Università degli Studi di Milano
Computer Science department

Large-scale graph compression via asymmetric numeral systems

Francesco Tomaselli

Supervisor:

Sebastiano Vigna

Co-Supervisor:

Paolo Boldi

Academic year 2021/2022

Acknowledgements

...

Abstract

This thesis focuses on the application of asymmetric numeral systems to large-scale graph compression.

Asymmetric numeral systems are a family of entropy encoders that obtains compression quality comparable to arithmetic coding, thus optimal for a given source of symbols, while having a decoding speed similar to Huffman coding. They were presented by Jarek Duda and are heavily used in a lot of different scenarios, such as Linux kernels, Facebook Zstandard, JPEG XL and many others.

This presents an application of such encoders on large-scale web and social graphs. The final proposed methodology is the result of three iterations and combines asymmetric numeral systems, instantaneous codes and patched frame of reference encoding.

The experimental results show how this methodology saves as much as 76 percent of space with respect to quasi-succinct representations and is capable of storing graphs in as low as 3.5 bits per link.

Contents

Chapter 1

Introduction

Graphs are one of the most versatile data structure in computer science and mathematics, finding an extreme variety of applications from path finding, to ranking in search engines, community detection and so on. One of the key requirements of graph analysis is the possibility to efficiently visit them, that is a pretty easy task when they are small, but not so much when millions of nodes and billions of arcs are involved. A web graph is a structure where nodes are made out of web pages, and there exist a link between nodes x and y if x contains an hyperlink to page y. On the other hand, social graphs are structures where nodes are connected via a social relation, for instance pages of Wikipedia, friends and followers on a social network, and so on. These types of graphs certainty store a lot of information, thus we need a way to visit them efficiently.

Why the classical representation do not work The classical ways of storing graphs are through adjiacency matrixes or lists: in the first case a $n \times n$ square matrix is considered and a boolean flag on cell i, j indicates whenever there exist an arc between node i and j, adjiacency lists represent a graph through a list of successors for each node. The classic representations are perfectly fine for small graphs, but occupy a huge amount of memory for large-scale ones. In fact, one of the main problems associated with web and social graphs is the dimensionality, we are talking about structures with possibly millions of nodes and billions of

arcs, thus a classical representation can not be loaded in memory. A possible workaround is to store it on disk and extract smaller parts of the graph when needed, but this is pretty slow and unpractical.

Data compression One solution to visit large-scale graphs is data compression. The goal is to store structures in less bits than a classical representation, and this often translates to finding ways to compress integers. There exists many techniques for data compression, and in the end they all involve integer compression. Some classical approaches are found in the application of instantaneous codes to encode a given integer source, those have been widely used to store all sort of data and are straightforward to apply. The idea is to encode an integer n in a codeword, where the length of it determines its probability, hence one need to assign shorter codewords to frequent integers.

Outside of the realm of instantaneous codes we find techniques to efficiently code blocks of integers, such as patched frame of reference, and structures that uses a number of bits close to the succinct bound such as Elias-Fano lists.

Other techniques do not rely on an intrinsic distribution of codewords and are tailored to specific sources, such as asymmetric numeral systems, that are a family of entropy encoders. They use optimal space for a given source, paying the price of the need to store additional information. This type of encoders are the main focus of this thesis, indeed we want to find out if these type of entropy encoders can be fruitfully applied to graph compression.

Graph compression has been already studied in the past, a prime example is the Webgraph framework, that exploits structural properties of web and social graphs to store them in a few bits per link. The approach relies on the empirical similarity of successors of close nodes, using reference compression to represent a node outgoing links: the idea is to fix a reference node, and represent a node outgoing links with a bitmask. The residuals, that are nodes not included in the reference, are written exploiting consecutivity, that creates a lot of consecutive intervals in a node successors list.

Another starting point for this study is the compression of inverted indexes. An

inverted index is a data structure that build the backbone of a search engines, the idea is to assign to each word the documents ids, plus other additional information, where it appears. As the reader can imagine this structure is pretty similar to an adjiacency list, as we are storing a list of integers for each term. Many techniques can be applied to store this type of indexes, but one work in particular exploits asymmetric numeral systems for the task, that suggests some key disadvantages to keep into consideration when applying ANS to large-scale datasets.

The proposed methodology The proposed methodology is a compression scheme for graphs, where asymmetric numeral systems are applied on the gap encoding of node's successors. We start with a naïve application to then refine it. One of the characteristics of entropy encoders is that they can be tailored to a statistical source, hence they offer optimal compression ratios, the tradeoff is that we need to store additional information to rebuild these ad hoc compressors. Indeed, ANS in particular uses symbols maps to store observed probabilities, and they can become quite large if a source exhibits a lot of different symbols.

The naïve approach fails exactly for this reason, as we are building a specialized model for each node in a graph, it generates optimal compression while wasting an enormous amount of space writing all the additional information required by each encoder on disk. To solve this problem, various approaches are employed, but the idea is to first limit the size of the symbols maps of the encoders. One way to do so is through symbols escaping, that consists in removing some symbols from the maps, encoding them with another compression method. The other point left to solve is the excessive amount of specialized models, as we are dealing with a different encoder for each node. The idea that comes to mind is clustering, that consists in finding some similarity relations between models, reducing a group of similar ones to a single encoder. For this purpose, a Gray code ordering is employed on the models symbols, to then partition them heuristically. Finally, the idea of escaping is applied on the clustered models, where we find a suitable set of symbols to remove from the frequency maps by minimizing a space estimation function.

The final result is a methodology that combines asymmetric numeral systems,

patched frame of reference and instantaneous codes to store large scale graphs. The results are promising and the compression schema is capable of storing web graphs in as low as 3.5 bits per link, saving 76 percent of the space required by quasi-succinct data structures. The access speed to the graph is also comparable to the existing methods, keeping in mind that entropy encoding and decoding is on the slower side of compression methods.

Experimental results and conclusions All the implementation is written in Java and publicly available, the project implements Webgraph interfaces making it easier to store and load graphs from the public datasets. Compression results are expressed in bits per link, and are always in between of the Webgraph methodology and the quasi-succinct representation. Speed is tested by randomly accessing nodes successors, and the obtained speed per link is comparable to the Webgraph methodology.

Disseration organization The dissertation is organized as follows: Chapter 2 on the facing page gives an overview of different compression techniques, starting from instantaneous codes for integers, ending with asymmetric numeral systems, patched frame of reference and Elias-Fano monotone lists. Chapter 3 on page 13 introduces the Webgraph compression scheme and a relevant study about inverted indexes compression. In Chapter 4 on page 21 the proposed methodology is discussed, explaining the challenges, the refinements and the final approach to graph compression. Chapter 5 on page 42 presents all the experimental results. We start by defining the graphs used in the experiments, that comes from the Webgraph archive and are a mix of web and social ones, to then proceed with the compression results on all those graphs and the speed of access tests.

Chapter 2

Compression techniques

We now introduce various compression techniques. The first Section talks about instantaneous codes, which are a way to encode an integer source. Later, other techniques are presented, namely asymmetric numeral systems, patched frame of reference and Elias-Fano monotone lists.

2.1 Instantaneous codes

Instantaneous codes have been widely used to store integer sources. The idea is to assign a codeword to each integer, the length of the word determines a probability distribution, and thus each code is better suited to a given integer source.

Other applications can be found in inverted indexes storing, a structure where for each word we store the documents where it appears. A common use of codes is to pre-process the document list by gap encoding and then store the gaps with some code on disk.

What is a code A code is a set $C \subseteq 2^*$ of binary words. We define an ordering by prefix of the words in 2^* as the following:

$$x \preceq y \iff \exists z \mid y = xz$$

Two words are incomparable if neither is a prefix of the other.

A code is said to be instantaneous if each pair of distinct words in the code are incomparable. This property assures that every given word w obtained by concatenation of words in the code is uniquely decodable. An example of instantaneous code is $\{0, 1\}$, while $\{0, 01, 1\}$ is not. Considering $w = 01001$, the first code decodes it successfully, while the second creates ambiguity between the words 01 and 0.

A code is complete if every word $w \in 2^*$ is comparable with some word of the code. If this property holds, no words can be added to the code without violating instantaneousness.

Kraft-McMillan inequality Let $C \subseteq 2^*$ be a code. If C is instantaneous, then:

$$\sum_{w \in C} 2^{-|w|} \leq 1$$

A code C is complete if and only if the equality holds. Also, given the, eventually infinite, sequence $t_0, t_1, \ldots, t_{n-1}, \ldots$, if this equation holds

$$\sum_{i=0}^{n} 2^{-t_i} \leq 1$$

there exists an instantaneous code made up of words $w_0, \ldots, w_n$, such that $|w_i| = t_i$.

2.1.1 Codes examples

Unary code This is one of the simplest codes, indeed an integer n is written as a sequence of n zeros followed by a single one. This code is not really optimal to use as is, but is an integral part of more performing ones. The length of the codeword associated with n is trivially $n + 1$.

Note that zeros and ones could be swapped leading to the same result, this would create consistency between integer and lexicographic ordering of the codewords.

Elias γ code A more sophisticated encoding ig γ. This time an integer n is encoded is a *length*, *payload* fashion. The latter is computed with the binary representation of n minus the leading one, for example:

$$123 = 1111011 \to 111011$$

γ then writes the length of the payload in unary, followed by it, thus:

$$123 \to 0000001|111011$$

It follows that for a generic integer n, the length of the code is $2\lfloor \log n \rfloor + 1$.

Elias δ code The idea here is the same as γ, but this time we replace the unary code to write the length of the payload with γ itself. Hence the length of the codeword for n is $2\log(\log(n+1)+1)+1+\log(n+1)$.

Note that we could go on in this way and define a code that uses δ to write the payload length, but there are no significant advantages.

Golomb code This code fixes a base b and writes n as the quotient of n/b in unary, followed by $\log b$ bits for the remainder. The length of the codewords are $\lfloor \frac{n}{b} \rfloor + 1 + \log b$.

A note about this code is that, if we fix the base at

$$b = \left\lceil -\frac{\log(2-p)}{\log(1-p)} \right\rceil$$

we create an optimal code for a geometric distribution of parameter p.

Varyable byte codes This type of representation encodes an integer in some number of bytes. The idea is to consider a single byte as a *continuation bit* plus a *payload*, containing part of an integer. To encode an integer we write it as blocks of 7 bits, where all but the last one have concatenation bit set to 0.

To decode, we iterate over the blocks until a continuation bit of 1 is reached, all the 7-bit parts are then merged through bit manipulation. The size of the blocks

n	γ	δ	Golomb	VByte
0	1	1	100	1000
1	010	0100	101	1001
2	011	0101	110	1010
3	00100	01100	0100	1011
4	00101	01101	0101	1100
5	00110	01110	0110	1101
6	00111	01111	00100	1110
7	0001000	00100000	00101	1111
8	0001001	00100001	00110	00011000
9	0001010	00100010	000100	00011001
10	0001011	00100011	000101	00011010

Table 2.1: Codewords examples. Golomb code uses $b = 3$, while VByte has a block size set to 4.

creates a tradeoff between compression and decoding speed, some popular choices can be 32, 16, 8 or 4 bits blocks.

2.2 Other techniques

We now present other compression techniques that do not fall into the realm of instantaneous codes. The first one is asymmetric numeral systems, the second is a methodology to store a list of integers exploiting binary magnitudes of integers, while the third is a quasi-succinct data structure for lists. Note that many other techniques exist, for the sake of brevity we present here only those that are relevant to this study.

2.2.1 Asymmetric numeral systems

Asymmetric numeral systems is a family of entropy encoders presented by Jarek Duda that achieves a compression ratio comparable to arithmetic coding, while having a processing cost similar to Huffman coding [6]. They are used in a wide range of applications, for instance, Facebook Zstandard compressor, JPEG XL, Linux kernel and so on.

Standard numeral systems In standard numeral systems, symbols are treated as if they carry the same amount of information. Fixing the binary case, we have two symbols, 0 and 1, each with a probability of 1/2, thus the optimal space required to store each becomes $\log(1/2)$. If we think about it, this is exactly what happens in the classical binary encoding, given x as the current sequence, it becomes $x' = 2x$ or $x' = 2x + 1$, if we add a zero or a one, thus each symbol takes exactly $\log(1/2)$ bits. The idea of Duda is to extend this concept to the asymmetric case, creating a similar scenario where we have many symbols each with a different probability.

Asymmetric case What the binary example shows us is that a single integer holds an encoded sequence, this is also the case for the asymmetric case. The idea is to encode a generic sequence of symbols in a single integer x, that will be called *state*. More specifically, fixed a symbol s that appears with probability p_s, this method obtain a new state $x' \approx x/p_s$. The result is an encoding that uses a number of bits close to the Shannon entropy of the source of the symbols and is thus optimal.

Let S be a symbol source, consider now a generic symbol s and its frequency F_s. We scale all the counts to a fixed range 2^d and obtain new frequencies f_s for each symbol, thus the probability for s is approximated by $f_s/2^d$.

We define the cumulative c_s for each symbol as the quantity:

$$c_s = \sum_{t<s} f_t, \quad f_t > f_s \to t < s$$

this permits to define the primitive $sym(n) = \max_{c_s \leq n} S$, with $n \in [1, 2^d]$. This is required both to encode and decode a symbol.

Encoding and decoding The state, initially set to zero, is updated at each encoding and decoding, the primitives are defined as:

$$encode(state, s) = \lfloor state/f_s \rfloor * M + c_s + state \bmod f_s$$
$$decode(x) = \langle \lfloor (state - r)/M \rfloor * f_s - c_s + r, s \rangle$$

where $M = 2^d$, $r = 1 + (state - 1) \bmod M$, and the decoded $s = sym(r)$.

For the above definition, the encoded sequence is decoded in reverse order. Also, in practice, once the state overflows, it is normalized by storing the current state and resetting it to zero. This worsens the compression ratio but permits handling integers with a fixed length.

Example Lets assume to have a list of symbols:

$$1, 1, 1, 2, 2, 1, 2, 3, 1, 1, 3, 2, 1, 2, 3, 1$$

we can compute the probability of each symbol and scale it to a fixed range, assume it to be 1024, the cumulative can also be computed:

Sym	F_s	f_s	c_s
1	8/16	512	0
2	5/16	320	512
3	3/16	192	832

We can now start to encode the symbols:

$$\begin{aligned}
\text{encode}(0, 1) &= (0/512) * 1024 + 1 + 0 = 1 && (1\text{bit}/1)\\
\text{encode}(1, 1) &= (1/512) * 1024 + 1 + 1 = 2 && (2\text{bit}/2)\\
\text{encode}(2, 1) &= (2/512) * 1024 + 1 + 2 = 3 && (3\text{bit}/3)\\
\text{encode}(3, 2) &= (3/320) * 1024 + 513 + 3 = 516 && (11\text{bit}/4)\\
\text{encode}(516, 2) &= (516/320) * 1024 + 513 + 196 = 1733 && (12\text{bit}/5)\\
\text{encode}(1733, 1) &= (1733/512) * 1024 + 1 + 197 = 3270 && (13\text{bit}/6)\\
\text{encode}(3270, 2) &= (3270/320) * 1024 + 513 + 70 = 10823 && (15\text{bit}/7)\\
&\dots
\end{aligned}$$

The result is an integer of 30 bits storing 16 elements, overall less than 2 bits per element are used.

2.2.2 Patched frame of reference

This mechanism operates in the context of storing elements in a block of size B. Let b_i be the i-th element of the block, one could use a fixed representation set at $\max_{i=0,\dots,B} \lceil \log b_i \rceil$ bits to store every element. This is perfectly fine but it might be a waste of space in cases where a few outliers are present. For instance, given the block of 8 elements:

$$12, 23, 12, 1, 2, 176432, 3, 2$$

the method would use $\lceil \log 176432 \rceil = 18$ bits for each element, when we can clearly see that $\lceil \log 23 \rceil = 5$ bits would be enough to store all elements apart from the maximum.

We can build on top of this idea and compute a binary magnitude l that covers most but not all elements in a block b. The residuals are written in a another way at the end of the block. To determine l a search over likely values is performed and the best one in terms of space is selected. Many variants exists that encode exceptions is various ways, preferring speed, or compression ratio [8][12].

2.2.3 Elias-Fano monotone lists

This method was proposed by Peter Elias and it consists of a quasi-succinct representation of monotone sequences [7]. Quasi-succinct structures use a space that is close to the information-theoretical lower bound, and provides on average constant-time access to the stored data. In a more recent paper, this structure is used to store inverted indexes, obtaining good results both in compression and speed [11].

High/low bits representation Assume to have a monotonically increasing sequence of n values that are limited by an upper bound u:

$$0 \leq x_0 \leq x_1 \leq \cdots \leq x_{n-1} \leq u$$

Each element is split into high and low bits as follows:

- the lower $l = \max\{0, \log(u/n)\}$ bits are written in a *lower-bits array* sequentially;
- the upper bits are stored in the *upper-bits array* written by gap in unary code.

The upper/lower bits techniques uses at most $2 + \lceil \log(n/u) \rceil$ bits per element. The information-theoretical lower bound for a monotone list of n element in a universe of u elements is

$$\left\lceil \log \binom{u+n}{n} \right\rceil \approx n \log\left(\frac{u+n}{n}\right)$$

Elias proved that this representation is really close to the succinct bound, hence we can call it quasi-succinct.

Primitives To get the i-th element from the list, we perform i unary codes reads to obtain the upper part of the element, combining them with the lower bits found on position i in the lower array. One can speed up the search by fixing a quantum q and store in a table the position of the q-th zero or one. This permits to skip some part of the list and complete the remainder of the search sequentially.

Chapter 3

Graphs and inverted indexes

This chapter deals with graph and inverted index compression. The two tasks are closely related, as in a graph one must encode the successors' ids for each node, while in an inverted index each word is associated with the documents id where it appears.

The first Section introduces WebGraph, a compression method for large scale graphs coming from the web, while the second talks about relevant applications of asymmetric numeral systems to inverted indexes compression.

3.1 WebGraph

We now introduce an established graph compression methodology for web graphs that makes heavy use of their properties to store them in a highly compressed way [2][3]. This is the technique on top of which the proposed methodology is built and is referred to from now on as BV.

What is a web graph A web graph is a graph where there exists a node for each URL, and an arc between nodes x and y if a hyperlink in x leads to y. Such graphs can be really useful for a lot of reasons, for instance, one could improve search engine performances, detect cyber communities, improve crawlers and so on.

The problem that rises when dealing with such structures is the huge dimensionality. We are talking about million of nodes and billion of arcs. Such cardinalities make it impossible to hold the entire graph in memory, hence some sophisticated techniques are required to visit it. Although streaming could be an option, that means loading a particular part of the graph from disk only when necessary, data compression can be employed, decoding successors from a compressed representation that lives in memory instead of disk.

Web graph properties Assume for now that these graphs are ordered in a lexicographic way by node URL. Some empirical properties can be observed when dealing with web graphs, that are *locality*, *similarity* and *consecutivity*.

The first one refers to the fact that, on a given web page, most of the hyperlinks are navigational, i.e. they link a page to others on the same website. This property ensures that by ordering nodes in a lexicographic way, a lot of arcs are between close ones.

The second property means that pages close to each other in lexicographic order exhibit a lot of common successors. This is a consequence of locality, as close pages are usually of the same website, thus they share a lot of navigational links. This property is really strong and usually consecutive nodes share almost all successors of nothing at all.

Consecutivity is a property for which a lot of successors for a node are consecutive, so the adjacency list for a URL shows a lot of intervals.

Differences between social and web graphs While web graphs are a great source of information, the compression of social graphs is also relevant. A social graph is a graph where nodes are connected by some sort of social relation, many examples range from social networks to Wikipedia pages.

The compression of those graphs is more complicated as the structural properties introduced for web graphs are less pronounced. Also, the introduced empirical properties are valid if web pages are sorted in lexicographic order, and this ordering is not present in social graphs. Hence ordering the nodes accordingly can be really beneficial for the compression of social networks.

Datasets The WebGraph framework permits to make public a lot of big graphs in the BV representation[1]. They come from various crawling activities and are a mix of social and web graphs of different dimensions [4]. For each crawling result, both the normal and transposed graph are present.

The proposed methodology loads big graphs through this framework, to then re-store them via asymmetric numeral systems.

3.1.1 Compression format

Suppose we have a web or social graph where nodes are assigned labels from 0 to $N-1$, and consider a lexicographic ordering by URL for web graphs and a suitable one chosen for social graphs. Let us consider the list of successors of a node x, $S(x)$. A first naïve representation would be to store them sorted. This is not optimal as no empirical property is exploited.

Computing gaps A first optimization is to compute gaps between them, where the first successor of x is written as $s_1 - x$, and each subsequent ones as $s_i - s_{i-1}$. Note that the first one might be negative, thus it is encoded as $2x$ if greater than zero, or $2|x| - 1$ otherwise.

Copy list A second optimization is to exploit similarity, the idea is to write a list of successors of a node as the difference between a preceding list. Fixed a predecessor y, we write the successors of the node x as a *copy list*, that have a 1 or 0 in the i-th position whenever $S(x)$ contains the i-th successor of y. Some noded might not be represented this way, thus they are written as a list of extra nodes. The choice of the reference node is made inside a window of size W, which creates a tradeoff between retrieval speed and compression quality.

Residuals intervals Lastly, the extra nodes are written exploiting consecutivity. So we assume that they present a list of intervals, thus each one of them is written

[1] `http://law.dsi.unimi.it/`

as its left extreme plus its length. This creates again a list of residuals that are coded using differences.

3.1.2 Re-ordering web graphs

Until now, we assumed that web graphs are ordered via lexicographic ordering on nodes URLs, this creates some interesting properties such as similarity and consecutivity but perhaps is not the best ordering to employ. The ordering of the nodes in a crawled graph can have major impact on the resulting compression ratio, indeed, many studies tried to tackle this problem and to find a reordering technique that maximizes compression. The key points are that an ordering should create as much similarity and consecutivity as possible, as those are the two most useful properties for reference compression and intervals encoding.

We start by discussing some preliminary studies made on graph ordering to find an enumeration of nodes that leads to better compression ratios than lexicographic ordering by URL [1].

Gray code and ordering An n-bit Gray code is an arrangement of all the binary vectors of length n, such that any two successive elements differ only by one bit. An example of Gray code for $n = 3$ is the following:

x	$\overline{x}$
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Gray codes can be used in this context to order the adjiacency matrix rows, this creates "slow" changes from row to row, possibly increasing similarity. Permuting

the entire adjiacency matrix is unfeasible and for a web graph is even really sparse, so the idea is to compute gray ordering by iterating trough two nodes successors iterators. The comparator presented in the study is going to be adapted in the proposed methodology, and is thus presented in Algorithm 3 on page 34.

Preliminary studies on ordering Numerous experiments have been made to assess compression quality of different types of ordering, the tested ones are:

- URL ordering: nodes are ordered in lexicographic order by their url;
- Lexicographic row: given the boolean rows of an adjacency matrix, they are sorted lexicographically;
- Gray: gray ordering is performed on the adjacency matrix rows;
- lhbhGray: keep URLs from the same host adjacency, ordering them via Gray code;
- shbhGray: same as lhbhGray but Gray code are computed only on links to the same host.

The second and third orderings are called intrinsic, as they rely only on the graph structure, while the others do not and are called mixed, as they combine intrinsic and host information. Note that on social graphs we are missing host and URL information, thus a clustering algorithm is employed, where two nodes in the same cluster indicates that they have the same imaginary host.

Experiments shows how intrinsic orderings work well on transposed graphs, while failing to beat the mixed ones on normal graphs. The results on social graphs shows some potential but the best results are obtained on graph coming from the web.

Layered label propagation The presented preliminary studies shows how mixed ordering criterions outperform intrinsic ones, thus information about hosts and URLs is crucial to obtain a good compression ratio. Although the URL can be

easily obtained for web graphs, social ones fails to provide such data, hence a clustering algorithms must be used to build imaginary hosts on a graph. Some good performing algorithms are based on label propagation, the idea is simple: each node starts with a different label and, at each round of the algorithm, every node updates its label following some rules. Once no labels are updated the algorithm stops. A classic update rule can be the majority of labels in the neighborhood of a node.

The next study proposes a new ordering procedure that yield better compression ratio for web and social graphs [5]. The algorithm relies on label propagation clustering, in particular *Absolute Pott Model*: the update of a node label is done by maximizing a function that takes into consideration both the cardinality of a label in the neighborhood and the density of that particular cluster. The APM algorithm have an hyperparameter γ that regulates the impact of the cluster density factor in the update rule and is called the resolution parameter.

The idea of *layered label propagation* is to compute different rounds of the APM algorithm with randomly drawn γ, that produces a label assignment for each node. The labels obtained at every round are used to update the final ordering of the nodes, in particular, nodes with the same label are kept close, nodes inside the same cluster are left untouched, and finally, order among clusters is defined by the instrinsic ordering of the labels produced by the algorithm.

This ordering procedure produces a graph permutation that exhibit increased similarity and locality. The combination of LLP and the BV compression schema leads to great compression ratios for both web and social graphs.

3.2 Inverted indexes compression

An inverted index is of paramount importance in a text search system. A simple index consists of a set of posting lists, each of which contains a sequence of $d_{t,i}$, that are the ids of the documents that contain the term t.

One of the basic approaches to index compression is to transform each sequence with gap encoding, applying an integer compression technique such as instanta-

neous codes. For this purpose, variable byte codes creates a great tradeoff between speed and compression ratio, also Golomb codes can be tailored to a specific geometric distribution that best suit a given index. Patched frame of reference is heavily used on documents id and there exist studies on Elias-Fano lists applied to this domain.

One work in particular exploits asymmetric numeral systems on indexes. The idea of the authors is to combine classical methods with entropy coders to potentially obtain better compression ratios [9][10]. They show some interesting ideas and problems to deal with that are at the base of the presented methodology.

3.2.1 ANS and packed codes

The final approach to index compression presented by the authors combines packed codes with Asymmetric numeral systems.

Packed codes A packed code is a technique that exploits any localized consistencies in a sequence. A block size of consecutive values is selected, typically $B = 128$, and represented as B same-width binary values. A selector is considered, stored in the block header, that determines the fixed width.

An example is to define for instance 16 selector values,

$$S = 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 19, 22, 25$$

the block header will contain a 4 bits to store l, and that block will represent values up to $2^{S[l]}$. Note that this technique is really similar to 2.2.2 on page 11, as the latter encodes integers in the exact same way apart from exceptions.

Packed + ANS This technique aims at combining packed and ANS compression. The idea is to keep the selector value and a block size B but this time the selector refers to a specific ANS model, in the sense of a specific set of symbols frequencies.

This means that to encode a block of B integers, statistics about it are com-

puted and an appropriate model is selected. A trivial selector is the maximum m of the block, which refers to a model with m equiprobable symbols. Note that defining a model with equiprobable symbols permits to avoid writing symbols on disk to reconstruct the model, as the only thing needed to rebuild it is the value m and a given precision. The bits allocated for the selector determines the maximum number of allowed models, in the original paper the authors limit themselves to 16 of them.

Once a model is selected, all the integers in the block are encoded and the selector and the list of states are written on disk.

Two-dimensional context An improvement on the previous technique is to use additional information coming from the block as a selector value. The authors refer to this technique as a two-dimensional context. An example can be using both the maximum m and the median k of a block B. This means that an appropriate model would need to have m symbols where the first k ones share a probability of $\frac{1}{2}$. This improves significantly with respect to the one dimensional context, and a lot of more models are allowed.

3.2.2 Results and key takeaways

The presented methodology is really interesting and combines two unrelated compression methods to store the inverted index. The compression ratio is really good with respect to other classical techniques, while the access is on the slow side, but this is acceptable for entropy encoders.

Some key takeaway from this technique is that the number of ANS models must be somehow limited. The solution of the authors is to rely on a selector value, that defines how many models are possible. This both limit the number of models and symbols, as the first one is determined by the bits allocated for the selector, and the second by the context used to refer to a single model.

Limiting the number of encoders and their dimensionality is going to be a key factor in the proposed methodology, as not doing so leads to poor results.

Chapter 4

Proposed methodology

We now discuss the proposed methodology for graph compression. The final compression scheme is obtained through various iterations, that start with a basic approach, followed by an optimization, terminated by the refined final methodology.

The first approach consists in a naïve way of applying ANS to graph compression, presented in Section 4.1 on page 23. The idea is to encode every successors list in the graph with an ad hoc encoder. This means that, given the successors of a node, we compute the frequency of the symbols, where those are the integers appearing in the gap encoding of the list, and we encode them optimally. The result is two separate files, one containing the graph itself, and the other the models definitions.

Numerous problems arise with this approach and are somehow related to the key takeaways presented in 3.2.2 on the preceding page. Recalling the inverted index compression with ANS, the authors tackled two main problems, the first one is the number of different encoders used in the compression, and the other was about limiting the max number of symbols per model. The main drawbacks of the naïve approach are indeed the excessive number of total encoders and the symbols per encoder. If we think about it, we are considering as many compressors as the number of nodes in the graph, that is in the order of the millions, while also allowing all possible gaps to be considered as symbols, that assumes values

ranging from zero to the number of nodes in the graph. This methodology creates an encoding on disk that is heavily dominated by the space taken to dump the models, which is roughly double the size of the encoded successors.

Section 4.2 on page 27 introduces the first step toward the final methodology. The immediate question that arises is how to reduce the space taken by the models. There are numerous ways to do so, but the key point is that although we want to reduce space on the model size, we also want to preserve the goodness of the successors encoding. Some heuristic approaches have been tested but the one that appeared to be the most proficuous consists in adding escaping to the ANS models, where the idea is to avoid including symbols that are rare in the frequency maps. Starting from node successors, we compute the gap encoding and the statistics about the symbols. We order them and cut off a percentage of the final ones, leaving us with a reduced symbol map. The removed gaps are added as a single special symbol, that will be used to encode them in the state. Once an escaped integer is encoded, it is also written in an escaped list, that will be added to the successors encoding on disk. This approach worsens the graph encoding, as we are adding explicit symbols to be written, but reduces the size of the models by quite a lot, improving the total space. Although better, this approach provides a compression ratio that is similar to the non compressed representation and is thus not really useful, the space taken by the models is surely less with respect to the naïve methodology but still quite big.

Section 4.3 on page 31 introduces the final refinement, that focus on limiting the number of models. The first thing that comes to mind is clustering, the goal now is to find a way to merge similar models, so their union can be used to encode more than one node. Initially, we tough about K-means clustering, indeed, there exist some studies about clustering probability distributions that can be exploited in this case. The main difficulty of this method is choosing a distance metric and avoiding an extreme situation in which all the models get merged in a single cluster. Unfortunately, this classic clustering algorithm was not easily applicable to this use case, hence it was discarded for a more simple approach. The simplified task now becomes ordering the models in some way and partitioning them, merging

the single partitions. The ordering should put similar models close to each other, while the partitions need to limit the total number of symbols per split. A great way to order models is by Gray-code ordering. The details are discussed in the relative Section, but the idea is to sort models in a way that subsequent frequency maps differ by at most one symbol. Partitioning is done in a heuristic way, limiting the number of symbols per partition, while keeping the number of clusters low. Finally, the idea of escaping is used on the obtained clustered models, this time in a more sophisticated way. This final refinement reduced the overall space by over 40 percent.

4.1 Initial approach

The initial approach consists of encoding each node with an ad hoc encoder. Although optimal for a single node, this methodology becomes quickly unfeasible for larger graphs. The number of bits needed to write models on disk penalize the goodness of the optimal encoding, due to the fact that we are writing both too many models and symbols.

Performance vary, but this approach often creates an encoding that exceeds the space taken by quasi-succinct representation, making it not really useful. Although bad this initial implementation gives an idea about the issues that needs to be resolved in later phases.

4.1.1 Computing successors statistics

As stated in Section 2.2.1 on page 8, the idea of an asymmetric numeral system is to encode symbols taking into consideration their probabilities. Hence we need to define what are those symbols in our case and what probabilities we are considering.

The first step to encode the outgoing links of a node is to compute statistics about the gaps. Let *succ* be the array containing the sorted successors of the i-th

node, we compute the array *gaps*, with the following rule:

$$gaps[j] = \begin{cases} succ[j] & \text{if } j = 0 \\ succ[j] - succ[j-1], & \text{otherwise} \end{cases}$$

The symbols coded with ANS are going to be the gaps created by this preprocessing phase, so for each distinct value appearing in *gaps*, we compute its frequency. We then scale these counts to a given range 2^d, obtaining a quantity f_s for each $s \in S$, thus the probability of s can be written as $p_s = \frac{f_s}{2^d}$.

The magnitude of the gaps ranges from zero to the number of nodes in the graph, but, considering structural properties of web and re-ordered social graphs, consecutivity in particular, the gaps distribution should be concentrated in small values. This is the motivation for the approach, as without this property we would need to hope for a concentration in some values, that might not happen.

4.1.2 ANS based compression

After computing the frequencies of the gaps of a given node, we encode them with ANS compression.

We define an ANS Model, as the collection of the required data structures to encode a symbols source. This model contains the state storing the encoded symbols, the frequencies defined in the last section, so an array storing normalized counts and the two mappings, as well as the cumulatives.

Preventing overflows The model state is represented with a 64 bits long integer. Recall that the encode primitive takes the older state and starts by multiplying it by $M = 2^d$, and this could cause overflows. To prevent this from happening we perform a 128-bit multiplication between the state and M, checking that the upper 64 bits are all zeros. When inevitably an overflow happens, the current state is added at the end of an array, resetting it to zero.

Sym primitive The goal of the cumulative if to define this primitive, $sym(n) = \max_{c_s \leq n} S$. While one could store an array of length 2^d and fill each spot with the corresponding cumulative, this task can be solved with an Elias-Fano indexed monotone list using less memory. This implementation supports predecessors queries of the type, given an element x, find the maximum element less than x, that are exactly what you need to find. This is not only really fast but also uses optimal memory. Another approach would be to perform a binary search on the sorted cumulatives, having logarithmic complexity for the *sym* primitive, or to store an integer array of length 2^d, containing the associated symbol with each cell.

Reverse order Given the gaps of the successors of a node, they are encoded in reverse order to avoid inverting the decoded sequence. Once the process is completed, the state sequence and the required structures to rebuild the model are written to disk.

Algorithm 1 Encoding procedure. The fifth line is the one responsible for overflows, when one happens, the state must be added to a list and reset to zero.

1: $state \leftarrow 0$
2: **for** $s \in succ$ **do**
3: $\quad j \leftarrow state \text{ div } f_s$
4: $\quad r \leftarrow state \bmod f_s$
5: $\quad state \leftarrow j \cdot M + c_s + r$
6: **end for**
7: **return** *state*

Algorithm 2 Decoding procedure. In reality the state is not a single integer, thus when the current state reaches zero, the next one must be loaded.

1: $list \leftarrow [\,]$
2: **while** $state > 0$ **do**
3: $\quad r \leftarrow 1 + (state - 1) \bmod M$
4: $\quad j \leftarrow (state - r) \text{ div } M$
5: $\quad state \leftarrow j \cdot f_s - c_s + r$
6: $\quad list \leftarrow list + sym(r)$
7: **end while**
8: **return** *list*

Model coding on disk Once we encode the successors of a node, we need to write all the required information to decode them on disk. This translates to dumping all the required structures on two separate bit streams.

We start with the list of states produced by the encoding, those are saved on

the *.graph* stream, where each node stores, all in γ, its out-degree, the id of the corresponding model, the number of states, followed by the list of states written in binary fixed at 64 bits. The id of the model could be omitted here, as every node has its separate encoder, this is not true in the upcoming Sections. The model structures are then written on the *.model* file. We start with the size of the list of symbols, where each one is written and ordered by inverse mapping index. After that, the sorted descending frequencies are written backward by gap. All the integers are written in γ. δ code is also an option, but γ works empirically better in this case. To rebuild the symbols mapping, we just need to read the symbols, associating an incrementing index to each one of them. Frequencies must be filled in inverse order, and the cumulatives are obtained at run time.

Node	Outd.	Model id	N. states	States
...	...	...	...	...
14	5	14	1	1873215
15	2	15	1	4732153
16	130	16	16	1237953, 543843, ...
...	...	...	...	...

Table 4.1: Successors encoding on the *.graph* file

Model id	N. symbols	Inv. sorted syms	Gap sorted freq
...	...	...	...
14	3	1, 5, 3	211, 300, 513
15	1	1	1024
16	6	1, 2, 4, 9, 6, 1234	32, 84, 12, 4, 150, 742
...	...	...	...

Table 4.2: Ans models encoding on *.model* file, notice how the frequencies sums up to 1024, that is the assumed precision.

Performance As said before, the information is split in two files, one containing encoded successors and the other information about the model structures. The performance in terms of occupied space is optimal for the first one, but tremendous

for the second, as the required structures contain a lot of different symbols. The idea of the next Section is to tackle this problem.

Graph	BV	EF	Naïve ANS
eu-2005	3.726	17.504	13.505
uk-2014-tpd	10.117	18.482	29.394
eu-2015-tpd	4.492	14.867	12.223
en-wiki-2013	13.114	19.500	28.838
en-wiki-2022	13.518	19.923	29.162

Table 4.3: Bits per link obtained by the initial ANS compression method, compared to BV graph and Elias-Fano lists.

4.1.3 Approach problems

The main drawback of this methodology is the excessive number of models that we are considering. Not only the space occupied on disk is completely dominated by the models encoding, but also, we can not efficiently load the graph in memory, as one would need to load a decoder first and then load the node successors, the ideal situation would be to have a set of loaded models to use when necessary.

The immediate goal now becomes finding a way to limit the space taken by the structures both on disk and in memory, and for an initial refinement, discussed in the next Section, we think about escaping.

4.2 Escaped ANS

This Section focuses on the problem of limiting the space taken by each node model. Defining an encoder the perfect way fails for the mentioned problems about the excessive space of the .model file, hence we need to find a way to describe a model in fewer bits, or to make each one of them sub-optimal, by gaining on the disk representation.

One of the first approaches to the problem was to build encoders that are defined approximately, based on the median, third quartile, and maximum of the symbols to encode. Given these statistical measures, all the symbols under the

median would share a probability of 0.5, the ones between the median and third quartile 0.25, and so on. All the symbol's frequencies and cumulatives can be computed at run-time. This would permit to reduce the model encoding to just three integers, basically nothing with respect to the successors file. Although appealing this method fails to store successors efficiently, as the space taken by those increases by quite a lot, making the method worse than the naïve one. The main issue is that although the assumption that smaller gaps appears more often is true, a model is now considering a number of symbols that corresponds to the maximum observed gap in the successors list. This means that despite the incredibly compact representation of an encoder, it possibly contains a huge amount of symbols, making each state updates waste many bits. An example can be a gap encoded list of:

$$4189043, 1, 1, 1, 2, 3, 1, 874$$

In this case, we are assuming that the symbols from zero to 4189043 are present in the model, making the encoding of the rarer ones, namely 874 and 4189043, cost a lot, as they have probability of $\frac{0.25}{4188169} \approx 6 * e^{-8}$ each.

Given the gap encoding of a successors list, the big gaps are basically random, so perhaps they can be removed altogether and encoded in a separate way. This is exactly the idea at the base of this Section.

4.2.1 Escaping gaps

One of the scenarios in which ANS compression works well is when the overall probability of the alphabet is concentrated on a few symbols, and also the cardinality of the symbols set is not too big. As seen until now, the symbols and their frequencies must be written somewhere to rebuild the decoder, thus an excessive amount of them makes the really good performance of the ANS encoding useless.

We can build on top of this idea and escape some symbols to encode them in

a different way. Given a list of successors gaps as follows:

$$10, 1, 1, 1, 2, 1, 2, 123, 256, 312, 3, 3, 2, 1$$

the frequency map looks like this:

Symbol	Frequency
1	5
2	3
3	2
10	1
123	1
256	1
312	1

We can see some symbols appearing only once, hence we could avoid writing these ones to the .model file and encode them with an instantaneous code or a different compression method. The idea is to aggregate the rarer symbols into a single one, for instance, discarding only the hapax the map becomes:

Symbol	Frequency
1	5
2	3
3	2
*	4

Non-rare symbols are encoded as always, an escaped one is encoded as a $*$, and written explicitly in an auxiliary list in memory. This list is then going to be written on the .graph file with the states.

4.2.2 Choosing what to escape

In the example above, the symbols appearing once are discarded, in practice, a percentage p is selected. The symbols are sorted in descending order by frequency, in case of a tie, the smaller one comes first. Then, the last $\frac{n}{100} * p$ symbols are

escaped. The reason behind this sorting is that of course we want to escape rarer symbols, thus the descending frequency, but, if we had to choose, we prefer smaller symbols, as they take less space in the ANS model maps.

PFOR delta escape encoding The escaped symbols must be encoded in some explicit way on disk. The chosen approach is similar to PFOR delta but slightly modified. Given v as the list of integers to encode, we find b_{max}, the binary magnitude of the maximum, and we consider b_i, the required bits to store the i-th symbol.

Fixed a b_i, if a number can be written in that number of bits, a zero flag is written, followed by b_i bits representing the integer. If the number overflows the fixed number of bits, a one flag is written followed by b_{max} bits to represent it. For instance, given $b_i = 5$:

$$27 \rightarrow \mathit{flag} = 0,\ \mathit{bits} = 11011$$
$$327 \rightarrow \mathit{flag} = 1,\ \mathit{bits} = 101000111$$

Fixed a b_i, the space for the whole list can be easily computed, and the minimum is selected. The resulting coding on disk is b_i, followed by $b_{max} - b_i$ and the length of the list n, all three written in γ code. The remaining are n integers written as explained above. For example, given the escaped list

$$2103, 43829, 658, 89574, 6431$$

the final encoding on disk looks like this:

$$13, 4, 5|\langle 0, 2103\rangle, \langle 1, 43829\rangle, \langle 0, 658\rangle, \langle 1, 89574\rangle, \langle 0, 6431\rangle$$

Performance This approach cuts the number of symbols to be written by a lot, reducing the overall space taken by the graph and model files. The bits required to store only successors increase, as some values might be escaped and written separately, Table 4.4 on the next page shows compression ratios on example graphs. A positive result about this approach is that bigger graphs can now be

loaded successfully, as the smaller models permits to load a lot more node encoders, the issue is still present on bigger graphs.

Graph	BV	EF	Basic ANS	Escaped ANS
eu-2005	3.726	17.504	13.505	12.936
uk-2014-tpd	10.117	18.482	29.394	27.234
eu-2015-tpd	4.492	14.867	12.223	12.223
en-wiki-2013	13.114	19.500	28.838	28.838
en-wiki-2022	13.518	19.923	29.162	29.162

Table 4.4: Bits per link obtained by the Escaped ANS compression method, compared to previous results.

4.3 Clustered ANS

After reducing the number of symbols per model, the goal is to avoid having a different model for each node. The motivation is that although escaping the models permits to load bigger graphs and save overall space on disk, the occupied model space is still quite big, leading to overall poor compression performance.

The question now becomes, how can we merge models together and what criterion should one choose to do so. The first task is quite simple, but the latter requires a bit more work. We are now putting aside the escaping procedure presented in the previous Section and consider only the full models. The escape idea is reintroduced in a more sophisticated way after the models reduction.

Merging two models The first step to clustering is defining how we can merge two models together. Recall that the ad hoc encoders symbols are observed from a node successors, later, counts get normalized to a given range 2^d to estimate probability.

The union of two or more models is computed by merging the frequency maps before normalization, for instance, given two models M_1 and M_2, the frequency map of the union U is computed as follows:

Symbol	M_1	M_2	U
1	10	6	16
2	7	8	15
5	-	1	1
14	-	1	1
38	1	-	1

that corresponds to treating the gaps coming from node 1 and 2 as a single long list. The union map is then normalized as always.

K-means clustering for distributions When thinking about clustering, a classic algorithm that comes to mind is K-means, that requires a distance metric and operates iteratively. The idea is to assign each point to the closest centroid, recomputing them accordingly at each iteration. This method could work in our case, as the datapoints are the ANS models, and a suitable distance metric could be the Kullback-Leibler divergence between probability distributions, computed by averaging its quantity in both directions. The centroid can then be recomputed by summing the observed symbol frequencies for each model and normalizing them, just as discussed in the above paragraph.

Unfortunately this method fails to perform well for this problem. The main drawback is that the process is not really reliable, leading to the unfortunate situation in which more and more points gets added to a single big cluster. In the end, a simpler heuristic approach is selected, removing the need to perform a classical clustering algorithm.

4.3.1 Ordering and partitioning

After defining how two models can be merged, the problem of finding suitable subsets of models to cluster remains open, as the application of K-means clustering fails to perform well for the given task.

Computing all possible subsets would obviously be unfeasible, thus an heuristic approach is employed. The idea is, starting from all the models, to order them

and to find contiguous partitions to merge. This reduces the problem to choosing a suitable ordering for the models and a partition criteria.

The ordering criterion should closely place similar models, while the partition logic must find a tradeoff between the number of symbols per model and the number of models themselves. The first problem is solved with Gray-code ordering, while an heuristic split procedure is developed to tackle partitions.

Gray-code ordering Let k_i be a sorted array of symbols in the i-th model, if we fix an upper bound n, that array can be seen as a binary vector on the possible symbols. For example, given a model with some symbols, $3, 1, 8, 5$, and a limit of 10, we sort the array and imagine a bitmask like this:

$$[3, 1, 8, 5] \longrightarrow [1, 3, 5, 8] \longrightarrow 01010100100$$

If the goal is to minimize the difference in symbols between adjacent models, a Gray ordering is what we need. As we saw in 3.1.2 on page 16, Gray codes find another application in web graphs, in particular they can be permuted via Gray ordering on the adjacency matrix rows, the idea is to have slow changes in successors when going to the next row. The intuition here is the same, we want slow changes in the model symbols when we go from one model to the following. The mentioned study also proposes an interesting comparator that makes it possible to order two adjacency lists by Gray code, without having a complete instance of the row. We can use the exact same comparator to order the models by symbols, its definition can be seen on Algorithm 3 on the next page.

Partitioning the sorted models After sorting the models accordingly, one must find a way to determine how to merge them. The idea of the approach is to specify an upper limit to the number of partitions and let the algorithm do its work, ideally, we want models with a limited and somehow equal number of symbols. Starting from an initial max symbols quantity, that it set to the average number of symbols in the un-partitioned models, we try to build partitions not exceeding it, when this inevitably happens, a new partition is created.

Algorithm 3 Gray code comparator for two ANS Models. i and j denote two iterators, returning the model keys in sorted order, where ∞ means that the end is reached. The expression $[a < b]$ has value 1 if a is smaller than b, zero otherwise. The return value of the comparator is less, equal or greater than zero if the j-th model precedes, is equal or follows i-th model.

```
1: p ← false
2: while true do
3:     a ← next(i)
4:     b ← next(j)
5:     if a = ∞ and b = ∞ then
6:         return 0
7:     end if
8:     if a ≠ b then
9:         if p ⊕ [a < b] then
10:            return 1
11:        else
12:            return −1
13:        end if
14:    end if
15: end while
```

If the result exceeds the limit of partitions set at the beginning, a new round is performed doubling the quantity of the max symbols. This method repeats until the partition limit is met. The process is quite fast as the number of allowed symbols doubles at each iteration, hence the number of partitions is typically cut in half or more. The result is partitions that have a limited number of symbols and are similar in terms of frequency maps cardinality. Algorithm 4 illustrates the procedure.

Algorithm 4 Partitioning procedure after Gray code ordering.

```
1: maxSymbols ← avergage # symbols per model
2: limit ← n * partitionsPercentage
3: splits ← n
4: assignments ← [−1, . . . , −1]
5: while splits > limit do
6:     i ← 0
7:     symCounter ← ∅
8:     for model ∈ models do
9:         if |symCounter| > maxSymbols then
10:            symCounter ← ∅
11:            i ← i + 1
12:        end if
13:        symCounter ← symCounter ∪ model.symbols
14:        assignments[model] = i
15:    end for
16:    splits ← i + 1
17:    maxSymbols ← maxSymbols * 2
18: end while
19: return assignments
```

4.3.2 Heuristic escape

After the partitioning procedure, each split is condensed in a single model, merging the frequency maps as introduced at the beginning of the Section. The merging procedure should create models that are related to each other, as they are ordered via Gray-code, but it is safe to assume that the cardinality of those frequency maps

are way bigger than a single optimal model for a node. To mitigate this behavior, escaping can be applied to the merged models, this time in a more sophisticated way.

Escaping the aggregated models To reduce the number of symbols in the merged models we apply escaping before initializing the ANS structures, by minimizing a function. All the frequencies are now considered at the cluster level. The formula to minimize is:

$$S = E_{ans} + E_{esc} + \textit{escape bits} + \textit{model bits}$$

The first component is the entropy of the ANS encoding:

$$E_{ans} = -\sum_{s \in S} c_s \log p_s$$

where S is the alphabet, c_s is the count of a symbol, p_s is defined as the frequency of the symbol over the total count of frequencies. This part of the formula models the number of bits required by ANS states.

The second part is the entropy of the escapes, let S' be the set of escaped symbols, $p_{esc} = \sum_{s \in S'} p_s$ is the sum of the probabilities, $c_{esc} = \sum_{s \in S'} c_s$ the sum of counts:

$$E_{esc} = -c_{esc} \log p_{esc}$$

This factor model accounts for the bits required to encode the escape symbol into the ANS state. The sum of this and the previous term results in the total bits spent by the ANS encoding.

The third element is the cost of writing escapes on disk, that can be approximated with:

$$\textit{escape bits} = \alpha(c_{esc} * b_{max})$$

where b_{max} is the binary magnitude of the maximum escape, α is the percentage of space gained by using the PFOR-like approach instead of b_{max} bits for each element. It can be measured empirically, a good value can be 0.9.

The final term of the formula is the space required for the ANS models, that is all the symbols written in γ code, plus the sorted frequencies by gap, this term is referred to as *model bits*.

An important note is that by minimizing this formula, we are considering the symbols statistics at the cluster level, that means, the aggregated symbol counts coming from the nodes. To make a more accurate estimation, one would need to fix a subset S' of escaped symbols, iterate over all the points in the cluster and compute the space required by each node. This would cost too much in terms of efficiency.

Choosing the escape symbols set The escape symbols set S' is the core of the space formula to minimize. The approach does not consider all possible subsets of S, indeed, given the symbols sorted by descending frequency, we fix a cut point and take all the elements after that point as the escape set.

For instance, considering the symbols ordered by frequency:

$$syms = 1, 2, 5, 3, 13, 110, 45$$

The cut point starts at the end of the symbols, so $S' = \emptyset$, then with each iteration, the last symbol of the list is added, so the set progressively becomes $\{45\}$, $\{45, 110\}$, $\{45, 110, 13\}$ and so on.

This permits to minimize the formula in $O(|S|)$, as all the components can be updated in constant time from one iteration to the other. Figure 4.1 on the next page shows a comparison between the heuristic minimum with respect to the actually occupied space.

Escaping before partitioning The idea of escaping a single node model introduced in the previous section can be used before partitioning. Indeed, the escape wildcard is treated like any symbol in the alphabet during the algorithm. When choosing the S' set, a prior escape translates to starting with a non-empty set. This reduces the execution time of the algorithm by quite a lot. About the sorting criteria when choosing what to escape in the single nodes, preferring smaller sym-

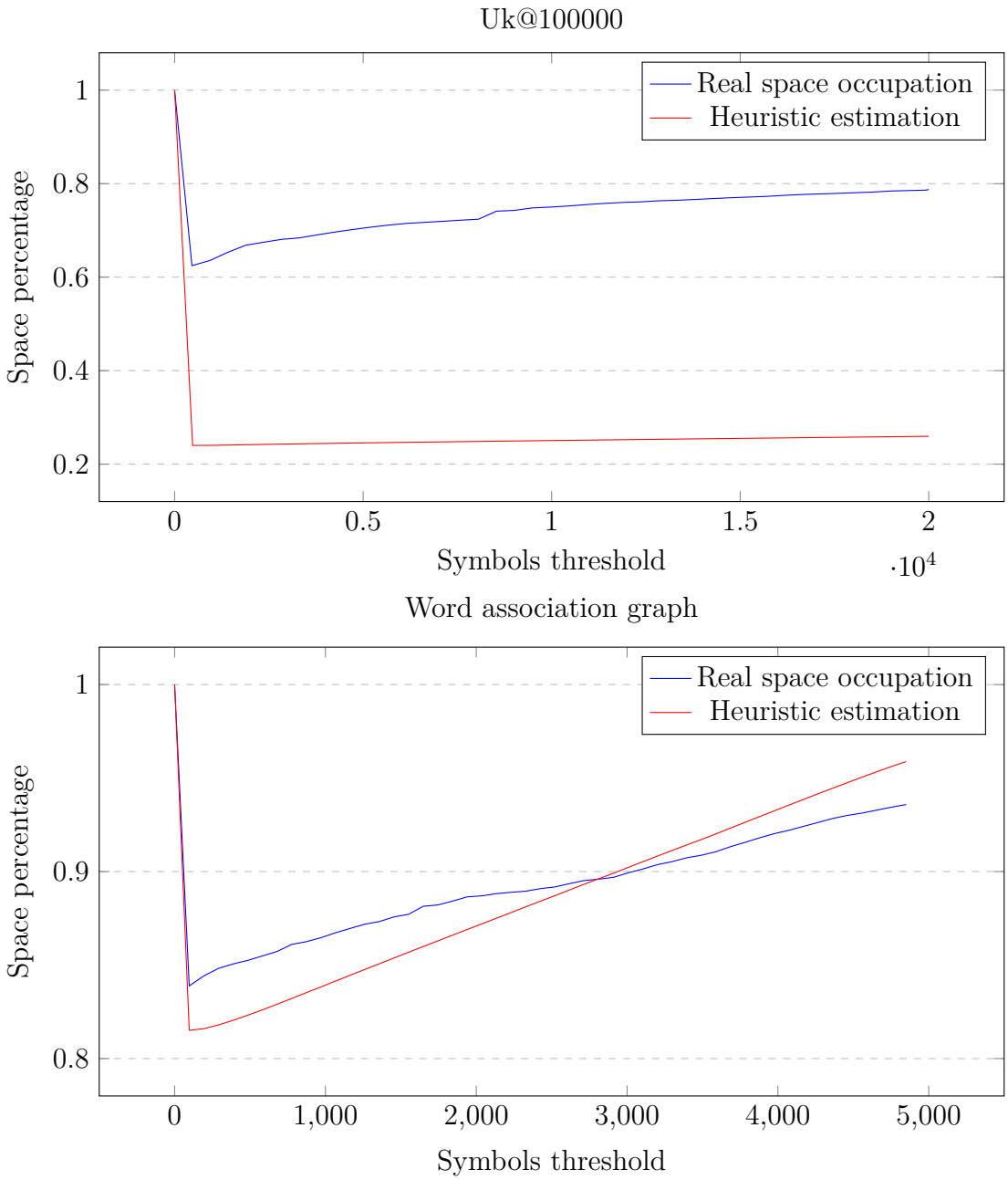

Figure 4.1: The x axis represent the threshold for the escape symbols set, for instance $x = 100$ means that the first 100 symbols are considered in the ANS model, while the remaining $S - 100$ are escaped. The blue line is the real space occupied by the graph with that escape set, while the red one show the heuristic function. Both function values are normalized, to have a better visualization. The two minimums for both graphs are quite close.

Graph	BV	EF	Basic ANS	Clustered ANS
eu-2005	3.726	17.504	13.505	7.821
uk-2014-tpd	10.117	18.482	29.394	17.855
eu-2015-tpd	4.492	14.867	12.223	7.723
eu-2015-tpd-t	5.158	15.678	14.682	8.877
en-wiki-2013	13.114	19.500	28.838	16.771
en-wiki-2022	13.518	19.923	29.162	16.779
en-wiki-2022-t	13.518	19.923	29.162	14.891

Table 4.5: Bits per link obtained by clustered ANS with respect to previous results.

bols is more useful for the clustering procedure, as they are more likely to appear in a wider range of nodes. Lastly, escaping before clustering does not lead to the same results as clustering alone, sometimes it worsens the performance, however, it speeds up the method by a lot.

Performance This approach limits the excessive size of the *.model* file, while increasing the space taken by the *.graph* stream. On average, clustering saves 40 percent of space compared to the initial approach, the results can be seen in Table 4.5.

4.4 Implementation details

To have a clearer idea about the implementation of the proposed methodology, we here present the details about the procedures followed when storing and loading a graph on and from disk. It follows an explanation on how successors of a node are retrieved when a request to them is made. The proposed methodology implementation is designed to work seamlessly alongside Webgraph[1], to have an easy time loading and storing graphs, indeed the package defines an ImmutableGraph interface that is extended by our AnsGraph class. The fastutil[2] and dsiutils[3] Java packages are also heavily used, to have more space and time efficient data

[1] `https://github.com/vigna/webgraph`
[2] `https://github.com/vigna/fastutil`
[3] `https://github.com/vigna/dsiutils`

structures, and some useful utilities functions. All the implementation is publicly available on Github[4].

4.4.1 Storing and loading graphs

By developing the code in extension of the Webgraph framework, the publicly available compressed graphs can be used to make experiments, in particular, by loading an existing one, we can visit it and re-store it with the proposed methodology to assess its goodness.

Re-encoding an existing graph To store a graph with ANS, we simply need access to every node successors list. To do so, we can load an existing compressed representation with BVGraph, that is the class defined by the Webgraph framework. So, to apply for instance the basic initial ANS approach one would need to iterate over all the nodes, get the successors, compute statistics and encode them via ANS. Clustering is a different story, as one need the observed frequency maps in memory to order and partition them.

When the required computation is done, the successors and the models are stored on disk on two separate files, plus an additional one that contains graph properties. This file includes useful information to rebuild the graph, such as the endianness of the integer coding, the number of ANS models, the number of nodes, arcs, and so on. This is also where statistics are stored, such as bits per link, maximum number of states, and many more.

Loading from disk Once a graph is encoded on disk, it can be loaded and visited by decoding the successors of nodes. Given a graph, the property file is used to determine the endianness of the encoding and the number of total models m. Once this is known, m models are allocated in memory loading them from the *.model* file.

[4] `https://github.com/tomfran/ANS-Graph-compression`

4.4.2 Successors retrieval

The most useful thing to do with a loaded graph is to request the successors of a node. This is straightforward and makes use of the pre-loaded models. Recall that in the .graph file, each node contains the model id, outdegree, and list of successors states followed by the escaped ones, thus to load a node successors a copy of the corresponding loaded model is obtained, then an ANS decoder is build loading all the states and escapes to memory. Successors are then retrieved performing decode primitives on the states, retrieving escaped symbols when the special character is encountered.

To make the model copy faster, we cache the last required model in memory, this increases performances and avoid unnecessary copies.

Chapter 5

Experimental results

The experimental results are now presented, focusing on the compression ratio and the speed of various graph algorithms and successors access. The implementation is publicly available at Github and is designed to work seamlessly with the Webgraph framework. This permits us to load and re-store the compressed representation of large scale graphs with out proposed methodology. The datasets used in the experiments comes from the WebGraph archive, and are presented in Table 5.1 on the next page. We find a mixture of social and web graphs of different dimensions.

5.1 Compression

The compression results on all tested graphs can be seen in the tables and plots in the next pages. We can see how the proposed methodology always fall between the succinct representation and the WebGraph framework. Overall it saves 52.07 percent of space compared to the non-compressed representation. The best results are obtained on web graphs, as the average space taken by ANS is 29.19 percent of the succinct bound. Social graphs are a different story, ANS uses on average 74.96 percent compared to Elias-Fano lists.

The main differences between storing these type of graphs are in the number of escaped edges. Indeed, web graphs presents really high consecutivity, this means that higher gaps are not really present in the node successors gap encoding, leading

to lower escaping on the ANS models. Social graphs shows less consecutivity, thus higher gaps are more frequent. This means that the heuristic escaping procedure prefers to remove more symbols, as the space taken by the models would otherwise be too much. More escaping means more overhead on the .graph file, hence performance on social graphs is worse.

Graph	Nodes	Arcs
amazon-2008	735 323	5 158 388
arabic-2005	22 744 080	639 999 458
cnr-2000	325 557	3 216 152
dblp-2011	986 324	6 707 236
enwiki-2013	4 206 785	101 355 853
enwiki-2020	6 047 510	142 691 609
enwiki-2021	6 261 502	150 124 927
enwiki-2022	6 492 490	159 047 205
eu-2005	862 664	19 235 140
eu-2015-host	11 264 052	386 915 963
hollywood-2011	2 180 759	228 985 632
imdb-2021	2 996 317	10 739 291
in-2004	1 382 908	16 917 053
indochina-2004	7 414 866	194 109 311
uk-2002	18 520 486	298 113 762
uk-2007-05@100000	100 000	3 050 615
uk-2007-05@1000000	1 000 000	41 247 159

Table 5.1: Graphs used in the experiments.

Graph	BV		ANS		EF
amazon-2008	9.251	(42.85%)	17.735	(82.14%)	21.591
arabic-2005	1.841	(8.68%)	5.149	(24.29%)	21.201
cnr-2000	2.897	(16.24%)	9.152	(51.31%)	17.838
dblp-2011	8.716	(40.78%)	20.541	(96.10%)	21.374
enwiki-2013	13.114	(67.25%)	16.514	(84.69%)	19.5
enwiki-2020	13.506	(67.83%)	16.872	(84.74%)	19.911
enwiki-2021	13.504	(67.85%)	16.809	(84.45%)	19.904
enwiki-2022	13.518	(67.85%)	16.782	(84.23%)	19.923
eu-2005	3.726	(21.29%)	10.041	(57.36%)	17.504
eu-2015-host	3.273	(20.80%)	7.066	(44.91%)	15.734
hollywood-2011	5.113	(34.92%)	7.719	(52.72%)	14.642
imdb-2021	8.763	(34.18%)	19.6	(76.45%)	25.639
in-2004	2.172	(11.44%)	7.593	(40.00%)	18.981
indochina-2004	1.457	(7.89%)	4.371	(23.67%)	18.469
uk-2002	2.271	(9.98%)	7.314	(32.14%)	22.757
uk-2007-05@100000	2.033	(14.84%)	4.736	(34.57%)	13.7
uk-2007-05@1000000	1.68	(10.19%)	3.939	(23.90%)	16.479

Table 5.2: Bits per link obtained on the natural version of the used graphs. The quantity in parenthesis indicates the space occupied with respect to the quasi-succinct bound set by EF.

Graph	BV		ANS		EF
arabic-2005	1.251	(7.05%)	5.433	(30.62%)	17.744
cnr-2000	2.343	(16.15%)	10.808	(74.52%)	14.504
enwiki-2013	10.954	(64.26%)	14.356	(84.22%)	17.046
enwiki-2020	11.635	(65.44%)	15.112	(84.99%)	17.781
enwiki-2021	11.697	(65.76%)	15.135	(85.09%)	17.788
enwiki-2022	11.684	(65.51%)	15.069	(84.49%)	17.836
eu-2005	2.858	(19.50%)	7.521	(51.32%)	14.656
eu-2015-host	3.578	(22.87%)	7.764	(49.63%)	15.644
in-2004	1.813	(10.83%)	8.677	(51.83%)	16.741
indochina-2004	1.063	(6.69%)	4.923	(30.98%)	15.893
uk-2002	1.703	(8.50%)	8.31	(41.46%)	20.045
uk-2007-05@100000	1.431	(14.49%)	4.51	(45.67%)	9.876
uk-2007-05@1000000	1.046	(9.20%)	3.449	(30.34%)	11.369

Table 5.3: Bits per link obtained on the transposed version of the used graphs. Note that undirected graphs are removed from this table, such as amazon-2008, imdb-2021 and hollywood-2011.

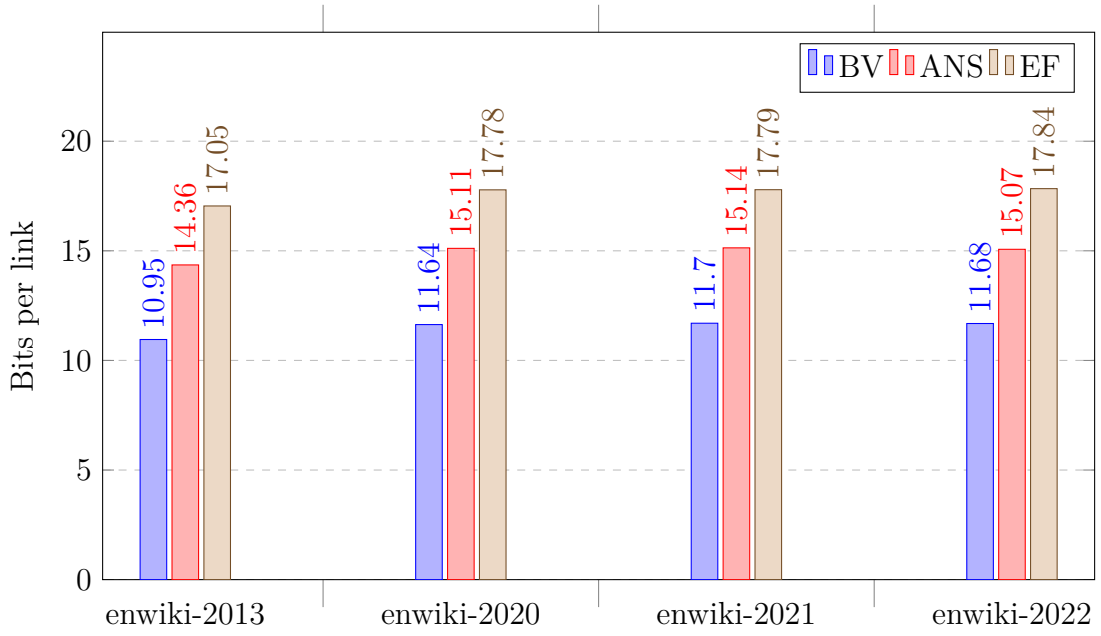

Figure 5.1: Compression ratio for normal wikipedia graphs.

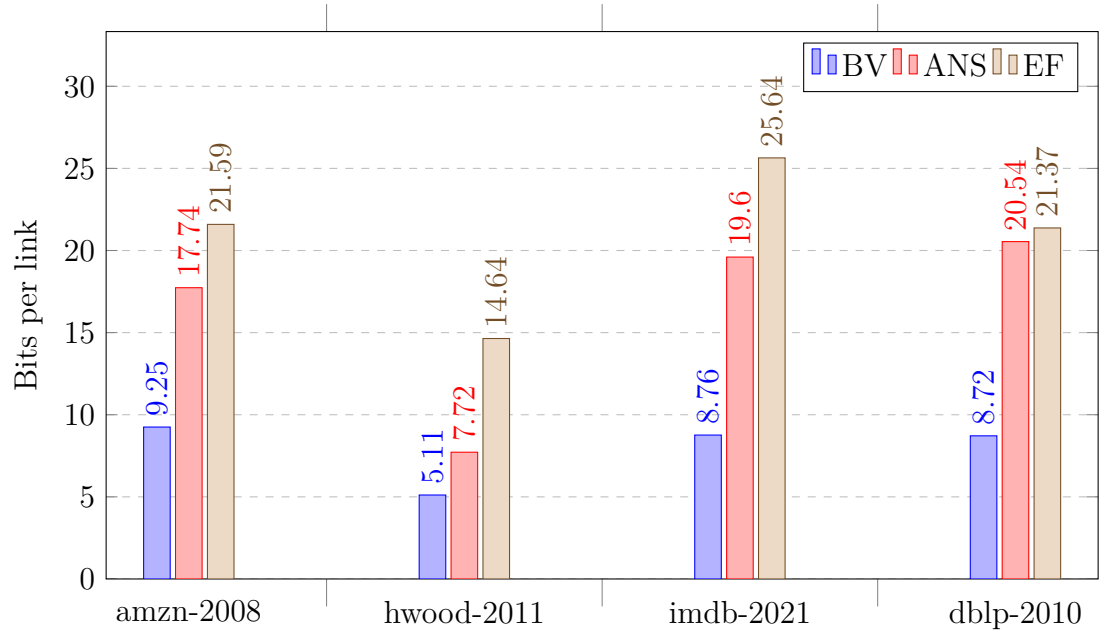

Figure 5.2: Compression ratio for other social graphs

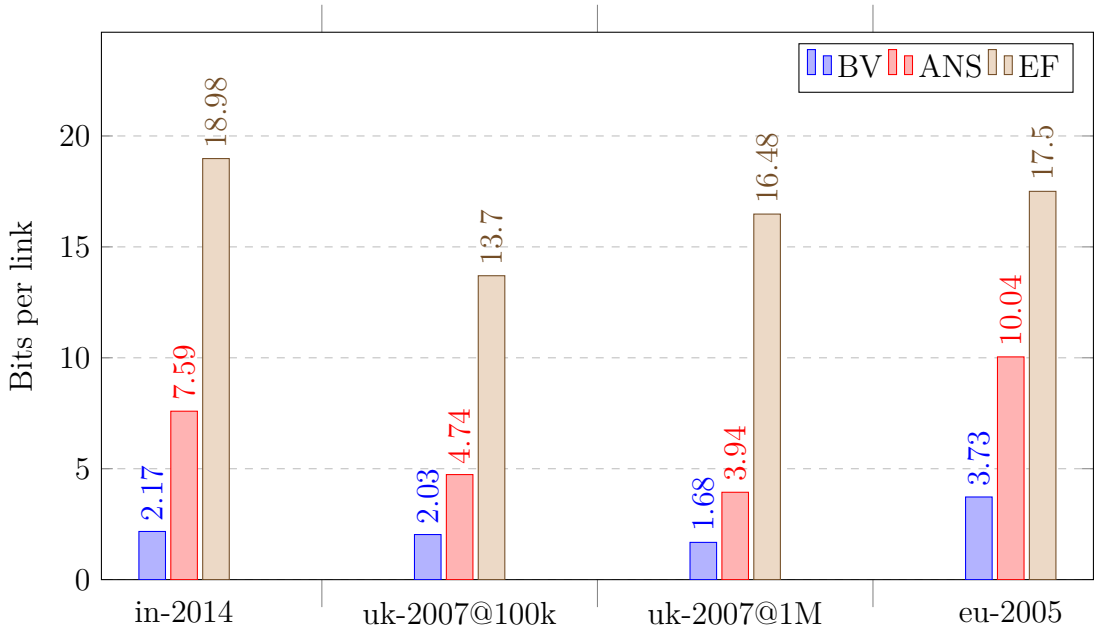

Figure 5.3: Compression ratio for normal smaller web graphs

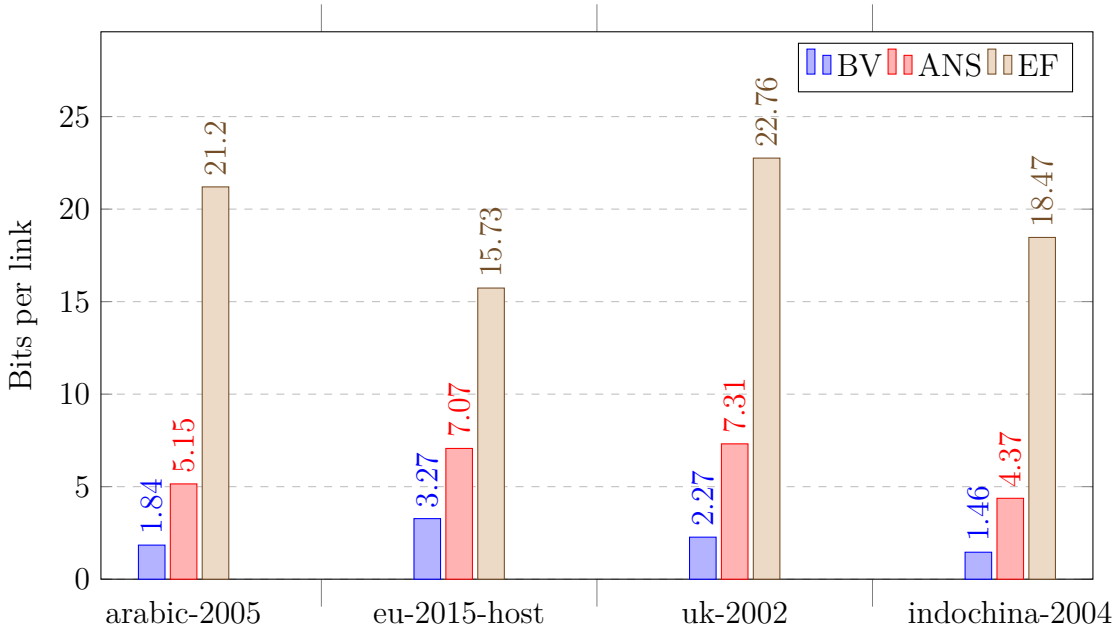

Figure 5.4: Compression ratio for normal larger web graphs

5.2 Speed

To evaluate the speed of the proposed methodology a test is performed for random accesses to nodes successors. We fix a number warmup and measurements rounds, the first ones are to prepare the JVM, and the last ones are where the measurements are made. At each step, a random node is extracted and its successors are visited, this permits to find the time it takes to visit a single node and a single link.

The tests are performed on four graphs, that are uk-2007-05@1000000, indochina-2004, enwiki-2013 and dblp-2011 respectively. The number of warmup iterations is set to one thousand, while five thousand iterations are performed as measurement ones. Each iteration randomly draw five thousands nodes. The measured speed is in terms of nodes and arcs per second, and nanosecond per node and link.

The results depends on how much escaping is performed on the proposed methodology: visiting the web graphs is quite fast, they indeed have little to no escape, at only three percent of the total arcs. Visiting social graphs is slower, the escape percentage for those two graphs is more or less 48 and that many arcs needs to be handled twice, firstly for the state decoding and secondly to read them from the escape symbols list. Overall performance is promising as it is close to BV, EF is way faster than both methods, but no compression takes place there.

Method	nodes/s	arcs/s	ns/node	ns/link
BV	600,083.431	24,559,134.514	1,666.434	40.718
ANS	529,725.126	21,679,636.460	1,887.771	46.126
EF	3 641,793.041	149,044,750.344	274.590	6.709

Table 5.4: Speed results on uk-2007-05@1000000.

Method	nodes/s	arcs/s	ns/node	ns/link
BV	845,512.741	24,223,770.915	1,182.714	41.282
ANS	604,860.320	17,329,127.197	1,653.274	57.706
EF	3,177,472.452	91,033,950.267	314.715	10.985

Table 5.5: Speed results on indochina-2004.

Method	nodes/s	arcs/s	ns/node	ns/link
BV	554,698.925	12,921,822.402	1,802.779	77.388
ANS	482,158.298	11,231,973.980	2,074.007	89.032
EF	4,160,763.520	96,925,818.342	240.340	10.317

Table 5.6: Speed results on enwiki-2013.

Method	nodes/s	arcs/s	ns/node	ns/link
BV	1,558,100.343	10,649,615.842	641.807	93.900
ANS	1,890,557.931	12,921,963.455	528.944	77.388
EF	5,247,060.435	35,863,658.071	190.582	27.883

Table 5.7: Speed results on dblp-2011.

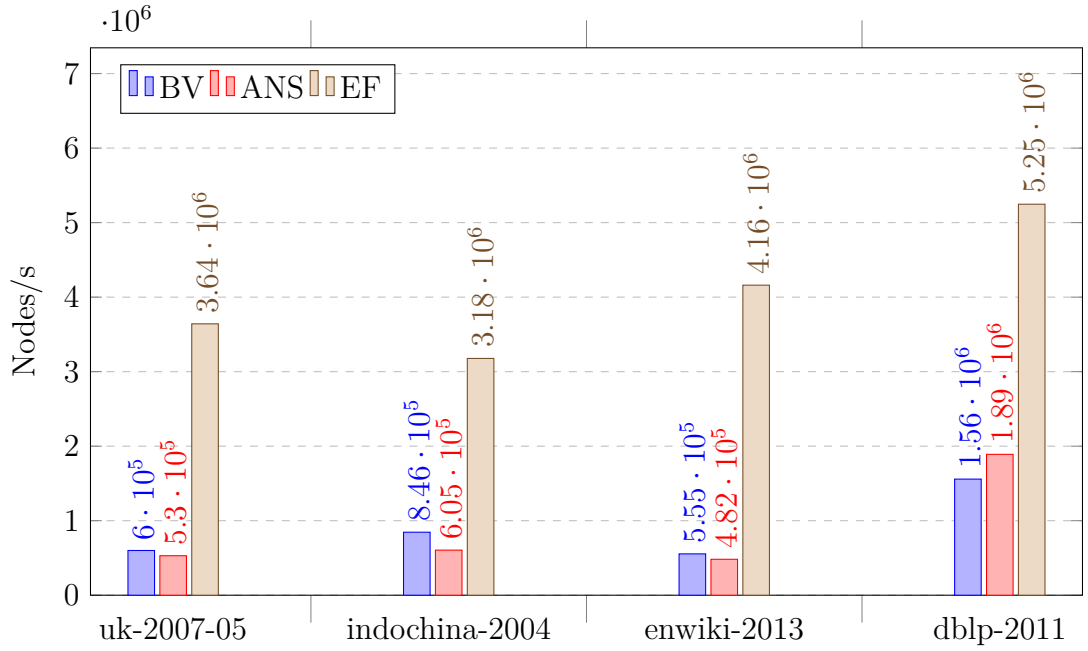

Figure 5.5: Nodes/s over the four tested graphs, the higher the better.

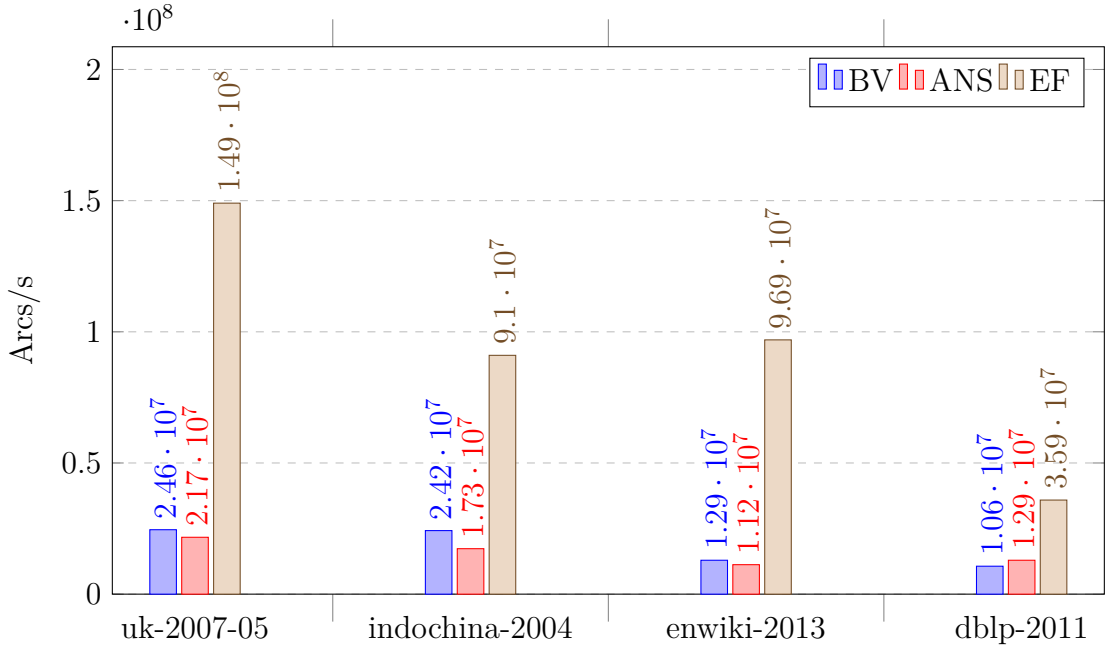

Figure 5.6: Arcs/s on the four tested graphs, the higher the better.

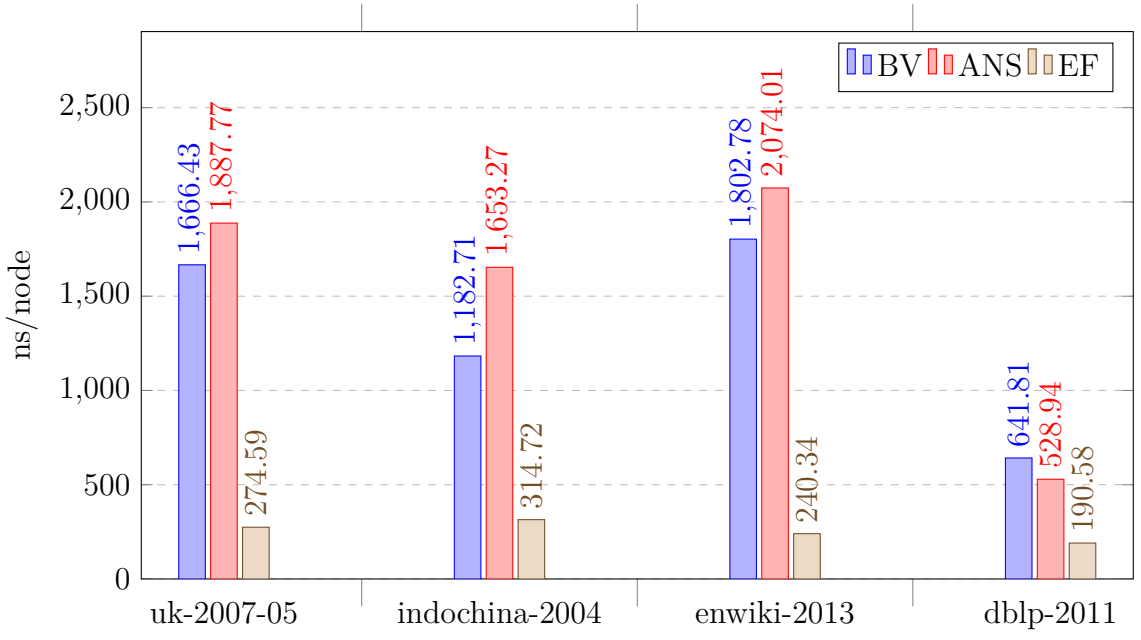

Figure 5.7: Nanoseconds per node on the tested graphs, the lower the better.

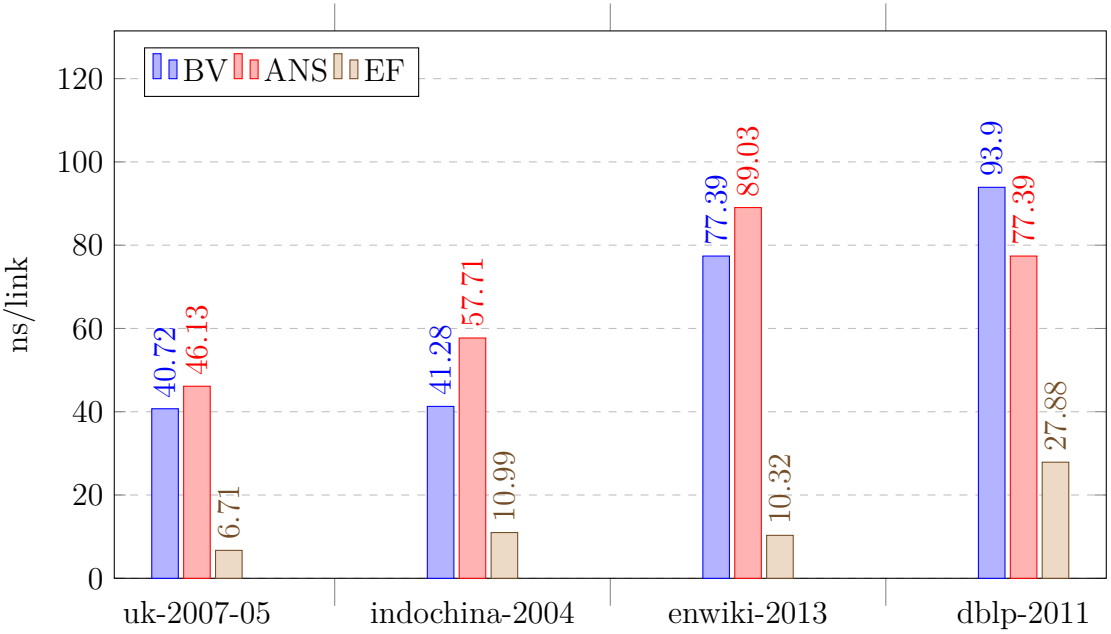

Figure 5.8: Nanoseconds per link, the lower the better.

Chapter 6

Conclusions

Bibliography

[1] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. "Permuting Web and Social Graphs". In: *Internet Mathematics* 6 (Jan. 2009), pp. 257–283. DOI: 10.1080/15427951.2009.10390641.

[2] Paolo Boldi and Sebastiano Vigna. "The WebGraph framework I: compression techniques". In: *WWW '04*. 2004.

[3] Paolo Boldi and Sebastiano Vigna. "The WebGraph framework II: codes for the World-Wide Web". In: *Data Compression Conference, 2004. Proceedings. DCC 2004*. 2004, pp. 528–. DOI: 10.1109/DCC.2004.1281504.

[4] Paolo Boldi et al. "BUbiNG: Massive Crawling for the Masses". In: *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2014, pp. 227–228.

[5] Paolo Boldi et al. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *Proceedings of the 20th international conference on World Wide Web*. Ed. by Sadagopan Srinivasan et al. ACM Press, 2011, pp. 587–596.

[6] Jarek Duda. "Asymmetric numeral systems as close to capacity low state entropy coders". In: *CoRR* abs/1311.2540 (2013). arXiv: 1311.2540. URL: http://arxiv.org/abs/1311.2540.

[7] Peter Elias. "Efficient Storage and Retrieval by Content and Address of Static Files". In: *J. ACM* 21.2 (Apr. 1974), pp. 246–260. ISSN: 0004-5411.

DOI: 10.1145/321812.321820. URL: https://doi.org/10.1145/321812.321820.

[8] D. Lemire and L. Boytsov. "Decoding billions of integers per second through vectorization". In: *Software: Practice and Experience* 45.1 (May 2013), pp. 1–29. DOI: 10.1002/spe.2203. URL: https://doi.org/10.1002%2Fspe.2203.

[9] Alistair Moffat and Matthias Petri. "ANS-Based Index Compression". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. CIKM '17. Singapore, Singapore: Association for Computing Machinery, 2017, pp. 677–686. ISBN: 9781450349185. DOI: 10.1145/3132847.3132888. URL: https://doi.org/10.1145/3132847.3132888.

[10] Alistair Moffat and Matthias Petri. "Index Compression Using Byte-Aligned ANS Coding and Two-Dimensional Contexts". In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. WSDM '18. Marina Del Rey, CA, USA: Association for Computing Machinery, 2018, pp. 405–413. ISBN: 9781450355810. DOI: 10.1145/3159652.3159663. URL: https://doi.org/10.1145/3159652.3159663.

[11] Sebastiano Vigna. "Quasi-Succinct Indices". In: (2012). DOI: 10.48550/ARXIV.1206.4300. URL: https://arxiv.org/abs/1206.4300.

[12] Hao Yan, Shuai Ding, and Torsten Suel. "Inverted index compression and query processing with optimized document ordering". In: Jan. 2009, pp. 401–410. DOI: 10.1145/1526709.1526764.