Efficient and Compact Representations of Prefix Codes

Travis Gagie† Gonzalo Navarro‡ Yakov Nekrich‡ Alberto Ordóñez‡

† Department of Computer Science, University of Helsinki, Finland
‡ David R. Cheriton School of Computer Science, University of Waterloo, Canada
+ Database Laboratory, Universidade da Coruña, Spain

Abstract

Most of the attention in statistical compression is given to the space used by the compressed sequence, a problem completely solved with optimal prefix codes. However, in many applications, the storage space used to represent the prefix code itself can be an issue. In this paper we introduce and compare several techniques to store prefix codes. Let $N$ be the sequence length and $n$ be the alphabet size. Then a naive storage of an optimal prefix code uses $O(n \log n)$ bits. Our first technique shows how to use $O(n \log \log (N/n))$ bits to store the optimal prefix code. Then we introduce an approximate technique that, for any $0 < \epsilon < 1/2$, takes $O(n \log \log (1/\epsilon))$ bits to store a prefix code with average codeword length within an additive $\epsilon$ of the minimum. Finally, a second approximation takes, for any constant $c > 1$, $O(n^{1/c} \log n)$ bits to store a prefix code with average codeword length at most $c$ times the minimum. In all cases, our data structures allow encoding and decoding of any symbol in $O(1)$ time. We implement all those techniques and compare their space/time performance against classical alternatives, showing significant practical improvements.

1 Introduction

Statistical compression is a well-established branch of Information Theory [12]. Given a text $T$ of length $N$, over an alphabet of $n$ symbols $\Sigma = \{a_1, \ldots, a_n\}$ with relative frequencies $P = \langle p_1, \ldots, p_n \rangle$ in $T$, the binary empirical entropy of the text is $H(P) = \sum_{i=1}^n p_i \log(1/p_i)$, where $\log$ denotes the logarithm in base 2. An instantaneous code assigns a binary code $c_i$ to each symbol $a_i$ so that the symbol can be decoded as soon as the last bit of $c_i$ is read from the compressed stream. An optimal instantaneous code (also called a prefix code) like Huffman’s [26] finds a prefix-free set of codes $c_i$ of length $\ell_i$, such that its average length $L(P) = \sum_{i=1}^n p_i \ell_i$ is optimal and satisfies $H(P) \leq L(P) < H(P) + 1$. This guarantees that the encoded text uses less than $N(H(P) + 1)$ bits.
Arithmetic codes achieve less space, $N\mathcal{H}(P) + 2$ bits, however they are not instantaneous, which complicates and slows down both encoding and decoding.

In this paper we are interested in instantaneous codes. In terms of the redundancy of the code, $\mathcal{L}(P) - \mathcal{H}(P)$, Huffman codes are optimal and the topic can be considered closed. How to store the prefix code itself, however, is much less studied. It is not hard to store it using $O(n \log n)$ bits, and this is sufficient when $n$ is much smaller than $N$. There are several scenarios, however, where the storage of the code itself is problematic. One example is word-based compression, which is a standard to compress natural language text. Word-based Huffman compression not only performs very competitively, offering compression ratios around 25%, but also benefits direct access, text searching, and indexing. In this case the alphabet size $n$ is the number of distinct words in the text, which can reach many millions. Other scenarios where large alphabets arise are the compression of East Asian languages and general numeric sequences. Another case is high-order compression, where one distinct code is maintained for each context of length $q$ in the text. The larger $q$, the better the compression, except that the storage of the (up to $n^q$) codes soon raises the space beyond control. A better code storage then impacts in better compression. Finally, another case arises when the text is short, for example when it is cut into several pieces that are statistically compressed independently, for example for compression boosting or for interactive communications or adaptive compression. The more effectively the codes are stored, the finer-grained can the text be cut.

During encoding and decoding, the code must be maintained in main memory to achieve reasonable efficiency, whereas the plain or the compressed text can be easily read or written in streaming mode. Therefore, the size of the code, and not that of the text, is what poses the main memory requirements for efficient compression and decompression. This is particularly stringent on mobile devices, for example, where the supply of main memory is comparatively short.

In this paper we obtain various relevant results of theoretical and practical nature about how to store a code space-efficiently, while also considering the time efficiency of compression and decompression. Our specific contributions are the following.

1. We show that it is possible to store an optimal prefix code within $O(n \log \ell_{\text{max}})$ bits, where $\ell_{\text{max}} = O\left(\min(n, \log N)\right)$ is the maximum length of a code (Theorem 1). Then we refine the space to $O(n \log \log(1/\epsilon))$ bits (Corollary 1). Within this space, encoding and decoding are carried out in constant time on a RAM machine with word size $w = \Omega(\log N)$. The result is obtained by using canonical Huffman codes, fast predecessor data structures to find code lengths, and multiary wavelet trees to represent the mapping between codewords and symbols.

2. We show that, for any $0 < \epsilon < 1/2$, it takes $O(n \log \log(1/\epsilon))$ bits to store a prefix code with average codeword length at most $\mathcal{L}(P) + \epsilon$. Encoding and decoding can be carried out in constant time on a RAM machine with word size $w = \Omega(\log n)$. Thus, if we can tolerate a small constant additive increase in the average codeword length, we can store a prefix code using only $O(n)$ bits. We obtain this result by building on the above scheme, where we use length-limited optimal prefix codes with a carefully chosen $\ell_{\text{max}}$ value.

3. We show that, for any constant $c > 1$, it takes $O(n^{1/c} \log n)$ bits to store a prefix code with average codeword length at most $c \mathcal{L}(P)$. Encoding and decoding can be carried out in constant time on a RAM machine with word size $w = \Omega(\log n)$. Thus, if we can tolerate a
small constant multiplicative increase, then we can store a prefix code in $o(n)$ bits. To achieve this result, we only store the codes that are shorter than about $\ell_{\text{max}}/c$, and use a simple code of length $\ell_{\text{max}} + 1$ for the others. Then all but the shortest codewords need to be explicitly represented.

4. We engineer and implement all the schemes above and compare them with classical implementations of optimal and suboptimal codes, showing that our techniques yield significant improvements. Our model representations use an order of magnitude less space at the price of being several times slower for compression and decompression. The additive approximations reduce these spaces and times at the expense of a very small increase in the average code length. The multiplicative approximations obtain models that are 2 or 3 orders of magnitude smaller than the classical representations and equally fast for compression and decompression, yet they pose non-negligible overheads on the optimality of the codes.

2 Related Work

A simple pointer-based implementation of a Huffman tree takes $O(n \log n)$ bits, and it is not difficult to show this is an optimal upper bound for storing a prefix code with minimum average codeword length. For example, suppose we are given a permutation $\pi$ over $n$ symbols. Let $P$ be the probability distribution that assigns probability $p_{\pi(i)} = 1/2^i$ for $1 \leq i < n$, and probability $p_{\pi(n)} = 1/2^{n-1}$. Since $P$ is dyadic, every optimal prefix code assigns codewords of length $\ell_{\pi(i)} = i$, for $1 \leq i < n$, and $\ell_{\pi(n)} = n - 1$. Therefore, given any optimal prefix code and a bit indicating whether $\pi(n-1) < \pi(n)$, we can reconstruct $\pi$. Since there are $n!$ choices for $\pi$, in the worst case it takes $\Omega(\log n!) = \Omega(n \log n)$ bits to store an optimal prefix code.

Considering the argument above, it is natural to ask whether the same lower bound holds for probability distributions that are not so skewed, and the answer is no. A prefix code is canonical \[12, 34\] if a shorter codeword is always lexicographically smaller than a longer codeword. Given any prefix code, without changing the length of the codeword assigned to any symbol, we can put the code into canonical form by just exchanging left and right siblings in the code tree. Moreover, we can reassign the codewords such that, if a symbol is lexicographically the $j$th with a codeword of length $\ell$, then it is assigned the $j$th consecutive codeword of length $\ell$. It is clear that it is sufficient to store the codeword length of each symbol to be able to reconstruct such a code, and thus the code can be represented in $O(n \log \ell_{\text{max}})$ bits.

There are more interesting upper bounds than $\ell_{\text{max}} \leq n$. Katona and Nemetz \[29\] (see also Buro \[10\]) showed that, if a symbol has relative frequency $p$, then any Huffman code assigns it a codeword of length at most $\lceil \log_\phi(1/p) \rceil$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618$ is the golden ratio, and thus $\ell_{\text{max}}$ is at most $\lceil \log_\phi(1/p_{\text{min}}) \rceil$, where $p_{\text{min}}$ is the smallest relative frequency in $P$. Note also that, since $p_{\text{min}} \geq 1/N$, it must hold $\ell_{\text{max}} \leq \log_\phi N$, therefore the canonical code can be stored in $O(n \log \log N)$ bits.

Alternatively, one can enforce a value for $\ell_{\text{max}}$ (which must be at least $\lceil \log n \rceil$) and pay a price in terms of average codeword length. The same bound above \[29\] hints at a way to achieve any desired $\ell_{\text{max}}$ value: artificially increase the frequency of the least frequent symbols until the new $p_{\text{min}}$ value is over $\phi^{-\ell_{\text{max}}}$, and then an optimal prefix code built on the new frequencies will hold the given maximum code length. Another folklore technique is to start with an optimal prefix code, and then spot all the highest nodes in the code tree with depth $\ell_{\text{max}} - d$ and more than $2^d$
leaves, for any \( d \). Then the subtrees of the parents of those nodes are made perfectly balanced. A
more sophisticated technique, by Milidiú and Laber [32], yields a performance guarantee. It first builds a
Huffman tree \( T_1 \), then removes all the subtrees rooted at depth greater than \( \ell_{\text{max}} \), builds a
complete binary tree \( T_2 \) of height \( h \) whose leaves are those removed from \( T_1 \), finds the node \( v \in T_1 \)
at depth \( \ell_{\text{max}} - h - 1 \) whose subtree \( T_v \)’s leaves correspond to the symbols with minimum total
probability, and finally replaces \( v \) by a new node whose subtrees are \( T_2 \) and \( T_3 \). They show that
the resulting average code length is at most \( \mathcal{L}(P) + 1/\phi^\ell_{\text{max}} - \lfloor \lg(n) - \ell_{\text{max}} \rfloor - 1 \).

All these approximations require \( O(n) \) time plus the time to build the Huffman tree. A technique
to obtain the optimal length-restricted prefix code, by Larmore and Hirshberg [31], performs in
\( O(n \ell_{\text{max}}) \) time by reducing the construction to a binary version of the coin-collector’s problem.

The above is an example of how an additive increase in the average codeword length may yield
less space to represent the code itself. Another well-known additive approximation follows from
Gilbert and Moore’s proof [23] that we can build an alphabetic prefix code with average codeword
length less than \( \mathcal{H}(P) + 2 \), and indeed no more than \( \mathcal{L}(P) + 1 \) [36] [43]. In an alphabetic prefix
code, the lexicographic order of the codewords is the same as that of the source symbols, so we
need to store only the code tree and not the assignment of codewords to symbols. Any code tree,
of \( n - 1 \) internal nodes, can be encoded in \( 4n + o(n) \) bits so that it can be navigated in constant
time per operation [13], and thus encoding and decoding of any symbol takes time proportional to
its codeword length.

Multiplicative approximations have the potential of yielding codes that can be represented
within \( o(n) \) bits. Adler and Maggs [1] showed it generally takes more than \( (9/40)n^{1/(20c)} \lg n \) bits
to store a prefix code with average codeword length at most \( cH(P) \). Gagie [18] [19] [20] showed that,
for any constant \( c \geq 1 \), it takes \( O(n^{1/c} \lg n) \) bits to store a prefix code with average codeword
length at most \( cH(P) + 2 \). He also showed his upper bound is nearly optimal because, for any
positive constant \( \epsilon \), we cannot always store a prefix code with average codeword length at most
\( cH(P) + o(\log n) \) in \( O(n^{1/c-\epsilon}) \) bits. Note that our result does not have the additive term “+2”
in addition to the multiplicative term, which is very relevant on low-entropy texts.

3 Representing Optimal Codes

Figure [1] (left) illustrates a canonical Huffman code. For encoding in constant time, we can simply
use an array like Codes, which stores at position \( i \) the code \( c_i \) of source symbol \( a_i \), using
\( \ell_{\text{max}} = O(\log N) \) bits for each. For decoding, the source symbols are written in an array Symb,
left-to-right order of the leaves. This array requires \( n \lg n \) bits. The access to this array is done via two
smaller arrays, which have one entry per level: \( sR[\ell] \) points to the first position of level \( \ell \) in Symb,
whereas \( \text{first}[\ell] \) stores the first code in level \( \ell \). The space for these two arrays is \( O(\ell_{\text{max}}^2) \) bits.

Then, if we have to decode the first symbol encoded in a bitstream, we first have to determine
its length \( \ell \). In our example, if the bitstream starts with 0, then \( \ell = 2 \); if it starts with 10, then
\( \ell = 3 \), and otherwise \( \ell = 4 \). Once the level \( \ell \) is found, we read the next \( \ell \) bits of the stream in \( c_i \),
and decode the symbol as \( a_i = \text{Symb}[sR[\ell] + c_i - \text{first}[\ell]] \).

The problem of finding the appropriate entry in first can be recast into a predecessor search
problem [22] [28]. We extend all the values \( \text{first}[\ell] \) by appending \( \ell_{\text{max}} - \ell \) bits at the end. In our
example, the values become 0000 = 0, 1000 = 8, and 1100 = 12. Now, we find the length \( \ell \) of
the next symbol by reading the first \( \ell_{\text{max}} \) bits from the stream, interpreting it as a binary number,
and finding its predecessor value in the set. Since we have only \( \ell_{\text{max}} = O(\log N) \) numbers in the
Figure 1: On the left, an arbitrary canonical prefix code. On the right, sorting the source symbols at each level.

set, and each has \( \ell_{\text{max}} = O(\log N) \) bits, the predecessor search can be carried out in constant time using fusion trees \([17]\) (see also Patrascu and Thorup \([41]\)) within \( O(\ell_{\text{max}}^2) \) bits of space.

Although the resulting structure allows constant-time encoding and decoding, its space usage is still \( O(n \ell_{\text{max}}) \) bits. In order to reduce it to \( O(n \log \ell_{\text{max}}) \), we will use a multiary wavelet tree data structure \([24, 15]\). In particular, we use the version that does not need universal tables \([4, \text{Thm. 7]}\).

This structure represents a sequence \( L[1,n] \) over alphabet \([1, \ell_{\text{max}}]\) using \( n \log \ell_{\text{max}} + o(n \log \ell_{\text{max}}) \) bits, and carries out the operations in time \( O(\log \ell_{\text{max}}/\log w) \). In our case, where \( \ell_{\text{max}} = O(w) \), the space is \( n \log \ell_{\text{max}} + o(n) \) bits and the time is \( O(1) \). The operations supported by wavelet trees are the following: (1) Given \( i \), retrieve \( L[i] \); (2) given \( i \) and \( \ell \in [1, \ell_{\text{max}}] \), compute \( \text{rank}_\ell(L,i) \), the number of occurrences of \( \ell \) in \( L[1,i] \); (3) given \( j \) and \( \ell \in [1, \ell_{\text{max}}] \), compute \( \text{select}_\ell(S,j) \), the position in \( L \) of the \( j \)-th occurrence of \( \ell \).

Now we reorder the symbols of the canonical Huffman tree so that, within each depth, they are in increasing order, as in Figure 1 (right). Now, the key property is that \( \text{Codes}[i] = \text{first}[i] + \text{rank}_\ell(L,i) - 1 \), where \( \ell = L[i] \), which finds the code \( c_i = \text{Codes}[i] \) of \( a_i \) in constant time. The inverse property is useful for decoding code \( c_i \) of length \( \ell \): the symbol is \( a_i = \text{Symb}[\text{sR}[\ell] + c_i - \text{first}[\ell] + 1] = \text{select}_\ell(L,c_i - \text{first}[\ell] + 1) \). Therefore, arrays \( \text{Codes}, \text{Symb}, \) and \( \text{sR} \) are not required; we can encode and decode in constant time using just the wavelet tree of \( L \) and \( \text{first} \), plus its predecessor structure. This completes the result.

**Theorem 1** Let \( P \) be the frequency distribution over \( n \) symbols for a text of length \( N \), so that an optimal prefix code has maximum codeword length \( \ell_{\text{max}} \). Then, under the RAM model with computer word size \( w = \Omega(\ell_{\text{max}}) \), we can store an optimal prefix code using \( n \log \ell_{\text{max}} + o(n) + O(\ell_{\text{max}}^2) \) bits, note that \( \ell_{\text{max}} \leq \log_2 N \). Within this space, encoding and decoding any symbol takes \( O(1) \) time.

Therefore, under mild assumptions, we can store an optimal code in \( O(n \log \log N) \) bits, with constant-time encoding and decoding operations. In the next section we refine this result further.
On the other hand, note that Theorem 1 is also valid for nonoptimal prefix codes, as long as they are canonical and their \( \ell_{\text{max}} \) is \( \mathcal{O}(w) \).

4 Additive Approximation

In this section we exchange a small additive penalty over the optimal prefix code for an even more space-efficient representation of the code, while retaining constant-time encoding and decoding.

It follows from Milidić and Laber’s bound \(^3\) that, for any \( \epsilon \) with \( 0 < \epsilon < 1/2 \), there is always a prefix code with maximum codeword length \( \ell_{\text{max}} = \lceil \lg n \rceil + \lceil \log_2(1/\epsilon) \rceil + 1 \) and average codeword length within an additive

\[
\frac{1}{\phi_{\ell_{\text{max}} - \lceil \lg(n + \lfloor \lg n \rfloor - \ell_{\text{max}}) \rceil - 1} \leq \frac{1}{\phi_{\ell_{\text{max}} - \lceil \lg n \rceil - 1}} \leq \frac{1}{\phi_{\log_2(1/\epsilon)}} = \epsilon
\]

of the minimum \( \mathcal{L}(P) \). The techniques described in Section 3 give a way to store such a code in \( n \log \ell_{\text{max}} + \Theta(n + \ell_{\text{max}}^2) \) bits, with constant-time encoding and decoding. In order to reduce the space, we note that our wavelet tree representation \(^4\) Thm. 7 in fact uses \( n \mathcal{H}_0(L) + o(n) \) bits when \( \ell_{\text{max}} = \mathcal{O}(w) \). Here \( \mathcal{H}_0(L) \) denotes the empirical zero-order entropy of \( L \). Then we obtain the following result.

**Theorem 2** Let \( \mathcal{L}(P) \) be the optimal average codeword length for a distribution \( P \) over \( n \) symbols. Then, for any \( 0 < \epsilon < 1/2 \), under the RAM model with computer word size \( w = \Omega(\log n) \), we can store a prefix code over \( P \) with average codeword length at most \( \mathcal{L}(P) + \epsilon \), using \( n \log \log(1/\epsilon) + \mathcal{O}(n) \) bits, such that encoding and decoding any symbol takes \( \mathcal{O}(1) \) time.

**Proof.** Our structure uses \( n \mathcal{H}_0(L) + o(n) + \mathcal{O}(\ell_{\text{max}}^2) \) bits, which is \( n \mathcal{H}_0(L) + o(n) \) because \( \ell_{\text{max}} = \mathcal{O}(\log n) \). To complete the proof it is sufficient to show that \( \mathcal{H}_0(S) \leq \lceil \log \log(1/\epsilon) \rceil + \mathcal{O}(1) \).

To see this, consider \( L \) as two interleaved subsequences, \( L_1 \) and \( L_2 \), of length \( n_1 \) and \( n_2 \), with \( L_1 \) containing those lengths \( \leq \lceil \lg n \rceil \) and \( L_2 \) containing those greater. Thus \( n \mathcal{H}_0(L) \leq n_1 \mathcal{H}_0(L_1) + n_2 \mathcal{H}_0(L_2) + n \) (from an obvious encoding of \( L \) using \( L_1 \), \( L_2 \), and a bitmap).

Let us call \( \text{occ}(\ell, L_1) \) the number of occurrences of symbol \( \ell \) in \( L_1 \). Since there are at most \( 2^\ell \) codewords of length \( \ell \), assume we complete \( L_1 \) with spurious symbols so that it has exactly \( 2^\ell \) occurrences of symbol \( \ell \). This completion cannot decrease \( n_1 \mathcal{H}_0(L_1) = \sum_{\ell=1}^{\lceil \lg n \rceil} \text{occ}(\ell, L_1) \log_{\phi_{\text{occ}(\ell, L_1)}} n_1 \), as increasing some \( \text{occ}(\ell, L_1) \) to \( \text{occ}(\ell, L_1) + 1 \) produces a difference of \( f(n_1) - f(\text{occ}(\ell, L_1)) \geq 0 \), where \( f(x) = (x + 1) \log(x + 1) - x \log x \) is increasing. Hence we can assume \( L_1 \) contains exactly \( 2^\ell \) occurrences of symbol \( 1 \leq \ell \leq \lceil \lg n \rceil \); straightforward calculation then shows \( n_1 \mathcal{H}_0(L_1) = \mathcal{O}(n_1) \).

On the other hand, \( L_2 \) contains at most \( \ell_{\text{max}} - \lceil \lg n \rceil \) distinct values, so \( \mathcal{H}_0(L_2) \leq \lceil \ell_{\text{max}} - \lceil \lg n \rceil \rceil \), unless \( \ell_{\text{max}} = \lceil \lg n \rceil \), in which case \( L_2 \) is empty and \( n_2 \mathcal{H}_0(L_2) = 0 \). Thus \( n_2 \mathcal{H}_0(L_2) \leq n_2 \log(\lceil \log_2(1/\epsilon) \rceil + 1) = n_2 \log \log(1/\epsilon) + \mathcal{O}(n_2) \). Combining both bounds, we get \( \mathcal{H}_0(L) \leq \log \log(1/\epsilon) + \mathcal{O}(1) \) and the theorem holds.

In other words, under mild assumptions, we can store a code using \( \mathcal{O}(n \log \log(1/\epsilon)) \) bits at the price of increasing the average codeword length by \( \epsilon \), and in addition have constant-time encoding and decoding. For constant \( \epsilon \), this means that the code uses just \( \mathcal{O}(n) \) bits at the price of an arbitrarily small constant additive penalty over the shortest possible prefix code. Figure 2 shows an example. Note that the same reasoning of this proof, applied over the encoding of Theorem 1, yields a refined upper bound.
Corollary 1 Let $P$ be the frequency distribution of $n$ symbols for a text of length $N$. Then, under the RAM model with computer word size $w = \Omega(\log N)$, we can store an optimal prefix code for $P$ using $n \log \log(N/n) + \mathcal{O}(n + \log^2 N)$ bits, while encoding and decoding any symbol in $\mathcal{O}(1)$ time.

Proof. Proceed as in the proof of Theorem 2 using that $\ell_{\text{max}} \leq \log_{\phi} N$ and putting inside $L_1$ the lengths up to $\lceil \log_{\phi} n \rceil$. Then $n_1 \mathcal{H}(L_1) = \mathcal{O}(n_1)$ and $n_2 \mathcal{H}(L_2) \leq \log \log(N/n) + \mathcal{O}(n_2)$.

\section{Multiplicative Approximation}

In this section we obtain a multiplicative rather than an additive approximation to the optimal prefix code, in order to achieve a sublinear-sized representation of the code. We will divide the alphabet into frequent and infrequent symbols, and store information about only the frequent ones.

Given a constant $c > 1$, we use Milidiú and Laber’s algorithm \cite{MilidiuLaber2007} to build a prefix code with maximum codeword length $\ell_{\text{max}} = \lceil \log n \rceil + \lceil 1/(c-1) \rceil + 1$ (our final codes will have length up to $\ell_{\text{max}} + 1$). We call a symbol’s codeword short if it has length at most $\ell_{\text{max}}/c + 2$, and long otherwise. Notice there are $S \leq 2^{\ell_{\text{max}}/c+2} = \mathcal{O}(n^{1/c})$ symbols with short codewords. Also, although applying Milidiú and Laber’s algorithm may cause some exceptions, symbols with short codewords are usually more frequent than symbols with long ones. We will hereafter call frequent/infrequent symbols those encoded with short/long codewords.

Note that, if we build a canonical code, all the short codewords will precede the long ones. We first describe how to handle the frequent symbols. A perfect hash data structure \cite{HnichGervet2005} will map the frequent symbols in $[1,n]$ to the interval $[1,S]$ in constant time. The reverse mapping is done via a plain array $\text{ihash}[1,S]$ that stores the original symbol that corresponds to each mapped symbol. We use this mapping also to reorder the frequent symbols so that the corresponding prefix in array $\text{Symb}$ (recall Section \ref{sec:prefix-code}) reads $1,2,\ldots,S$. Thanks to this, we can encode and decode any frequent symbol using just $\text{first}$, $\text{sR}$, predecessor structures on both of them, and the tables $\text{hash}$ and $\text{ihash}$. To encode a frequent symbol $a_i$, we find it in $\text{hash}$, obtain the mapped symbol $a' \in [1,S]$, find the predecessor $\text{sR}[\ell]$ of $a'$ and then the code is the $\ell$-bit integer $c_i = \text{first}[\ell] + a' - \text{sR}[\ell]$. To decode a short code $c_i$, we first find its corresponding length $\ell$ using the predecessor structure on $\text{first}$, then obtain its mapped code $a' = \text{sR}[\ell] + c_i - \text{first}[\ell]$, and finally the original symbol is $i = \text{ihash}[a']$. Structures $\text{hash}$ and $\text{ihash}$ require $\mathcal{O}(n^{1/c} \log n)$ bits, whereas $\text{sR}$ and $\text{first}$, together with their predecessor structures, require less, $\mathcal{O}(\log^2 n)$ bits.

The long codewords will be replaced by new codewords, all of length $\ell_{\text{max}} + 1$. Let $c_{\text{long}}$ be the first long codeword and let $\ell$ be its length. Then we form the new codeword $c'_{\text{long}}$ by appending $\ell_{\text{max}} + 1 - \ell$ zeros at the end of $c_{\text{long}}$. The new codewords will be the $(\ell_{\text{max}}+1)$-bit integers $c'_{\text{long}}, c'_{\text{long}} + 1, \ldots, c'_{\text{long}} + n - 1$. An infrequent symbol $a_i$ will be mapped to code $c'_{\text{long}} + i - 1$ (frequent symbols $a_i$ will leave unused symbols $c'_{\text{long}} + i - 1$). Figure \ref{fig:long-codewords} shows an example.

Since $c > 1$, we have $n^{1/c} < n/2$ for sufficiently large $n$, so we can assume without loss of generality that there are fewer than $n/2$ short codewords and thus there are at least $n/2$ long codewords. Since every long codeword is replaced by at least two new codewords, the total number of new codewords is at least $n$. Thus there are sufficient slots to assign codewords $c'_{\text{long}}$ to $c'_{\text{long}} + n - 1$.

\footnote{If this is not the case, then $n = \mathcal{O}(1)$, so we can use any optimal encoding: there will be no redundancy over $\mathcal{L}(P)$ and the asymptotic space formula for storing the code will still be valid.}
Figure 2: An example of Milidiú and Laber’s algorithm [32]. On top, a canonical Huffman Tree. We set $l_{\text{max}} = 5$ and remove all the symbols below that level (marked with the dotted line), which yields three empty nodes (marked as black circles in the top tree). In the middle, those black circles are replaced by the deepest symbols below level $l_{\text{max}}$: 1, 8, and 10. The other symbols below $l_{\text{max}}$, 9, 13, 12 and 5, form a balanced binary tree that is hung from a new node created as the left child of the root (in black in the middle tree). The former left child of the root becomes the left child of this new node. Finally, on the bottom, we transform the middle tree into its canonical form, but sorting those symbols belonging to the same level in increasing order.
Figure 3: An example of the multiplicative approximation, with \( n = 16 \) and \( c = 3 \). The top tree is the result of applying the algorithm of Milidiú and Laber to a given set of codes. Now, we set \( \ell_{\text{max}} = 6 \) according to our formula, and declare short those codewords of lengths up to \( \lfloor \ell_{\text{max}} / c \rfloor + 2 = 4 \). Short codewords (above the dashed line on top) are stored unaltered but with all symbols at each level sorted in increasing order (middle tree). Long codewords (below the dashed line) are extended up to length \( \ell_{\text{max}} + 1 = 7 \) and reassigned a code according to their values in the contiguous slots of length 7 (those in gray in the middle). Thus, given a long codeword \( x \), its code is directly obtained as \( c'_{\text{long}} + x - 1 \), where \( c'_{\text{long}} = 1100000 \) is the first code of length \( \ell_{\text{max}} + 1 \). On the bottom, a representation of the hash and inverse hash to code/decode short codewords. We set the hash size to \( m = 13 \) and \( h(x) = (5x + 7) \mod m \). We store the code associated with each cell.
To encode an infrequent symbol \( a_i \), we first fail to find it in table \( \text{hash} \). Then, we assign it the \((\ell_{\text{max}}+1)\)-bits long codeword \( c_{i}^{'\text{long}} + i - 1 \). To decode a long codeword, we first read \( \ell_{\text{max}} + 1 \) bits into \( c_i \). If \( c_i \geq c_{i}^{\text{long}} \), then the codeword is long, and corresponds to the source symbol \( a_{c_i-c_{i}^{\text{long}}+1} \). Note that we use no space to store the infrequent symbols. This leads to proving our result.

**Theorem 3** Let \( L(P) \) be the optimal average codeword length for a distribution \( P \) over \( n \) symbols. Then, for any constant \( c > 1 \), under the RAM model with computer word size \( w = \Omega(\log n) \), we can store a prefix code over \( P \) with average codeword length at most \( c L(P) \), using \( O(n^{1/c} \log n) \) bits, such that encoding and decoding any symbol takes \( O(1) \) time.

**Proof.** Only the claimed average codeword length remains to be proved. By analysis of the algorithm by Milidiú and Laber [32] we can see that the codeword length of a symbol in their length-restricted code exceeds the codeword length of the same symbol in an optimal code by at most 1, and only when the codeword length in the optimal code is at least \( \ell_{\text{max}} - [\log n] - 1 = [1/(c-1)] \). Hence, the codeword length of a frequent symbol exceeds the codeword length of the same symbol in an optimal code by a factor of at most \( \frac{1}{[1/(c-1)]+1} \leq c \). Every infrequent symbol is encoded with a codeword of length \( \ell_{\text{max}} + 1 \). Since the codeword length of an infrequent symbol in the length-restricted code is more than \( \ell_{\text{max}}/c + 2 \), its length in an optimal code is more than \( \ell_{\text{max}}/c + 1 \). Hence, the codeword length of an infrequent symbol in our code is at most \( \frac{\ell_{\text{max}}+1}{\ell_{\text{max}}/c+1} < c \) times greater than the codeword length of the same symbol in an optimal code. Hence, the average codeword length for our code is less than \( c \) times the optimal one. \( \square \)

Again, under mild assumptions, this means that we can store a code with average length within \( c \) times the optimum, in \( O(n^{1/c} \log n) \) bits and allowing constant-time encoding and decoding.

### 6 Experimental Results

We engineer and implement the optimal and approximate code representations described above, obtaining complexities that are close to the theoretical ones. For the sake of comparison, we also implement some folklore alternatives to represent prefix codes. Our comparisons will measure the size of the code representation, the encoding and decoding time and, in the case of the approximations, the redundancy on top of \( L(P) \).

#### 6.1 Implementations

Our constant-time results build on two data structures. One is the multiary wavelet tree [15, 4]. A practical study [6] shows that multiary wavelet trees can be faster than binary ones, but require significantly more space (even with the better variants they design). To prioritize space, we will use binary wavelet trees, which perform the operations in time \( O(\log \ell_{\text{max}}) = O(\log \log N) \).

The second constant-time data structure is the fusion tree [17], of which there are no practical implementations as far as we know. Even implementable logarithmic predecessor search data structures, like van Emde Boas trees [44], are worse than binary search for small universes like our \([1, \ell_{\text{max}}] = [1, O(\log N)] \) range. With a simple binary search on \texttt{first} we obtain a total encoding and decoding time of \( O(\log \log N) \), which is sufficiently good for practical purposes. Even more, preliminary experiments showed that sequential search on \texttt{first} is about as good as binary search.
in our test collections. Although sequential search costs $O(\log N)$ time, the higher success of instruction prefetching makes it much faster. Thus, our experimental results use sequential search.

To achieve space close to $nH_0(L)$ in the wavelet tree, we use a Huffman-shaped wavelet tree \[38\]. The bitmaps of the wavelet tree are represented in plain form and using a space overhead of 37.5\% to support rank/select operations \[37\]. Besides, we enhance these bitmaps with a small additional index to speed up select operations \[40\]. The total space of wavelet tree is at most $n(H_0(L) + 2) + o(n(H_0(L) + 2)) + O(L \log n)$ bits, and very close to optimal in practice. An earlier version of our work \[39\] recasts this wavelet tree into a compressed permutation representation \[3\] of vector Symb, which leads to a similar implementation.

For the additive approximation of Section 4, we use the same implementation as for the exact version, after modifying the code tree as described in that section. The lower number of levels will automatically make sequence $L$ more compressible and the encoding/decoding faster (as it depends on the average leaf depth).

For the multiplicative approximation of Section 5, we implement table hash with double hashing. The hash function is of the form $h(x,i) = (h_1(x) + (i - 1) \cdot h_2(x)) \mod m$ for the $i$th trial, where $h_1(x) = x \mod m$, $h_2(x) = 1 + (x \mod m')$, $m$ is a prime number, and $m' = m - 1$. Predecessor searches over $sR$ and first are done via binary search, for the same reasons discussed above.

**Classical Huffman codes.** As a baseline to compare with our encoding, we use a simple representation like in Figure 1 (left), using $n\ell_{\max}$ bits for Codes, $n\log n$ bits for Symb, $\ell_{\max}^2$ bits for first, and $\ell_{\max}\log n$ bits for $sR$. For decompression, we iteratively probe the next $\ell$ bits from the compressed sequence, where $\ell$ is the next available tree depth. If the relative numeric code resulting from reading $\ell$ bits exceeds the number of nodes at this level, we probe the next level, and so on until finding the right length \[42\]. When measuring compression/decompression times, we will only consider the space needed for compression/decompression (whereas our structure is a single one for both operations).

**Hu-Tucker codes.** As a representative of a suboptimal code that requires little storage space \[9\], we also implement alphabetic codes, using the Hu-Tucker algorithm \[25\ 30\]. This algorithm takes $O(n \log n)$ time and yields the optimal alphabetic code, which guarantees an average code length below $H(P) + 2$. As the code is alphabetic, no permutation of symbols needs to be stored; the $i$th leaf of the code tree corresponds to the $i$th source symbol. On the other hand, the tree shape is arbitrary. We implement the code tree using succinct tree representations, more precisely the so-called FF \[2\], which efficiently supports the required navigation operations. This representation requires 2.37 bits per tree node, that is, 4.74$n$ bits for our tree (which has $n$ leaves and $n - 1$ internal nodes). FF represents general trees, so we convert the binary code tree into a general tree using the folklore mapping: we identify the left child of the code tree with the first child in the general tree, and the right child of the code tree with the next sibling in the general tree. The general tree has an extra root node whose children are the nodes in the rightmost path of the code tree.

With this representation, compression of symbol $c$ is carried out by starting from the root and descending towards the $c$th leaf. We use the number of leaves on the left subtree to decide whether to go left or right. The left/right decisions made in the path correspond to the code. In the general tree, we compute the number of nodes $k$ in the subtree of the first child, and then the number of leaves in the code tree is $k/2$. For decompression, we start from the root and descend left or right depending on the bits of the code. Each time we go right, we accumulate the number of leaves on
Table 1: Main statistics of the texts used.

| Collection | Length ($N$) | Alphabet Entropy ($H(P)$) | Depth ($\ell_{\text{max}}$) | Level entr. ($H_0(L)$) |
|------------|-------------|---------------------------|-----------------------------|------------------------|
| EsWiki     | 200,000,000 | 1,634,145                 | 11.12                       | 28                     | 2.24                   |
| EsInv      | 300,000,000 | 1,590,398                 | 12.47                       | 28                     | 2.82                   |
| Indo       | 120,000,000 | 3,715,187                 | 16.29                       | 27                     | 2.51                   |

Table 2: Estimated size of various model representations.

| Collection | Naive ($nw$) | Engineered ($n \ell_{\text{max}}$) | Canonical ($n \lg n$) | Ours ($nH_0(L)$) |
|------------|--------------|-------------------------------------|-----------------------|-----------------|
| EsWiki     | 6.23 MB      | 5.45 MB                              | 4.02 MB               | 0.44 MB         |
| EsInv      | 6.07 MB      | 5.31 MB                              | 3.91 MB               | 0.53 MB         |
| Indo       | 14.17 MB     | 11.96 MB                             | 9.67 MB               | 1.11 MB         |

the left, so that when we arrive at a leaf the decoded symbol is the final accumulated value plus 1.

6.2 Experimental Setup

We used an isolated Intel(R) Xeon(R) E5620 running at 2.40GHz with 96GB of RAM memory. The operating system is GNU/Linux, Ubuntu 10.04, with kernel 2.6.32-33-server.x86_64. All our implementations use a single thread and are coded in C++. The compiler is gcc version 4.4.3, with -O9 optimization. Time results refer to CPU user time.

We use three datasets in our experiments. EsWiki is a sequence of word identifiers obtained by stemming the Spanish Wikipedia with the Snowball algorithm. Compressing natural language using word-based models is a strong trend in text databases [33]. EsInv is the concatenation of the (differentially encoded) inverted lists of a random sample of the Spanish Wikipedia. These differences are usually encoded using suboptimal codes due to the large alphabet sizes [45]. Finally, Indo is the concatenation of the adjacency lists of Web graph Indochina-2004 available at http://law.di.unimi.it/datasets.php. Compressing adjacency lists to zero-order entropy is a simple and useful tool for graphs with power-law degree distributions, although it must be combined with stronger techniques [11]. We use a prefix of each of the sequences to speed up experiments.

Table 1 gives various statistics on the collections. Apart from $N$ and $n$, we give the empirical entropy of the sequence ($H(P)$, in bits per symbol or bps), the maximum length of a code ($\ell_{\text{max}}$), and the zero-order entropy of the sequence of levels ($H_0(L)$, in bps). It can be seen that $H_0(L)$ is significantly smaller than $\lg \ell_{\text{max}}$, thus our compressed representation of $L$ will be much smaller than the worst-case upper bound $n \lg \ell_{\text{max}}$ bits.

Table 2 shows how the sizes of different representations of the model can be estimated using these numbers. The first column gives $nw$, the size of a naive model representation using computer words of $w = 32$ bits. The second gives $n \ell_{\text{max}}$, and corresponds to a more engineered representation where we use only the number of bits required to describe a codeword. In these two, more structures are needed for decoding but we ignore them. The third column gives $n \lg n$, which is the main space cost for a canonical Huffman tree representation: basically the permutation of symbols (different ones for encoding and decoding). Finally, the fourth column gives $nH_0(L)$, which is a good predictor of the size of our model representation. From the estimations, it is clear that our technique is much
more effective than folklore representations to reduce the space, and that we can expect space reductions of around one order of magnitude.

6.3 Representing Optimal Codes

Figure 4 compares compression and decompression times, as a function of the space used by the code representations, of our new representation (COMPR) versus the engineered representation described in Section 6.1 (TABLE). We used sampling periods of \{8, 16, 32, 64, 128, \infty\} for the auxiliary data structures added to the wavelet tree bitmaps to speed up \textit{select} operations \cite{40}.

It can be seen that our compressed representations takes just around 12\% of the space of the table implementation for compression (almost an order of magnitude smaller), while being 3–4 times slower. Note that compression is performed by carrying out \textit{rank} operations on the wavelet tree bitmaps. Therefore, we do not consider the space overhead incurred to speed up \textit{select} operations, thus we only plot a single point for technique COMPR at compression charts.

For decompression, our solution (COMPR) takes 17\% to 80\% of the space of the naive implementation (TABLE), but it is also 4–11 times slower. This is because our solution uses operation \textit{select} for decompression, and this is slower than \textit{rank} even with the structures for speeding it up.

Overall, our compact representation is able to compress at a rate around 21 MB/sec and decompress at 4–8 MB/sec, while using much less space than a classical Huffman implementation.

Finally, note we only need a single data structure to both compress and decompress, while the naive approach uses different tables for each operation. In the cases where both functionalities are simultaneously necessary (as in compressed sequence representations \cite{38}), our structure uses around 7\%–33\% of the space needed by a classical representation (depending on the desired decompression speed).

6.4 Length-Limited Codes

In the theoretical description, we refer to an optimal technique for limiting the length of the code trees to a given value \( \ell_{\text{max}} \geq \lceil \log n \rceil \) \cite{31}. Figure 5 compares this strategy with several approximations (exact numbers are given at the end, in Table 3):

- **Milidiú**: the approximate technique proposed by Milidiú and Laber \cite{32} that nevertheless guarantees the upper bound we have used in the paper. It takes \( O(n) \) time.

- **Increase**: inspired in the bounds of Katona and Nemetz \cite{29}, we start with \( f = 2 \) and set to \( f \) the frequency of each symbol whose frequency is \( < f \). Then we build the Huffman tree, and if its height is \( \leq \ell_{\text{max}} \), we are done. Otherwise, we increase \( f \) by 1 and repeat the process. Since the Huffman construction algorithm is linear-time once the symbols are sorted by frequency and the process does not need to reorder them, this method takes \( O(n \log(n \varphi^{-(\ell_{\text{max}})})) = O(n \log n) \) time if we use exponential search to find the correct \( f \) value.

- **Increase-A**: analogous to \textit{Increase}, but instead adds \( f \) to the frequency of each symbol.

- **Balance**: the folklore technique that balances the parents of the maximal subtrees that, even if balanced, exceed the maximum allowed height. It also takes \( O(n) \) time. In the case of a canonical Huffman tree, this is even simpler, since only one node along the rightmost path of the tree needs to be balanced.
Figure 4: Code representation size versus compression/decompression time for a classical representation (TABLE) and ours (COMPR). Time is measured in nanoseconds per symbol.
• **Optimal**: the package-merge algorithm of Larmore and Hirshberg [31]. Its time complexity is $O(n \ell_{\text{max}})$.

Table 3 also shows the average code length required by Huffman and Hu-Tucker codes [25].

It can be seen that the average code lengths obtained by Milidiú, although they have theoretical guarantees, are among the worst in practice. They are comparable with those of Balance, a simpler and still linear-time heuristic, which however does not provide any guarantee and sometimes can only return a completely balanced tree. On the other hand, technique **Increase** performs better than or equal to **Increase-A**, and actually matches the average code length of **Optimal** systematically in the three collections.

Techniques Milidiú, Balance, and **Optimal** are all equally fast in practice, taking about 2 seconds to find their length-restricted code in our collections. The time for **Increase** and **Increase-A** depends on the value of $\ell_{\text{max}}$. For large values of $\ell_{\text{max}}$, they also take around 2 seconds, but this raises up to 20 seconds when $\ell_{\text{max}}$ is closer to $\lceil \lg n \rceil$ (and thus the value $f$ to add is larger, up to 100–300 in our sequences).

In practice, technique **Increase** can be recommended for its extreme simplicity to implement and very good approximation results. If the construction time is an issue, then **Optimal** should be used. It performs fast in practice and it is not so hard to implement. For the following experiments, we will use the results of **Optimal/Icrease**.

### 6.5 Approximations

Now we evaluate the additive and multiplicative approximations, in terms of average code length $L$, compression and decompression performance. We compare two optimal model representations, OPT-T and OPT-C, which correspond to TABLE and COMPR of Section 6.3. We also include in the comparison ADD+T and ADD+C, which are the additive approximations (Section 4) obtained by restricting the maximum code lengths, and storing the resulting codes using TABLE or COMPR, respectively. Finally, for the multiplicative approximation (Section 5), we introduce MULT-27, MULT-26, and MULT-25, which limit $\ell_{\text{max}}$ to 27, 26, and 25, respectively, and use consecutive $c$ values starting at 2. For all the solutions that use a wavelet tree, we have fixed a select sampling rate to 64.

Figure 6 shows the results in terms of bps for storing the model versus the resulting redundancy of the code, measured as $L(P)/H(P)$, where $H(P)$ is Huffman’s average code length. Both columns of Figure 6 plot the same results, but on the right we put the $x$-axis in logscale. Not also that the $y$-axis is upper limited by $\lg n / H(P)$.

It is clear that the multiplicative approach is extremely efficient for storing the model, but it is also the worst in terms of code optimality. In general, it quickly goes beyond $[\lg n] / H(P)$, at which point it is better to use a plain code of $[\lg n]$ bits. Still, when used cautiously, it can offer model sizes 2 or 3 orders of magnitude smaller than the classical ones, with a degradation of 15% to 40% in the average code length.

The additive approximations have a milder impact. They do not worsen the average code length by more than 1–2 bps. In terms of model size, ADD+T reduces the size of OPT-T by a moderate margin. The gain is more noticeable, however, when using our compact representation: ADD+C uses about half the space needed by OPT-C to represent the model.

---

2There are even some public implementations, for example [https://gist.github.com/imaya/3985581](https://gist.github.com/imaya/3985581)
Figure 5: Comparison of the length-restricted approaches measured as their additive redundancy over Huffman codes, $H(P)$, for each value of $\ell_{\text{max}}$. The maximum code length of Huffman is one plus the last $x$ value in each plot.
| $\ell_{\text{max}}$ | Milidiú Balance Increase Increase-A Optimal Hu-Tucker Huffman |
|----------------|---------------------------------------------------------------|
| 30             |                                                                 |
| 11.616         | 11.1690 11.1522 11.1574 11.1522                               |
| 28             | 11.6016 11.2145 11.1576 11.1607 11.1576                       |
| 27             | 11.5873 11.3446 11.1763 11.1778 11.1763                       |
| 26             | 11.5853 20.9991 11.2271 11.2277 11.2271                       |
| 25             | 11.5797 20.9991 11.3554 11.3559 11.3554                       |
| 24             | 20.9991 20.9991 11.6882 11.6882 11.6882                       |
| 23             | 20.9991 20.9991 12.9683 12.9684 12.9683                       |
| 22             | 20.9991 20.9991 12.9683 12.9684 12.9683                       |
| 21             | 20.9993 20.9993 13.3880 13.4251 13.3880                       |
| 29 30 28 27 26 25 24 23 22 21 | 12.6524 12.5276 12.5276 12.5285 12.5285 12.5285 12.5376 12.5376 12.5376 12.5376 |
Figure 6: Relation between the size of the model and the average code length. The right column plots the same data as the left one, but with the x-axis in logscale.
Figure 7 compares these representations in terms of compression and decompression performance. We use ADD+C+\( \ell_{\text{max}} \) to refer to a specific value of \( \ell_{\text{max}} \) (in the case of ADD+T, the results are the same for all the tested \( \ell_{\text{max}} \) values). In this case, we set the select samplings for the wavelet tree to \((8, 32, \infty)\) to generate the curves.

It can be seen that the multiplicative approach is very fast, as much as the table-based approaches ADD+T and OPT-T. The additive approaches that use a reduced model (ADD+C+\( \ell_{\text{max}} \)) become faster as the model becomes smaller, since the wavelet tree operates on a smaller alphabet of depths. That is, in an additive context, it is convenient to limit \( \ell_{\text{max}} \) as much as possible within a tolerated code length. For instance, in EsWiki, an optimal solution (OPT-C) with a sampling rate \( \infty \) requires more model space and much more compression/decompression time than an additive approximation with sampling rate 32. On our sequences, we already obtain small models with an additive overhead of 0.05.

Finally, we can see that our compact implementation of Hu-Tucker codes achieves competitive space, but it is an order of magnitude slower than our additive approximations, which can always use simultaneously less space and (much) less time. With respect to the average code length, Table 3 shows that Hu-Tucker codes are equivalent to our additive approximations with \( \ell_{\text{max}} = 23 \) on EsWiki and EsInv, and \( \ell_{\text{max}} = 24 \) on Indo. This shows that the use of alphabetic codes as a suboptimal code to reduce the model representation size is inferior, in all aspects, to our additive approximations. We remark that alphabetic codes are interesting by themselves for other reasons, in contexts where preserving the order of the source symbols is important.

7 Conclusions

We have explored the problem of providing compact representations of Huffman models. The model size is relevant in several applications, particularly because it must reside in main memory for efficient compression and decompression. We have proposed new representations achieving constant compression and decompression time per symbol while using \( O(n \log \log (N/n)) \) bits per symbol, where \( n \) is the alphabet size and \( N \) the sequence length. This is in contrast to the \( O(n \log N) \) bits used by classical representations. In practice, we achieved 8-fold space reductions for compression and up to 6-fold for decompression. While this comes at the price of increased compression and decompression time (3–4 times slower at compression and 4–11 at decompression), the obtained throughputs are still acceptable, and the space reduction can make the difference between fitting the model in main memory or not.

We also showed that, by accepting a small additive overhead of \( \epsilon \) on the average code length, the model can be stored in \( O(n \log \log (1/\epsilon)) \) bits, while maintaining constant compression and decompression time. In practice, these additive approximations can halve both the model size and the compression/decompression time, while incurring a very small increase (\( \epsilon = 0.05 \)) in the average code length.

Finally, we showed that a multiplicative penalty in the average code length allows storing the model in \( o(n) \) bits. In practice, the reduction in model size is sharp: 2 to 3 orders of magnitude, while the compression and decompression times are as good as with classical implementations. The penalty in the average code length goes in practice from 15% to 40%. This makes multiplicative approximations an interesting alternative when the space for the model is severely limited and we still want to obtain some compression.

Some interesting challenges for future work are:
Figure 7: Space/time performance of the approximate approaches. Times are in nanoseconds per symbol.
• Adapt these representations to dynamic scenarios, where the model undergoes changes as compression/decompression progresses. While our compact representations can be adapted to support updates, the main problem is how to efficiently maintain a dynamic canonical Huffman code. We are not aware of such a technique.

• Find more efficient representations of alphabetic codes. Our baseline achieves reasonably good space, but the navigation on the compact tree representations slows it down considerably. It is possible that faster representations supporting left/right child and subtree size exist.

• Find constant-time encoding and decoding methods that are fast and compact in practice. Multiary wavelet trees [6] are faster than binary wavelet trees, but generally use much more space. Giving them the shape of a (multiary) Huffman tree and using plain representations for the sequences in the nodes could reduce the space gap with our binary Huffman-shaped wavelet trees used to represent $L$. As for the fusion trees, looking for a practical implementation of trees with arity $w^\epsilon$, which outperforms a plain binary search, is interesting not only for this problem, but in general for predecessor searches on small universes.

References

[1] M. Adler and B. M. Maggs. Protocols for asymmetric communication channels. *Journal of Computer and System Sciences*, 63(4):573–596, 2001.

[2] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97, 2010.

[3] J. Barbay and G. Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013.

[4] D. Belazzougui and G. Navarro. New lower and upper bounds for representing sequences. In *Proc. 20th Annual European Symposium on Algorithms (ESA)*, LNCS 7501, pages 181–192, 2012.

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

[6] A. Bowe. *Multiary Wavelet Trees in Practice*. Honours thesis, RMIT University, Australia, 2010.

[7] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, 15(6):527–557, 2012.

[8] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.

[9] N. Brisaboa, G. Navarro, and A. Ordóñez. Smaller self-indexes for natural language. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 7608, pages 372–378, 2012.

[10] M. Buro. On the maximum length of Huffman codes. *Information Processing Letters*, 45(5):219–223, 1993.
[11] F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4):article 16, 2010.

[12] T. Cover and J. Thomas. *Elements of Information Theory*. Wiley, 1991.

[13] P. Davoodi, R. Raman, and S. Rao Satti. Succinct representations of binary trees for range minimum queries. In *Proc. 18th Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 7434, pages 396–407, 2012.

[14] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

[15] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.

[16] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[17] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[18] T. Gagie. Compressing probability distributions. *Information Processing Letters*, 97(4):133–137, 2006.

[19] T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[20] T. Gagie. Dynamic asymmetric communication. *Information Processing Letters*, 108(6):352–355, 2008.

[21] T. Gagie, G. Navarro, and Y. Nekrich. Fast and compact prefix codes. In *Proc. 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5901, pages 419–427, 2010.

[22] T. Gagie and Y. Nekrich. Worst-case optimal adaptive prefix coding. In *Proc. 9th Symposium on Algorithms and Data Structures (WADS)*, pages 315–326, 2009.

[23] E. N. Gilbert and E. F. Moore. Variable-length binary encodings. *Bell System Technical Journal*, 38:933–967, 1959.

[24] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[25] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal of Applied Mathematics*, 21(4):514–532, 1971.

[26] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1090–1101, 1952.

[27] J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proc. 18th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 174–184, 2011.
M. Karpinski and Y. Nekrich. A fast algorithm for adaptive prefix coding. *Algorithmica*, 55(1):29–41, 2009.

G. O. H. Katona and T. O. H. Nemetz. Huffman codes and self-information. *IEEE Transactions on Information Theory*, 22(3):337–340, 1976.

D. E. Knuth. *The Art of Computer Programming. Vol. 3: Sorting and Searching*. Addison-Wesley, 1973.

L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464–473, 1990.

R. L. Milidiú and E. S. Laber. Bounding the inefficiency of length-restricted prefix codes. *Algorithmica*, 31(4):513–529, 2001.

A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.

A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.

E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

N. Nakatsu. Bounds on the redundancy of binary alphabetical codes. *IEEE Transactions on Information Theory*, 37(4):1225–1229, 1991.

G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics (JEAl)*, 13:article 2, 2009.

G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

G. Navarro and A. Ordóñez. Compressing Huffman models on large alphabets. In *Proc. 23rd Data Compression Conference (DCC)*, pages 381–390, 2013.

G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. 11th International Symposium on Experimental Algorithms (SEA)*, LNCS 7276, pages 295–306, 2012.

M. Patrascu and M. Thorup. Time-space trade-offs for predecessor search. *CoRR*, cs/0603043v1, 2008. [http://arxiv.org/pdf/cs/0603043v1](http://arxiv.org/pdf/cs/0603043v1).

E. S. Schwarz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, 1964.

D. Sheinwald. On binary alphabetic codes. In *Proc. 2nd Data Compression Conference (DCC)*, pages 112–121, 1992.

P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.