1,2 and 3:

$$1 : \{3,4,7\},\ 2 : \{3,4,7,9\},\ 3 : \{4,7,9\}$$

We may want to represent 2's list using 1's as a reference: this way we do not need to represent 3, 4, and 7, but just the node 9 in the difference; similarly, if we represent 3's list using 2's as reference, we just need to omit node 3. However, in order to decompress 3's list we will need to read (hence decompress) the list of its reference 2, which in turn requires decompressing 1's list. The longer the chain, the longer the decompression time: the parameter R allows us to keep this overhead under control.

We can formally state the problem of choosing the references as follows. We are given a directed acyclic graph D, where the nodes represent the adjacency lists. There is an arc between two nodes if one adjacency list can refer to the other. The weight of the arc corresponds to the number of bits saved by choosing that reference. The larger the weights, the better the compression gain. Thus, we aim at finding a maximum-weight directed forest O for D, where each node has out-degree at most one (its reference), and there are no directed paths longer than R (i.e. a reference chain longer than R). Finding an optimal solution seems not trivial, and it is unclear whether it can be done in polynomial-time. [2]

Zuckerli uses an efficient heuristic with approximation guarantees. Given D, it first builds the optimal directed forest F, ignoring the constraint that directed paths cannot be longer than R (this corresponds to the solution of the full decompression case).

---

[1] Each node u may refer in turn to any of its W preceding ones during a hop, which makes R unrelated to W: indeed, R is the maximum number of these hops.

[2] We speculate it may be NP-complete due to similarities with maximum directed cuts [83].

Instead of solving our problem on D as we formulated above, Zuckerli computes an optimal sub-forest H on F, as the latter be found by the following dynamic programming algorithm, answering the question *"what is the sub-forest H of maximum weight that is contained in the current subforest of F and does not have paths of length $R+1$?"*.

Clearly, H is not necessarily the optimal solution for D, as it is computed for its subgraph F. However, there may still be arcs of D that were not in F, but can now be added to H without creating long chains. Zuckerli tries to extend H with such arcs in a greedy way, obtaining the final heuristic solution.

7.2.5 Approximation guarantee

Interestingly, our heuristic algorithm not only works quite well in practice, but it also provides a guaranteed $(1-\frac{1}{R+1})$-approximation of the optimal solution on D, i.e. of the maximum number of bits to be saved.

To see why, let O be the optimal solution, and let w_O, w_F and w_H be the total weights of O, F, and H, respectively.

Next, let H' be a sub-forest of F obtained by splitting the arcs of F in $R+1$ groups, depending on their distance from the root of their tree in F modulo $R+1$, then removing the group of smallest weight; it is evident that H' has no paths longer than R, and that its weight $w_{H'}$ is at least $(1-\frac{1}{R+1})w_F$, as the weight of smallest of the $R+1$ groups could not be more than $\frac{1}{R+1}w_F$.

Now observe the following:

- $w_F \geq w_O$, as F is the optimal solution for $R=\infty$, a problem with less constraints.
- $w_H \geq w_{H'} \geq (1-\frac{1}{R+1})w_F$, as H' is a sub-forest of F, and H is the best sub-forest in a search space that contains the optimal sub-forest of F (both with path length bounded by R).
- Thus, $w_H \geq (1-\frac{1}{R+1})w_F \geq (1-\frac{1}{R+1})w_O$, which proves the approximation bound.

7.2.6 Details on computing the optimal sub-forest of *F*

Given a sub-forest F' of F rooted in the node x, let $M_i(x)$ be the maximum weight sub-forest of F' that has no paths longer than R, and in which the root x is in no path longer than i. If r_j are the roots of F, $\bigcup_j M_R(r_j)$ is the optimal sub-forest of F we are looking for. We implement a dynamic programming procedure based on the following

invariant: if, for all sub-forests rooted in each child y of x, we know $M_i(y)$ for each $i \in \{0, \dots, R\}$, we can compute $M_i(x)$ for each $i \in \{0, \dots, R\}$.

First, as paths are always directed from nodes to their parent, observe that we can consider each child y of x independently. Furthermore, $M_i(x)$ is computed as follows: if the arc (x, y) is taken, then y in its sub-forest may only be part of paths of length at most $i - 1$; on the other hand, if we do not choose (x, y), y may be included in paths of any length up to R. Finally, for the base case, observe that for any leaf l of F, $M_i(l) = \emptyset$. We thus obtain each $M_i(x)$ by the following formula:

$$M_i(x) = \bigcup_{y \in \texttt{children}(x)} \texttt{max_weight}\left(M_R(y), \{(x, y)\} \cup M_{i-1}(y)\right)$$

where $\texttt{children}(x)$ are the children of x in F, and $\texttt{max_weight}(A, B)$ returns the set of arcs having greater weight between A and B (breaking ties arbitrarily).

Finally, we give a brief remark on the complexity. This is important since a trivial implementation would take quadratic time and space to represent each set $M_i()$, making this approach unfeasible on graphs with millions of nodes. However, we can implement it in $O(nR)$ time and space, where n is the number of nodes in F, as follows. We can first run the above dynamic programming algorithm, but associate with each $M_i(y)$ just its weight. Furthermore, we keep track for each $M_i(x)$ of which was the choice performed on each child y of x (i.e., whether we used (x, y) or not). Computing the weights of $M_i(x)$ this way takes just $O(1)$ time for each child, costing us in total $O(nR)$ time as F has $O(n)$ arcs. With this information, we can reconstruct exactly which arcs are used in the optimal solution $M_R(r)$ in a top-down manner by looking at the information about its children we previously computed.

7.3 Experimental results

In order to evaluate the efficiency of Zuckerli, we first study the effects of various choices of parameters on compressed size. We also evaluate the effectiveness of the approximation algorithm for reference selection.

We then compare the compression ratio of Zuckerli with respect to existing state-of-the-art compression systems for large graphs, either with novel experiments (WebGraph [22], Graph Compression by BFS [5], k^2-trees [25], LM [53], Backlinks [34]) or by referring to the experiments in the relevant papers (LogGraph [13] and 2D-Block Trees [24]). We remark that the proposed scheme does not change the order of nodes before compression, and as such a comparison with works that propose algorithms

name	nodes	edges
cnr-2000	325 557	3 216 152
in-2004	1 382 908	16 917 053
eu-2005	862 664	19 235 140
hw-2009	1 139 905	113 891 327
bn-jung	784 262	267 844 668
uk-2002	18 520 486	298 113 762
tw-2010	41 652 230	1 468 365 182
pp-miner	8 254 694	1 847 117 370
sk-2005	50 636 154	1 949 412 601
uk-2007-02	110 123 614	3 944 932 566
eu-2015	1 070 557 254	91 792 261 600

Table 7.1: Graphs used during experiments, with node and edge counts. All graphs are web graphs, except `hw-2009` (`hollywood-2009`) and `tw-2010` (`twitter-2010`), which are social networks, `bn-jung`, which is a brain network, and `pp-miner`, which is a protein interaction network.

to find a better node permutation (such as [40]) is out of scope of this experimental comparison, although it is an interesting direction for future work.

To evaluate the CPU and memory usage of Zuckerli, we compare its decompression time and memory usage with the corresponding metrics for WebGraph. Moreover, we compare the running time of a depth-first search and a breadth-first search on Zuckerli-compressed graphs, on WebGraph-compressed graphs and on uncompressed graphs.

Finally, to evaluate the parallelism of access, we compute the speedup achieved by Zuckerli on an edge-summing problem when running on 2, 4, 8, 16, 32 and 64 cores.

For all experiments where list decompression is required, R is set to 3 (similarly to the compressed WebGraph files that used for comparisons), the chunk size C is set to 32, and the minimum run of 0s to use RLE L' is set to 3.

The code to run the experiments was written in C++ and compiled with `clang++`, version 10; it is available at `https://github.com/google/zuckerli`.

The experiments were ran on a 32-core AMD 3970X CPU (with SMT enabled) with 256GB of RAM.

Timing information was obtained by using the `chrono` utilities in the C++ standard library, or equivalent utilities in Java. Memory usage was measured by the `time` command available on Linux systems.

	Size (bits per edge)					
k	3	4	4	4	5	5
i	1	1	2	2	2	2
j	0	0	0	1	0	1
cnr-2000-hc	**1.86**	1.87	1.88	1.89	1.90	1.91
cnr-2000	**2.23**	2.24	2.25	2.29	2.26	2.31
in-2004-hc	**1.32**	1.33	1.33	1.33	1.33	1.34
in-2004	**1.69**	**1.69**	1.71	1.77	1.72	1.79
eu-2005-hc	2.49	2.49	**2.47**	**2.47**	**2.47**	**2.47**
eu-2005	**2.88**	2.89	**2.88**	2.92	**2.88**	2.93
uk-2002-hc	1.38	1.38	**1.37**	**1.37**	1.38	**1.37**
uk-2002	**1.75**	1.76	1.78	1.87	1.79	1.89
tw-2010-hc	**11.99**	12.00	**11.99**	**11.99**	12.00	12.00
tw-2010	**12.12**	12.13	**12.12**	12.58	12.26	12.62
uk-2007-02-hc	0.92	0.92	0.92	**0.91**	0.92	0.93
uk-2007-02	**1.20**	**1.20**	1.22	1.30	1.23	1.31

Table 7.2: Effects of changing hybrid integer encoding parameters.

7.3.1 Datasets

To run the comparisons, we use graphs from the WebGraph corpus [22, 20, 19], which are available at http://law.di.unimi.it/datasets.php. The datasets we use include both social networks and web graphs, with a number of edges varying from a few millions to 91 billions and a number of nodes varying from a few hundred thousands to 1 billion. More details about the graphs can be found in Table 7.1. When reporting results, graphs with a -hc suffix represent the full decompression versions, while other graphs represent the compressed versions also supporting list decompression.

We also compare Zuckerli with other compression schemes on networks outside the core focus of Zuckerli itself, namely, a protein interaction network (pp-miner, from [105]) and a brain network (bn-jung, from [92]).

7.3.2 Parameter Choice

We first investigate the effect of the parameters controlling the integer encoding scheme, trying different combinations of the number of bits that are included in the entropy-coded part and the number of integers that are entropy coded as-is. The results are

shown in Table 7.2. They show that using more fine-grained integer representations, i.e. entropy-coding more bits or having more direct-coded integers, does not give significant improvements or even worsens the compression ratio.

name	Size (bits per edge)		
	$W = 16$	$W = 32$	$W = 64$
cnr-2000-hc	1.95	1.87	**1.82**
cnr-2000	2.31	2.24	**2.20**
in-2004-hc	1.34	1.33	**1.31**
in-2004	1.71	1.69	**1.68**
eu-2005-hc	2.60	2.49	**2.43**
eu-2005	2.99	2.89	**2.83**
uk-2002-hc	1.42	1.38	**1.35**
uk-2002	1.79	1.76	**1.73**
tw-2010-hc	12.05	12.00	**11.95**
tw-2010	12.18	12.13	**12.09**
uk-2007-02-hc	0.95	0.92	**0.90**
uk-2007-02	1.23	1.20	**1.18**

Table 7.3: Effects of changing window size.

Next, we compare the effect of changing the window size W, choosing between values of 16, 32, and 64. The results are reported in Table 7.3. They show that increasing window size gives significant, although diminishing, savings on compressed size.

Finally, we compare the effect of changing the number of iterations through which reference lists are chosen (see Subsection 7.2.2), varying between 1 (corresponding to only using the simple fixed model) to 3. The results are shown in Table 7.4. They show that using a non-fixed model provides significant savings compared to the fixed one. On the other hand, further refinement of this model does not improve the compressed size, and is thus not worth the extra encoding effort.

As a consequence of these results, we perform further experiments using $k = 4$, $i = 1$, $j = 0$, $W = 32$, and 2 rounds of reference selection. We remark that $W = 64$ would have achieved better compression, but the WebGraph dataset was compressed using $W = 32$. We therefore pick this value for ease of comparison.

name	Size (bits per edge)		
	1 iter.	2 iter.	3 iter.
cnr-2000-hc	1.87	**1.84**	**1.84**
cnr-2000	2.24	**2.19**	**2.19**
in-2004-hc	1.33	**1.31**	**1.31**
in-2004	1.69	**1.65**	**1.65**
eu-2005-hc	2.49	**2.46**	**2.46**
eu-2005	2.89	**2.83**	**2.83**
uk-2002-hc	1.38	**1.36**	**1.36**
uk-2002	1.76	**1.72**	**1.72**
tw-2010-hc	12.00	**11.97**	**11.97**
tw-2010	12.13	**12.12**	**12.12**
uk-2007-02-hc	0.92	**0.91**	**0.91**
uk-2007-02	1.20	**1.18**	**1.18**

Table 7.4: Effects of changing number of iterations for reference list selection.

name	bits/edge greedy	bits/edge approx
cnr-2000	2.49	**2.24**
in-2004	1.82	**1.69**
eu-2005	3.18	**2.89**
uk-2002	1.95	**1.76**
tw-2010	12.29	**12.13**
uk-2007-02	1.36	**1.20**

Table 7.5: Comparison of the compressed size achieved by using the greedy algorithm used by WebGraph for reference selection and the size achieved by our approximation algorithm described in Subsection 7.2.2.

7.3.3 Effect of Approximation Algorithm and Context Modeling

We evaluate the gain from using the improved algorithm for reference selection (in Subsection 7.2.2), as opposed to the simple greedy algorithm used by WebGraph. The results are shown in Table 7.5. We remark that, as the reference selection is employed only when list decompression is supported, the table does not report results for the `-hc` version of the graphs.

We also report the effects of disabling Zuckerli's context model, by using the same

name	bits/edge	
	no ctx model	default
cnr-2000-hc	2.17	**1.84**
cnr-2000	2.47	**2.19**
in-2004-hc	1.55	**1.31**
in-2004	1.86	**1.65**
eu-2005-hc	2.84	**2.46**
eu-2005	3.14	**2.83**
uk-2002-hc	1.58	**1.36**
uk-2002	1.92	**1.72**
tw-2010-hc	13.21	**11.97**
tw-2010	13.27	**12.12**
uk-2007-02-hc	1.04	**0.91**
uk-2007-02	1.31	**1.20**

Table 7.6: Effects of disabling Zuckerli's context model.

probability distribution for all the entropy coded symbols. The results are shown in Table 7.6.

The results show that the gains from the approximation algorithm are significant, reaching up to 12% for web graphs, and also providing some benefits for social networks like `tw-2010`. The gains from the context model are similar.

We remark that this improvement is significant in a lossless compression context. In comparison, one of the most well-known advances in general purpose compression, the Burrows-Wheeler Transform [30], achieved roughly a 16% size reduction compared to previous approaches.

7.3.4 Compression Results and Resource Usage

For the chosen set of parameters, we report the compression speed and the resulting compression ratio on various graphs. We also compare the resulting compressed size with the ones achieved by WebGraph and by Graph Compression By BFS (GCBFS), k^2-trees, List Merging (LM) and Backlinks. To perform this comparison, we use the files available from the WebGraph corpus itself, without any recompression, and the implementation of GCBFS that is available at https://github.com/drovandi/GraphCompressionByBFS, with parameters $l = 10000$ for full decompression and $l = 8$ for list decompression. We include two publicly available implementation for k^2-

name	compression speed	bits/edge										
	($10^6 e/s$)	Zuckerli	WebGraph		GCBFS		k^2-trees		LM		Backlinks	
cnr-2000-hc	1.01	1.84	2.45	133%	1.88	102%	-		2.28	123%	-	
cnr-2000	0.89	2.19	3.12	142%	2.72	124%	3.12	142%	3.46	158%	10.51	480%
in-2004-hc	1.19	1.31	1.76	134%	1.42	108%	-		1.74	132%	-	
in-2004	1.03	1.65	2.15	130%	2.16	131%	2.25	136%	2.60	157%	9.62	583%
eu-2005-hc	1.03	2.46	3.16	128%	2.81	114%	-		2.34	95%	-	
eu-2005	0.97	2.83	3.72	131%	3.50	124%	3.23	114%	3.48	123%	10.26	362%
bn-jung-hc	1.04	2.07	2.57	124%	4.97	240%	-		2.86	138%	-	
bn-jung	1.01	2.41	2.96	123%	4.78	200%	-		3.21	133%	3.52	146%
uk-2002-hc	1.15	1.36	1.80	132%	1.71	126%	-		1.78	130%	-	
uk-2002	1.03	1.72	2.24	130%	2.43	141%	2.26	131%	2.66	154%	10.54	613%
hw-2009-hc	0.69	4.26	4.80	113%	7.29	171%	-		5.11	120%	-	
hw-2009	0.60	4.47	4.94	111%	7.51	168%	6.76	151%	5.83	124%	4.76	106%
tw-2010-hc	0.50	11.97	13.89	116%	15.34	128%	-		13.75	115%	-	
tw-2010	0.42	12.12	14.46	119%	15.21	125%	19.53	161%	15.70	129%	15.43	127%
pp-miner-hc	1.02	3.48	3.80	109%	4.60	132%	-		4.34	124%	-	
pp-miner	0.99	3.81	4.37	114%	4.60	120%	-		5.19	136%	3.73	98%
uk-2007-02-hc	1.63	0.91	1.18	130%	1.28	141%	-		1.06	116%	-	
uk-2007-02	1.53	1.18	1.56	132%	1.80	152%	2.58	219%	1.62	137%	8.03	681%
eu-2015-hc	1.46	0.74	0.89	120%	-		-		-		-	
eu-2015	1.34	0.92	1.20	130%	-		-		-		-	

Table 7.7: Comparison of compressed size between Zuckerli, WebGraph, Graph Compression with BFS, k^2-trees, Backlinks and LM (up to LM-16 for random access, up to LM-64 for high compression). We report compression speed and bits per edge for Zuckerli, and bits per edge and relative size with respect to Zuckerli for the other methods. The GCBFS encoder crashed when compressing eu-2015. Other methods were not run on eu-2015 because the available implementations couldn't handle graphs of this size within the available RAM. The k^2-tree compression code crashed on pp-miner and bn-jung.

trees: the one at https://lbd.udc.es/research/k2tree/ and the one available in the SDSL-lite library [51], modified to not consider the cost of the rank data structures, which are only required for navigation and are not accounted for in the other algorithms; we report the best result of the two implementations, for all graphs where they both ran to completion. In particular, the implementation at https://lbd.udc.es/research/k2tree/ did not successfully compress `tw-2010` and `uk-2007-02`. Experiments have been run with $k = 2, 3, 4$ and the best result was reported. For the LM algorithm the implementation provided by the authors was used; a chunk size of 16 was chosen for list decompression (which provides access times comparable to WebGraph), while for full decompression a chunk size of 64 was chosen (the same setting used in the original paper). The Backlinks algorithm uses the implementation available at https://github.com/snuke/backlinks-compression.

The results are shown in Table 7.7. They show that Zuckerli typically achieves 20% to 30% size savings when compared to WebGraph on web graphs and brain networks, and 10% to 15% size savings on social and protein interaction networks. In comparison, GCBFS achieves worse compression ratios than WebGraph in the larger datasets (`hw-2009`, `tw-2010`, `uk-2007`), and worse compression ratios than Zuckerli in all datasets (by up to 42%). WebGraph achieves significantly better compression than LM in the "list decompression" cases, and similar compression in the "full decompression" cases. Backlinks achieves significantly worse compression than WebGraph in all cases but three. Moreover, the decompression speed reported in the introductory paper for most algorithms is comparable with the one of WebGraph. Thus, we decide to run the remaining experiments comparing only with WebGraph, given that decompression speed is not the focus of Zuckerli as long as it is acceptable for general use.

We also compare Zuckerli's compression ratios to those achieved by k^2-trees and 2D-Block Trees [24]. While those data structures allow for single edge queries and reverse neighbour listing, Zuckerli only allows, in its least dense configurations, for individual forward adjacency list queries. Thus, the methods are not directly comparable. However, according to the results reported in [24] and in Table 7.7, both representations are significantly less dense than Zuckerli, with the best of the two producing compressed representations bigger by 30% or more in most cases. Further, according to the reported speed, the faster of the methods is able to process roughly 200 thousand edges per second, due to the intense use of sophisticated succinct data structures causing many cache misses, which is orders of magnitude slower than Zuckerli.

Finally, while we did not perform a direct comparison with LogGraph [13], we remark that while it offers improved performance for list access compared to WebGraph, it does

name	degree	reference	block	residuals first	residuals oth.	total
cnr-2000-hc	0.24	0.23	0.36	0.34	0.65	1.84
cnr-2000	0.27	0.24	0.33	0.41	0.92	2.19
in-2004-hc	0.20	0.17	0.26	0.24	0.44	1.31
in-2004	0.22	0.19	0.24	0.32	0.68	1.65
eu-2005-hc	0.15	0.14	0.40	0.32	1.45	2.46
eu-2005	0.16	0.14	0.36	0.38	1.79	2.83
uk-2002-hc	0.19	0.16	0.23	0.24	0.54	1.36
uk-2002	0.20	0.16	0.21	0.32	0.80	1.72
hw-2009-hc	0.05	0.02	0.34	0.09	3.73	4.26
hw-2009	0.05	0.02	0.33	0.10	3.95	4.47
tw-2010-hc	0.15	0.09	0.30	0.61	10.21	11.97
tw-2010	0.15	0.09	0.27	0.62	10.29	12.12
uk-2007-02-hc	0.09	0.07	0.14	0.14	0.42	0.91
uk-2007-02	0.09	0.07	0.12	0.19	0.60	1.18

Table 7.8: Breakdown of bit allocation, reported as bits per edge.

not achieve better compression ratios, as reported in [13]. [13] also shows experimental evidence of WebGraph exhibiting higher compression ratio than [13, 1, 17].

We also explore how the bit budget of Zuckerli is spent across the various parts of the graph that get encoded: degrees, references, blocks, and residuals, with the first residual being considered separately. The results are shown in Table 7.8. They show a remarkable difference between web graphs and social networks. Indeed, in social networks, almost all the bits are spent encoding residuals, while in web graphs the fraction of bits used for residuals is not as significant. This can be explained by the greater effectiveness of the block copying mechanism on web graphs, due to greater similarity in outgoing adjacency lists.

7.3.5 Performance Evaluation

We evaluate the performance characteristics of Zuckerli by comparing its running time and memory usage for running depth-first and breadth-first traversals with WebGraph (only for the variants that allow access to single lists), as well as with uncompressed graphs, as a baseline. Those algorithms were chosen to obtain real-world access patterns for adjacency lists, and every list was decompressed exactly once. The running time and

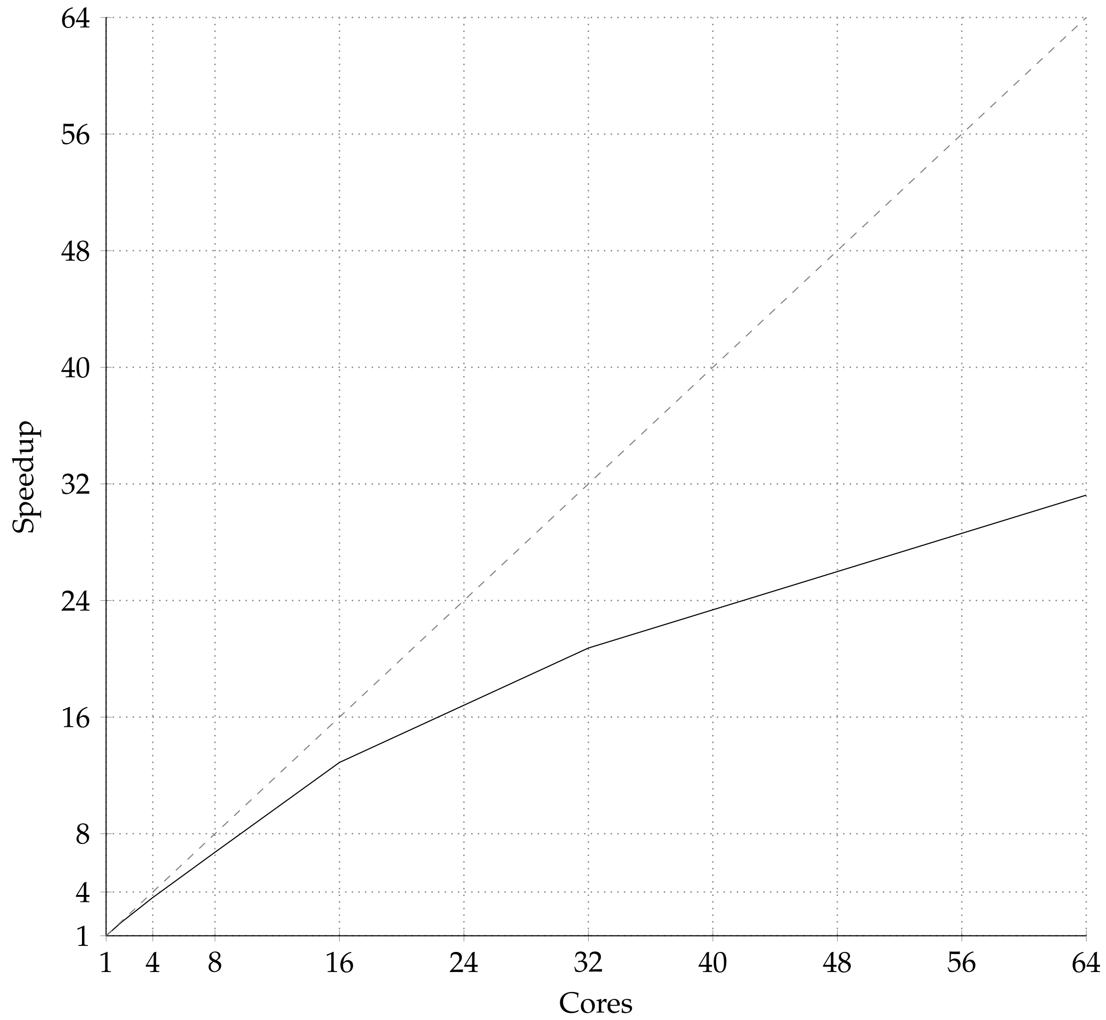


Figure 7.2: Speedup obtained by Zuckerli when computing the sum of the endpoints of all the edges using a variable number of cores on `uk-2007-02`. The dashed line represents the ideal speedup.

name	uncompressed			Zuckerli			WebGraph		
	time	μs/adj list	memory	time	μs/adj list	memory	time	μs/adj list	memory
`cnr-2000` DFS	15	0.04	17	300	0.92	10	395	1.21	186
`cnr-2000` BFS	13	0.04	17	302	0.92	10	389	1.19	181
`in-2004` DFS	72	0.05	79	1319	0.95	32	1300	0.94	262
`in-2004` BFS	68	0.05	79	1326	0.95	32	1392	1.00	408
`eu-2005` DFS	89	0.10	84	1278	1.48	30	1542	1.78	293
`eu-2005` BFS	80	0.09	84	1285	1.49	30	1764	2.04	381
`uk-2002` DFS	2791	0.15	1315	20 808	1.12	422	17 974	0.97	1778
`uk-2002` BFS	1556	0.08	1314	21 256	1.14	431	19 865	1.07	1956
`hw-2009` DFS	328	0.28	458	3922	3.44	147	6499	5.70	451
`hw-2009` BFS	320	0.28	458	3871	3.39	148	6366	5.58	429
`tw-2010` DFS	11 403	0.27	6069	115 120	2.76	5094	196 588	4.71	13 377
`tw-2010` BFS	11 400	0.27	6156	114 356	2.74	5154	192 121	4.61	13 587
`uk-2007-02` DFS	13 154	0.11	16 286	154 338	1.40	2883	177 945	1.61	2248
`uk-2007-02` BFS	13 155	0.11	16 287	156 467	1.42	2936	179 206	1.62	2781

Table 7.9: Running time (in milliseconds) and memory usage (in MB) for running breadth-first and depth-first search on both the uncompressed and the compressed representations (both with Zuckerli and WebGraph) of various graphs. We also report the average time (in μs) to access each adjacency list.

name	Zuckerli		WebGraph	
	time	memory	time	memory
`cnr-2000-hc`	0.03	5	0.36	107
`cnr-2000`	0.03	4	0.36	109
`in-2004-hc`	0.14	7	0.66	179
`in-2004`	0.14	6	0.63	182
`eu-2005-hc`	0.18	10	0.67	179
`eu-2005`	0.18	9	0.71	176
`uk-2002-hc`	2.20	54	4.93	763
`uk-2002`	2.16	65	5.08	829
`hw-2009-hc`	1.35	64	2.36	175
`hw-2009`	1.33	65	2.38	171
`tw-2010-hc`	28.19	2415	36.53	1308
`tw-2010`	24.50	2439	35.65	1719
`uk-2007-02-hc`	21.32	445	46.61	1701
`uk-2007-02`	20.84	573	49.25	1617

Table 7.10: Running time (in seconds) and memory usage (in MB) for decompressing the graphs sequentially with Zuckerli and with Webgraph.

the memory usage are reported in Table 7.9. We also compare the time and memory usage for running a full sequential decompression of the graphs, with results reported in Table 7.10.

From these comparisons, it emerges that the memory usage for decompression and random access required by WebGraph and Zuckerli is very different, with both methods using less memory in some situations. This can be explained by the different language of the implementation (C++ and Java), as well as the fact that WebGraph uses *lazy iteration* on adjacency lists, to avoid decompressing them fully to memory. While this can in principle be supported by Zuckerli, it was not implemented in this version of the code.

Regarding running time, Zuckerli is often faster than WebGraph. This is due to the fact that Zuckerli requires less memory bandwidth than WebGraph (as it uses less bits for compression), and that it is written in highly optimized C++ code.

Finally, to evaluate the scalability of Zuckerli on multiple cores, we wrote a simple program that computes the sum of all endpoints of all edges of a graph, and we ran it on `uk-2007-02` using 1, 2, 4, 8, 16, 32 and 64 cores. The results are shown in Figure 7.2 and show Zuckerli's good scalability; the speedup is likely limited by memory bandwidth.

7.4 Further improvements on the Zuckerli scheme

We will now show two improvements over the Zuckerli scheme described so far, for full decompression. We achieve those improvements by using the context modeling techniques from Section 4.1 and Section 4.2, as well as a new reference selection method.

7.4.1 Tree-based context modeling

We apply the tree building heuristic from Section 4.2, followed by the clustering heuristic from Section 4.1, to degrees, reference offsets, block counts, block sizes and residuals, defining $\mathcal{C}$ for each of them as follows:

- For degrees, $\mathcal{C}$ is a 3-tuple formed by the current node id i, the delta between the degree of node $i-1$ and $i-2$, and the degree of node $i-1$.
- For reference offsets, $\mathcal{C}$ is a 3-tuple formed by the current node id i, the reference offset of $i-1$ and the degree of the current node (because we expect a lower degree node to be less likely to use a reference).
- For block counts, $\mathcal{C}$ is a pair containing the current node and its reference offset.
- For block sizes, $\mathcal{C}$ for the j-th block of node i contains i, the reference offset for node i, j, the number of remaining blocks for the current node, $j \mod 2$ (since even and odd blocks have different meanings), the size of block $j-1$ and the size of block $j-2$.
- For residuals, $\mathcal{C}$ for the j-th residual of node i contains i, j, the value with respect to which the current residual is delta-coded, the number of remaining residuals to decode, and the length of the previous gap between residuals.

As an example, when encoding the residuals in the adjacency list from Figure 7.1 we have the following values for $\mathcal{C}$:

- $[7,0,7,4,0]$ for encoding -4: this is the first residual edge, hence the previous gap is set to 0, and delta-encoding is done with respect to the current node index.
- $[7,1,5,3,9]$ for encoding 3: this is the second residual edge, hence the previous gap might be negative and is thus represented using the bijection of Equation (7.1).
- $[7,2,9,2,3]$ for encoding the first 0.
- $[7,3,13,1,0]$ for encoding the second 0.

Algorithm 6: Algorithm to get candidates for reference list selections, using a search buffer of size k.

```
Function Init():
    last ← n × k array filled with n;
    last_idx ← n-sized array filled with 0;
Function UpdateAfterNode(i):
    /* used after finding candidates for i and before i+1 */
    for x ∈ N+(i) do
        last[x][last_idx[x]] ← i;
        last_idx[x] ← (p+1) mod k;
Function GetCandidates(i):
    candidates ← empty hash-map;
    for x ∈ N+(i) do
        for v ∈ last[x] do
            if v ≠ n then
                /* missing entries are treated as 0 */
                candidates[v] ← candidates[v] + 1;
    return keys from candidates with the W highest values;
```

7.4.2 Reference selection algorithm

The new reference selection algorithm differs from the one used in Zuckerli by, instead conducting an exhaustive search among the previous W nodes to find the one with the best size reduction, selecting candidate nodes by the following procedure:

- For each edge of node i, increment a counter by 1 for the last k nodes in the graph that also have that edge.
- Sort all nodes by decreasing order of their counter.
- Select the top W nodes in the resulting order.

Moreover, we also tweak the heuristic for deciding the best candidate, by adding to the cost a term that is logarithmic in the gap between the reference list and the current list: this estimates the cost of encoding the reference id itself.

The above procedure can be executed in $O(mk)$ time and using $O(n)$ extra memory: it is sufficient to keep a buffer of size k for the last k occurrences of a given node; then, for every node there will be $\leq \Delta_i k$ elements to sort; as we only need to know the top W elements, this can be done using for example the Quickselect algorithm, for a

name	Zuckerli	tree	tree+ref
`cnr-2000-hc`	1.89	1.57	1.53
`cnr-2000-nat-hc`	2.07	1.58	1.47
`in-2004-hc`	1.33	1.14	1.14
`in-2004-nat-hc`	1.56	1.22	1.12
`bn-jung-hc`	2.07	1.97	1.92
`uk-2002-hc`	1.37	1.26	1.22
`uk-2002-nat-hc`	1.52	1.31	1.21
`tw-2010-hc`	12.10	11.42	11.45
`tw-2010-nat-hc`	13.63	12.67	12.61
`pp-miner-hc`	3.48	3.42	3.55
`sk-2005-hc`	1.26	1.06	0.99
`sk-2005-nat-hc`	1.97	1.49	1.10

Table 7.11: Effects of tree-based context modeling and of the new reference selection algorithm on Zuckerli compression.

running time of $O(\Delta_i k)$. A more detailed description of the algorithm may be found in Algorithm 6.

This heuristic is intended to find in the graph adjacency lists that share a large number of edges with the current list: indeed, if we consider the case of $k = \infty$, the counters computed by the heuristic correspond exactly to the size of the intersection between the current and candidate adjacency lists. In this sense, the algorithm shares some similarities with the shingle ordering heuristic [34], which tries to sort a network so that nodes with a high Jaccard similarity are close together.

However, the version of the algorithm with unbounded k has $O(mn)$ time complexity and $O(n^2)$ space complexity , which is not feasible for large graphs. Instead, we keep track of a sliding window of the last k adjacency lists in which each node occurs, similarly to how LZ77 matching is often done with a sliding window,

7.4.3 Experimental results

Table 7.11 reports the results of applying these two techniques to some of the graph used in the experimental evaluation of Zuckerli. We remark that tree-based context modeling has a very significant cost both for encoding and decoding time (about 10x and 3x respectively), and thus we conducted a smaller experimental evaluation, without

considering the list decompression case as access times would be impractical. We also studied the impact of these compression improvements over the graphs in *natural order*, i.e. without the usage of the LLP node reordering scheme [20], that gives significant density improvements to both WebGraph and Zuckerli.

We first discuss the changes on Web Graphs. From the results in Table 7.11, we can observe that when using the compression scheme using both tree context modeling and the new reference selection the gap between natural and LLP order is significantly reduced, in some cases with the natural order even achieving better density.

As an explanation for the improved performance, we observe that the objective of LLP is to bring closer together nodes with similar adjacency lists, while at the same time reducing the gap between node IDs in each list. However, with the new reference selection algorithm it is no longer as important for nodes to be close together; moreover, tree-based context modeling can use the destination node ID of the previously-decoded edge, which allows the context model to encode information about the gaps in adjacency lists.

The results in Table 7.11, we can conclude that tree-based context modeling further reduces the compressed size by a $8-15\%$ factor for Web Graphs, and the new reference selection algorithm gives a further $3-7\%$. The improvements are even greater for graphs in natural order, achieving up to 80% size reduction in some cases and often producing the same size, or smaller, as when using LLP.

These conclusions also extend to brain networks such as `bn-jung`.

Finally, we observe that these techniques also benefit social networks such as `tw-2010` and protein interaction networks such as `pp-miner`, although the improvement is not as significant.

However, both graphs show some degradation when switching to the new reference algorithm. The worse performance in these cases can likely be explained by the different structure of the networks; it is interesting to note that, as seen in Table 7.7, these are also the graphs where Backlinks has the best results compared to Zuckerli.

CHAPTER 8

CONCLUSIONS

IN THIS THESIS we presented a theoretical and practical study of compression of large graphs, as well as novel techniques for encoding integers and for managing large context spaces in entropy coding.

From the theoretical perspective, we studied six different graph models, deriving lower bounds on their entropy and providing polynomial-time compression algorithms that are able to match (or almost match) these lower bounds up to lower-order terms, thus proving their optimality. In particular, we generalized a result obtained in [72] on the entropy of graphs from the Preferential Attachment model, and proved that they can be compressed in an asymptotically optimal way in polynomial time.

From the practical perspective, we introduced a novel compression scheme for large graphs that achieves significant space savings for compression of web graphs, brain networks, social networks and protein interaction networks compared to state-of-the-art schemes, such as the WebGraph framework, while retaining high performance for decoding operations. Experimental results show that the savings can be quantified as being about 25% on web graphs and brain networks, 12% on social networks and protein interaction networks, both for the full and the list decompression use cases. This improvement was achieved in part thanks to the novel integer encoding scheme that was introduced and analyzed in this thesis.

We introduced novel techniques for managing a large context space when doing

higher order entropy coding, and applied them to the proposed graph compression scheme. We also proposed a new reference-finding algorithm for graphs that is less reliant on the specific ordering of the nodes in the graph. The combination of these two techniques improves the compression density for web graphs and brain networks in the full decompression case by a further 12% to 27%, and for social networks and protein interaction networks by about 5%.

The proposed scheme can achieve significantly improved compression density on graphs in *natural order*, i.e. graphs that have not been permuted to fit a specific compression scheme. These improvements are often significant enough to result in graphs in natural order that have the same compressed size as graphs permuted using LLP [20].

8.1 Future work

An interesting question that comes natural when observing the compression density achieved by the tree-based context modeling techniques is whether those techniques can be applied to other compression tasks with similar benefits.

We know from JPEG XL that tree-based context modeling can achieve state-of-the-art compression ratios for images. A text compressor using similar techniques could potentially achieve significant improvements over existing text compressors.

Further improvements in compression ratio may be achieved by employing different strategies for tree constructions. Techniques from machine learning, such as genetic algorithms and gradient-based methods, could allow construction of better context trees while still allowing manageable complexity.

Another direction of improvement for tree-based context modelling is studying how to enhance its decompression speed; for this, one possibility would be for example to use a compiler to produce optimized executable code that represents a specific tree, as well as combining multiple levels of the decision tree together to make better use of the superscalar architectures of modern CPUs.

Regarding graph compression specifically, an interesting future direction of study is in the domain of graph permutations and, more specifically, in investigating whether a heuristic for graph reordering that is based on the same ideas as the novel reference selection heuristic proposed in Section 7.4 would provide better compression compared to other state-of-the-art permutation techniques.

Finally, it would be interesting to study how to extend the Zuckerli model to labeled and/or weighted graphs, as those graphs tend to be more used in real-world applications and require inherently more memory.

BIBLIOGRAPHY

[1] Micah Adler and Michael Mitzenmacher. Towards compressing web graphs. In *Proceedings DCC 2001. Data Compression Conference*, pages 203–212. IEEE, 2001.

[2] Jyrki Alakuijala, Ruud van Asseldonk, Sami Boukortt, Martin Bruse, Iulia-Maria Comșa, Moritz Firsching, Thomas Fischbacher, Evgenii Kliuchnikov, Sebastian Gomez, Robert Obryk, Krzysztof Potempa, Alexander Rhatushnyak, Jon Sneyers, Zoltan Szabadka, Lode Vandervenne, Luca Versari, and Jan Wassenberg. JPEG XL next-generation image compression architecture and coding tools. In Andrew G. Tescher and Touradj Ebrahimi, editors, *Applications of Digital Image Processing XLII*, volume 11137, pages 112 – 124. International Society for Optics and Photonics, SPIE, 2019.

[3] Jyrki Alakuijala and Lode Vandevenne. Data compression using zopfli. *Google, Tech. Rep.*, 2013.

[4] Vo Ngoc Anh and Alistair Moffat. Local modeling for webgraph compression. In *2010 Data Compression Conference*, pages 519–519. IEEE, 2010.

[5] Alberto Apostolico and Guido Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[6] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. Technical report, Stanford, 2006.

[7] Yasuhito Asano, Tsuyoshi Ito, Hiroshi Imai, Masashi Toyoda, and Masaru Kitsuregawa. Compact encoding of the web graph exploiting various power laws. In *International Conference on Web-Age Information Management*, pages 37–46. Springer, 2003.

[8] Yasuhito Asano, Yuya Miyawaki, and Takao Nishizeki. Efficient compression of web graphs. In *International Computing and Combinatorics Conference*, pages 1–11. Springer, 2008.

[9] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697, 2016.

[10] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[11] David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[12] Maciej Besta and Torsten Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations, 2018.

[13] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. Log(graph): a near-optimal high-performance graph representation. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–13, 2018.

[14] Krishna Bharat, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. The connectivity server: Fast access to linkage information on the web. *Computer networks and ISDN Systems*, 30(1-7):469–477, 1998.

[15] Alankrita Bhatt, Chi Wang, Lele Wang, and Ziao Wang. Universal graph compression: Stochastic block models. *arXiv preprint arXiv:2006.02643*, 2020.

[16] Philip Bille, Inge Li Gørtz, and Søren Vind. Compressed data structures for range searching. In *International Conference on Language and Automata Theory and Applications*, pages 577–586. Springer, 2015.

[17] Daniel K Blandford, Guy E Blelloch, and Ian A Kash. Compact representations of separable graphs. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 679–688. Society for Industrial and Applied Mathematics, 2003.

[18] Guy E Blelloch and Arash Farzan. Succinct representations of separable graphs. In *Annual Symposium on Combinatorial Pattern Matching*, pages 138–150. Springer, 2010.

[19] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. BUbiNG: Massive crawling for the masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*, pages 227–228. International World Wide Web Conferences Steering Committee, 2014.

[20] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[21] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. Permuting web and social graphs. *Internet Mathematics*, 6(3):257–283, 2009.

[22] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[23] Paolo Boldi and Sebastiano Vigna. The WebGraph framework II: Codes for the world-wide web. In *Data Compression Conference, 2004. Proceedings. DCC 2004*, page 528. IEEE, 2004.

[24] Nieves R Brisaboa, Travis Gagie, Adrián Gómez-Brandón, and Gonzalo Navarro. Two-dimensional block trees. In *2018 Data Compression Conference*, pages 227–236. IEEE, 2018.

[25] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. k^2-trees for compact web graph representation. In *International Symposium on String Processing and Information Retrieval*, pages 18–30. Springer, 2009.

[26] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39:152–174, 2014.

[27] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer networks*, 33(1-6):309–320, 2000.

[28] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 95–106, 2008.

[29] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature reviews neuroscience*, 10(3):186–198, 2009.

[30] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Systems Research Center*, 1994.

[31] Adenauer G. Casali, Olivia Gosseries, Mario Rosanova, Mélanie Boly, Simone Sarasso, Karina R. Casali, Silvia Casarotto, Marie-Aurélie Bruno, Steven Laureys, Giulio Tononi, and Marcello Massimini. A theoretically based index of consciousness independent of sensory processing and behavior. *Science Translational Medicine*, 5(198):198ra105–198ra105, 2013.

[32] Arthur Cayley. A theorem on trees. *Quart. J. Math.*, 23:376–378, 1889.

[33] Venkatesan T Chakaravarthy, Vinayaka Pandit, Sambuddha Roy, Pranjal Awasthi, and Mukesh Mohania. Decision trees for entity identification: Approximation algorithms and hardness results. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 53–62, 2007.

[34] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, 2009.

[35] Francisco Claude and Susana Ladra. Practical representations for web and social graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1185–1190, 2011.

[36] Francisco Claude and Gonzalo Navarro. A fast and compact web graph representation. In *International Symposium on String Processing and Information Retrieval*, pages 118–129. Springer, 2007.

[37] Alessio Conte, Tiziano De Matteis, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. D2K: Scalable community detection in massive networks via small-diameter k-plexes. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1272–1281, 2018.

[38] Alessio Conte, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. Discovering k-trusses in large-scale networks. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[39] Colin Cooper and Alan Frieze. The cover time of the preferential attachment graph. *Journal of Combinatorial Theory, Series B*, 97(2):269–290, 2007.

[40] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1535–1544, 2016.

[41] Jarek Duda. Asymmetric numeral systems. *arXiv preprint arXiv:0902.0271*, 2009.

[42] Peter Elias. Universal codeword sets and representations of the integers. *IEEE transactions on information theory*, 21(2):194–203, 1975.

[43] P Erdös and A Rényi. On random graphs I. *Publ. math. debrecen*, 6(290-297):18, 1959.

[44] Paul Erdős and András Hajnal. On chromatic number of graphs and set-systems. *Acta Mathematica Academiae Scientiarum Hungarica*, 17(1-2):61–99, 1966.

[45] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, pages 128–140, 1741.

[46] Reza Fathi, Anisur Rahaman Molla, and Gopal Pandurangan. Efficient distributed community detection in the stochastic block model. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 409–419. IEEE, 2019.

[47] Moritz Firsching, Luca Versari, Sami Boukortt, and Jyrki Alakuijala. Compression. https://almanac.httparchive.org/en/2020/compression, 2020 (accessed December 14, 2020).

[48] Johannes Fischer and Daniel Peters. Glouds: Representing tree-like graphs. *Journal of Discrete Algorithms*, 36:39–49, 2016.

[49] Harold N Gabow and Herbert H Westermann. Forests, frames, and games: algorithms for matroid sums and applications. *Algorithmica*, 7(1-6):465, 1992.

[50] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[51] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[52] Szymon Grabowski and Wojciech Bieniecki. Tight and simple web graph compression. *arXiv preprint arXiv:1006.0809*, 2010.

[53] Szymon Grabowski and Wojciech Bieniecki. Merging adjacency lists for efficient web graph compression. In *Man-Machine Interactions 2*, pages 385–392. Springer, 2011.

[54] Jean-Loup Guillaume, Matthieu Latapy, and Laurent Viennot. Efficient and simple encodings for the web graph. In *International Conference on Web-Age Information Management*, pages 328–337. Springer, 2002.

[55] Michael Hardy. Bounds on sum of inverses of first n square roots. https://math.stackexchange.com/questions/2416918, 2017 (accessed December 14, 2020).

[56] Cecilia Hernández and Gonzalo Navarro. Compressed representation of web and social networks via dense subgraphs. In *International Symposium on String Processing and Information Retrieval*, pages 264–276. Springer, 2012.

[57] Cecilia Hernández and Gonzalo Navarro. Compressed representations for web and social graphs. *Knowledge and information systems*, 40(2):279–313, 2014.

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

[59] Guy Jacobson. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*, pages 549–554. IEEE Computer Society, 1989.

[60] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In *SODA*, volume 7, pages 575–584, 2007.

[61] Haim Kaplan, Shir Landau, and Elad Verbin. A simpler analysis of Burrows–Wheeler-based compression. *Theoretical Computer Science*, 387(3):220–235, 2007.

[62] Chinmay Karande, Kumar Chellapilla, and Reid Andersen. Speeding up algorithms on compressed web graphs. *Internet Mathematics*, 6(3):373–398, 2009.

[63] Hamid Khalili, Amir Yahyavi, and Farhad Oroumchian. Web-graph precompression for similarity based algorithms. In *Proceedings of the Third International Conference on Modeling, Simulation and Applied Optimization*, 2009.

[64] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[65] Soon-kak Kwon, A Tamhankar, and KR Rao. Overview of H.264/MPEG-4 part 10. *Journal of Visual Communication and Image Representation*, 17(2):186–216, 2006.

[66] Eduardo S Laber and Loana Tito Nogueira. On the hardness of the minimum height decision tree problem. *Discrete Applied Mathematics*, 144(1-2):209–212, 2004.

[67] Lawrence L Larmore and Daniel S Hirschberg. A fast algorithm for optimal length-limited huffman codes. *Journal of the ACM (JACM)*, 37(3):464–473, 1990.

[68] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[69] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[70] Fengying Li, Qi Zhang, Tianlong Gu, and Rongsheng Dong. Optimal representation for web and social network graphs based on k^2-tree. *IEEE Access*, 7:52945–52954, 2019.

[71] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM Computing Surveys (CSUR)*, 51(3):1–34, 2018.

[72] Tomasz Łuczak, Abram Magner, and Wojciech Szpankowski. Asymmetry and structural information in preferential attachment graphs. *Random Structures & Algorithms*, 55(3):696–718, 2019.

[73] Matt Mahoney. Rationale for a large text compression benchmark. *Retrieved (Aug. 20th, 2006) from: http://cs. fit. edu/mmahoney/compression/rationale. html*, 2006.

[74] Sebastian Maneth and Fabian Peternek. Compressing graphs by grammars. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 109–120. IEEE, 2016.

[75] Hossein Maserrat and Jian Pei. Neighbor query friendly compression of social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 533–542, 2010.

[76] David W Matula and Leland L Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM (JACM)*, 30(3):417–427, 1983.

[77] Alistair Moffat, Radford M Neal, and Ian H Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems (TOIS)*, 16(3):256–294, 1998.

[78] Alistair Moffat and Matthias Petri. Large-alphabet semi-static entropy coding via asymmetric numeral systems. *ACM Transactions on Information Systems (TOIS)*, 38(4):1–33, 2020.

[79] J Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[80] Richard Otter. The number of trees. *Annals of Mathematics*, pages 583–599, 1948.

[81] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[82] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[83] Christos H Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *Journal of computer and system sciences*, 43(3):425–440, 1991.

[84] Richard Clark Pasco. *Source coding algorithms for fast data compression*. PhD thesis, Stanford University CA, 1976.

[85] Nicola Prezza. On locating paths in compressed tries. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 744–760. SIAM, 2021.

[86] Heinz Prüfer. New proof of a sentence about permutations (a new prof. of a theorem on permutations). *archive of mathematics and physics*, 3:27, 1918.

[87] Sriram Raghavan and Hector Garcia-Molina. Representing web graphs. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 405–416. IEEE, 2003.

[88] Naila Rahman, Rajeev Raman, et al. Engineering the louds succinct tree representation. In *International Workshop on Experimental and Efficient Algorithms*, pages 134–145. Springer, 2006.

[89] Keith H Randall, Raymie Stata, Rajiv G Wickremesinghe, and Janet L Wiener. The link database: Fast access to graphs of the web. In *Proceedings DCC 2002. Data Compression Conference*, pages 122–131. IEEE, 2002.

[90] Neil Robertson and Paul D Seymour. Graph minors. XX. Wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

[91] A. H. Robinson and C. Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967.

[92] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[93] Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 134–149. SIAM, 2010.

[94] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[95] Quan Shi, Yanghua Xiao, Nik Bessis, Yiqi Lu, Yaoliang Chen, and Richard Hill. Optimizing k^2 trees: A case for validating the maturity of network of practices. *Computers & Mathematics with Applications*, 63(2):427–436, 2012.

[96] Kijung Shin, Amol Ghoting, Myunghwan Kim, and Hema Raghavan. Sweg: Lossless and lossy summarization of web-scale graphs. In *The World Wide Web Conference*, pages 1679–1690, 2019.

[97] Torsten Suel and Jun Yuan. Compressing the graph structure of the web. In *Proceedings DCC 2001. Data Compression Conference*, pages 213–222. IEEE, 2001.

[98] Luca Versari, Iulia-Maria Comsa, Alessio Conte, and Roberto Grossi. Zuckerli: A new compressed representation for graphs. *IEEE Access*, 8:219233–219243, 2020.

[99] Gregory K Wallace. The JPEG still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv, 1992.

[100] Rajiv Wickremesinghe, Raymie Stata, and Janet Wiener. Link compression in the connectivity server. Technical report, Technical report, Compaq systems research center, 2000.

[101] Herbert S Wilf and Nancy A Yoshimura. Ranking rooted trees, and a graceful application. In *Discrete Algorithms and Complexity*, pages 341–349. Elsevier, 1987.

[102] Frans MJ Willems, Yuri M Shtarkov, and Tjalling J Tjalkens. The context-tree weighting method: Basic properties. *IEEE transactions on information theory*, 41(3):653–664, 1995.

[103] Aaron D Wyner and Jacob Ziv. The sliding-window Lempel-Ziv algorithm is asymptotically optimal. *Proceedings of the IEEE*, 82(6):872–877, 1994.

[104] N. A. Yoshimura. *Ranking and Unranking Algorithms for Trees and Other Combinatorial Objects*. PhD thesis, University of Pennsylvania, USA, 1987.

[105] Marinka Zitnik, Rok Sosič, Sagar Maheshwari, and Jure Leskovec. BioSNAP Datasets: Stanford biomedical network dataset collection. http://snap.stanford.edu/biodata, August 2018.

[106] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.