Compressing Huffman Models on Large Alphabets[*]

Gonzalo Navarro
Dept. of Computer Science
University of Chile
gnavarro@dcc.uchile.cl

Alberto Ordóñez
Database Laboratory
University of A Coruña
alberto.ordonez@udc.es

Abstract

A naive storage of a Huffman model on a text of length n over an alphabet of size σ requires $O(\sigma \log n)$ bits. This can be reduced to $\sigma \log \sigma + O(\sigma)$ bits using canonical codes. This overhead over the entropy can be significant when σ is comparable to n, and it also dictates the amount of main memory required to compress or decompress. We design an encoding scheme that requires $\sigma \log\log n + O(\sigma + \log^2 n)$ bits in the worst case, and typically less, while supporting encoding and decoding of symbols in $O(\log\log n)$ time. We show that our technique reduces the storage size of the model of state-of-the-art techniques to around 15% in various real-life sequences over large alphabets, while still offering reasonable compression/decompression times.

1 Introduction

Huffman coding [11] is a landmark in data compression, and used as a backend in a huge number of compression algorithms and formats, including the popular PKZIP, JPEG and MP3. Its speed and simplicity, as well as its ability to provide random access, makes it attractive in many scenarios, including compressed text databases and compressed data structures, which must operate without decompressing the data.

A Huffman encoder must maintain a data structure that, given a source symbol, returns its code. The decoder must, inversely, return the source symbol corresponding to a code. In both cases, the size of the data structure is proportional to that of the source alphabet, σ, and must be maintained in main memory while compressing or decompressing. In semi-static compression this mapping, in addition, must be stored with the compressed file, adding to the redundancy.

While in several cases σ is negligible compared to n, the length of the sequence to compress, there are various applications where storing the model is problematic.

[*]This work was partially supported by Xunta de Galicia (co-funded with FEDER), ref. 2010/17 and CN 2012/211 (strategic group), and by the Spanish MICINN ref. AP2010-6038 (FPU Program) for A.O., and Fondecyt Grant 1-110066, Chile for G.N.

A paradigmatic example is the word-wise compression of natural language texts [13], where Huffman codes not only perform very competitively but also benefit direct access [22], text searching [14], and indexing [2]. Other scenarios where large alphabets arise are the compression of Oriental languages and general numeric sequences. In other cases one wishes to apply Huffman coding to short texts (and thus even a small alphabet becomes relatively large), for example for compression boosting [8] or in interactive communications or adaptive compression [3].

Note that, while the plain and the compressed sequences may be read/written from/to disk, the model must be represented in memory for efficient compression and decompression. Therefore a large model cannot be handled by resorting to disk. This is also the case in mobile devices, where the amount of memory is limited and representing a large model may be prohibitive.

A naive representation of the encoding or decoding model requires as much as $O(\sigma \log n)$ bits, since $O(\log n)$ is the maximum possible code length. A more careful implementation, using canonical codes, requires $\sigma \log \sigma + O(\sigma)$ bits (our logarithms are in base 2), to store the Huffman tree shape and the permutation of the symbols at its leaves. The permutation can be avoided, and the space of the model reduced to $O(\sigma)$ bits, if we give up on the optimality and use optimal alphabetic codes (which cannot reorder the leaves; they are also called Hu-Tucker codes [12]). These codes, however, can increase the redundancy by up to n bits. Several other suboptimal codes that allow one to reduce the model size have been studied, see Gagie et al. [9] for recent results and references.

In this paper we show how the *optimal* model can be represented using $\sigma \log \log n + O(\sigma + \log^2 n)$ bits in the worst case (and usually better in practice) while encoding and decoding in time $O(\log \log n)$. The key idea is to alphabetically sort the source symbols that share the same depth in the canonical Huffman tree, so that $O(\log n)$ increasing runs are formed in the permutation, and then use a representation of permutations that takes advantage of those runs [1]. We implement our representation and experimentally show its impact on the model size in various real-life applications. Concretely, we decrease the storage space of a state-of-the art representation to around 12% (for compression) and 17% (for decompression), while still offering reasonable compression and decompression time. This opens the door to applying Huffman compression and decompression over large-alphabet sequences on severely limited memories.

2 Basic Concepts

2.1 Huffman Coding

Given a sequence $M[1, n]$ with vocabulary $\Sigma = [1, \sigma]$, the Huffman algorithm [11] produces an optimal variable-length encoding so that *(1)* it is prefix-free, that is, no code is prefix of another; *(2)* the size of the compressed sequence is minimized. Huffman [11] showed how to build a so-called Huffman tree to obtain a minimum redundancy encoding. The tree leaves will contain the symbols, whose codes are obtained by following the tree path from the root to their leaves. The optimality of such a set of codes depends only on the length of each code, not on its numerical value. Based on this idea, Schwartz et al. [21] proposed a different way to assign codes to symbols so that all the codes of the same length are consecutive numbers. This yields several interesting benefits like faster decoding and others [21, 20].

2.2 Wavelet Trees

Huffman coding is also widely used in compact data structures, being wavelet trees [10] a prominent example. A wavelet tree (WT) is a versatile data structure used to represent a sequence $M[1,n]$ over an alphabet $\Sigma[1,\sigma]$. It is a perfect balanced binary tree with σ leaves where each node handles a subset of symbols. At the root node, the building algorithm divides the vocabulary Σ into two halves (Σ_1 and Σ_2) and stores a bitmap $B[1,n]$ so that $B[i] = 0$ iff $M[i] \in \Sigma_1$, $B[i] = 0$ otherwise. The process is repeated recursively so that the left sub-tree will handle Σ_1 (and represent the subsequence of M formed by the symbols in Σ_1) and the right sub-tree will handle Σ_2. As tree has height $\lceil \log \sigma \rceil$ and there are n bits per level, the total length of all the bitmaps is $n\lceil \log \sigma \rceil$ bits. Storing the tree pointers will require $O(\sigma \log n)$ further bits, so the total size of a perfect balanced WT is $n \log \sigma + O(n + \sigma \log n)$ bits. This can be reduced to $n \log \sigma + O(n)$ if we use a levelwise representation where all bitmaps of the same level are concatenated [6].

Building a balanced WT is equivalent to using a fixed-length encoding of $\lceil \log \sigma \rceil$ bits per symbol. Instead, by giving the WT the shape of the Huffman tree, the total number of bits stored is exactly the output size of the Huffman compressor [10, 17]. In this Huffman-shaped WT (HWT), the path from the root to each leaf spells out the code assigned to the corresponding source symbol. The total size of the HWT is thus upper bounded by $n(1 + H_0(M)) + O(\sigma \log n)$ bits, being $H_0(M)$ the zero-order entropy of M. The term $O(\sigma \log n)$ accounts for the pointers and for the permutation of symbols induced by the code.

To access $M[i]$ we examine the root bitmap at position i. If $B[i] = 0$ we descend to the left sub-tree, output 0, and update $i \leftarrow rank_0(B, i)$. Here $rank_b(B, i)$ is the number of occurrences of the bit b in $B[1, i]$. Otherwise we descend to the right sub-tree, output 1, and update $i \leftarrow rank_1(B, i)$. When we reach a leaf we have output the Huffman code of $M[i]$. From that code, using the Huffman decoder, we obtain the source symbol contained in $M[i]$.

The WT can also compute two powerful operations on sequences. The first is $rank_s(M, i)$, the number of times symbol s appears in $M[1, i]$. We start by obtaining the Huffman code $c[1, l]$ associated to symbol s and setting $l \leftarrow 1$. At the root bitmap, if $c[l] = 0$ we descend to the left sub-tree with $i \leftarrow rank_0(B, i)$. Otherwise we descend to the right sub-tree with $i \leftarrow rank_1(B, i)$. In both cases we increment l. When we reach a leaf the answer is the value of i.

The other operation is $select_s(M, j)$, the position of the j-th occurrence of symbol s in M. This time we start at the leaf corresponding to s and at position j. If the leaf is a left child of its parent, whose bitmap is B, we update $j \leftarrow select_0(B, j)$. Else we update $j \leftarrow select_1(B, j)$. When we arrive at the root, the value of j is the answer.

In order to solve those operations, we must be able to solve *rank* and *select* on the bitmaps. Depending on the technique used we can obtain different space/time trade-offs for wavelet trees. Munro [15] and Clark [5] proposed a technique to solve *rank* and *select* in constant time by adding a sublinear-size overhead to the bitmap, which has since then be reduced to $O(n/\log^2 n)$ and less [18]. Then the space of the WT becomes at most $n(1+H_0(M))(1+o(1))+O(\sigma\log n) = nH_0(M)+O(n+\sigma\log n)$. By rebalancing deep leaves, the space figure is maintained and the worst-case time of the operations is limited to $O(\log\sigma)$, whereas their average time is reduced to $O(1+H_0(M))$ if positions in M are probed uniformly at random.

Another bitmap representation, by Raman, Raman, and Rao (RRR) [19], retains constant time for both operations but reencodes the bitmap $B[1,n]$ in total space $nH_0(B)+o(n)$ [10]. Using this representation, even on balanced WTs the space reaches zero-order entropy as well. In practice, combining HWTs and RRR achieves the bests results [6].

2.3 Compressed Permutations

The wavelet tree can be used for a wealth of purposes [17]. A recent application [1] is to compress and efficiently access a *permutation* $\pi[1..p]$. Two interesting operations defined over π are accessing any $\pi(i)$ and accessing its inverse $\pi^{-1}(i)$. While general permutations require $\log p! = p\log p - O(p)$ to represent π, Barbay and Navarro [1] show how to use less space on permutations that can be decomposed into a few *increasing runs*. An increasing run is a maximal interval $[i,j]$ where π is increasing. So, being $r_1,\ldots,r_\rho = p$ the endpoints of the increasing runs, we define the vector $Runs = (r_1, r_2-r_1,\ldots,r_\rho-r_{\rho-1})$ of the run sizes. For example, the permutation $(7,2,6,8,4,1,5,3)$ has $\rho=5$ increasing runs and $Runs = \{1,3,1,2,1\}$.

Let $P[1,p]$ be the sequence on alphabet $[1,\rho]$ obtained by replacing, in π^{-1}, every number belonging to the k-th run in π by symbol k. Then a HWT on P requires $pH(Runs)+O(p+\rho\log p)$ bits, where $H(Runs) = H_0(P) \le \log\rho$ is the entropy of the distribution of run sizes. If accessing $P[i]$ we arrive at position j inside the leaf corresponding to run k, then we have that $\pi^{-1}(i) = r_{k-1}+j$. Instead, if position i corresponds to offset j in run k, then we run the process of *select* on the WT, starting at offset j of the leaf of run k, and arrive at position $\pi(i)$ in the root. Hence $\pi(i)$ and $\pi^{-1}(i)$ are computed in time $O(\log\rho)$.

3 A Compact Model Representation

In this section we describe how to represent the Huffman model in compact form. Given a sequence $M[1..n]$ on alphabet $[1,\sigma]$, we first obtain a canonical Huffman code for the symbols.

To obtain a compressible permutation from the canonical Huffman tree, we read the Huffman tree leaves from left to right and store the corresponding symbols in a vector $\pi[1,\sigma]$. π is a permutation since the encoded symbols are distinct numbers in $[1,\sigma]$. For instance, applying this process to the Huffman tree of Figure 1, we

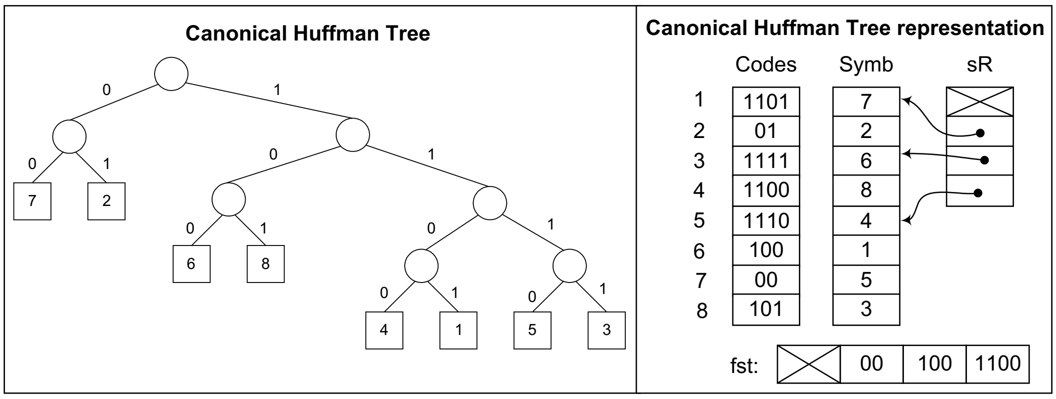

Figure 1: On the left, a Canonical Huffman Tree. On the right, a classical representation using $\sigma \log n + \sigma \log \sigma + O(\log^2 n)$ bits. Here $fst[i]$ contains the first code of length i and $sR[i]$ indicates where the codes of length i start in the vector of symbols.

obtain the permutation $\pi = (7, 2, 6, 8, 4, 1, 5, 3)$, which has $\rho = 5$ increasing runs. We wish to use the technique of Barbay and Navarro [1] to compress and access π. That technique will work better when the number of increasing runs is smaller. To reduce the number of runs, we reassign codes to symbols while keeping the encoding optimal and prefix-free as follows: For each Huffman tree level (i.e., for each distinct Huffman tree code length), we sort the tree leaves so that smaller codewords are assigned to smaller symbols. In our example, we obtain $\pi = (2, 7, 6, 8, 1, 3, 4, 5)$, with $\rho = 3$ increasing runs (see Figure 2).

Since the maximum Huffman code length is $O(\log n)$, the Huffman tree has at most $O(\log n)$ distinct levels, thus the number of increasing runs in π is $O(\log n)$ (one run per level). Now the technique of Barbay and Navarro [1] compresses permutation $\pi[1, \sigma]$, with run sizes $Runs$, into $\sigma H(Runs) + O(\sigma + \rho \log n) + O(\log^2 n) \leq \sigma \log \log n + O(\sigma + \log^2 n)$ bits. We also include in the $O(\log^2 n)$ term the space to store two vectors fst and sR: $fst[i]$ contains the first Huffman code of each length, while $sR[i]$ contains the position in π where each run begins, for $i \in [1, O(\log n)]$. Both vectors are also used by other canonical Huffman tree representations.

To obtain a code from a symbol s, we first apply $\pi^{-1}(s)$. This returns the position pos in π that contains s, and binary search in sR obtains the length l of the code. Then we return the code $c = fst[l] + pos - sR[l]$. That is, if we know the offset $(pos - sR[l])$ inside a run and the first code of that run $(fst[l])$, we know the code associated to s since all codes inside a run are consecutive. For instance, to obtain the code associated with symbol s_8 from the Huffman tree of Figure 2, we apply $\pi^{-1}(8)$ and obtain $pos = 4$. Binary searching sR we find that the code has length $l = 3$. Then the code of symbol s_8 is $fst[l] + pos - sR[l] = 100_2 + 4 - 3 = 4 + 4 - 3 = 5$. The time for this encoding operation is $O(\log \log n)$, both for computing π^{-1} (with a downward traversal on the wavelet tree used to represent the inverse of π [1]) and for binary searching sR.

To obtain the symbol corresponding to a code c of length l we do the inverse process. We compute $\pi[pos]$, for $pos = c - fst[l] + sR[l]$. For instance, to obtain the

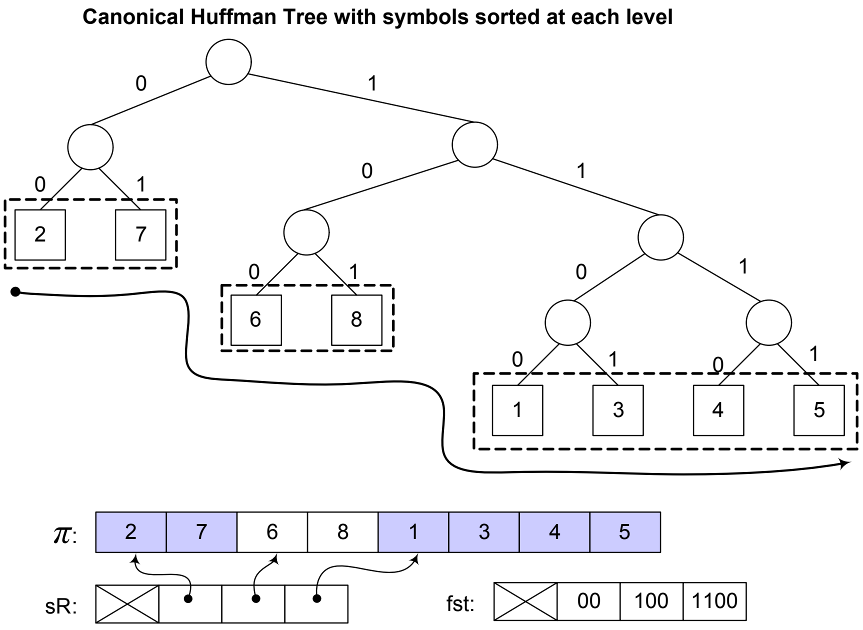

Figure 2: A canonical Huffman tree with the symbols at each level sorted in increasing order. Our representation uses *fst* and *sR*, plus a compressible permutation for π.

Collection	Length (n)	Alph. (σ)	Entropy (H_0)	Depth	Runs (ρ)	$H(Runs)$
EsWiki	200,000,000	1,634,145	11.12	28	25	2.53
EsInv	200,000,000	1,590,398	12.47	28	28	3.12
Indo	120,000,000	3,715,187	16.29	27	19	2.51

Table 1: Main statistics of the collections used.

symbol associated to code $c = 101_2$ of length $l = 3$, we compute $pos = 101_2 - fst[3] + sR[3] = 101_2 - 100_2 + 3 = 4$, and then return $\pi[4] = s_8$. The time complexity is also $O(\log \log n)$, dominated by the time to compute $\pi[pos]$ (recall that this is done via an upward traversal on the wavelet tree used to represent the inverse of π [1]).

4 Experimental results

We used an isolated Intel®Xeon®-E5520@2.26GHz with 72 GB DDR3@800 MHz RAM. It ran Ubuntu 9.10 (kernel 2.6.31-19-server), using `gcc` version 4.4.1 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 [13, 22, 14, 2]. *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 [22]. Finally, *Indo* is the concatenation of the adjacency lists of Web graph `Indochina-2004`

Collection	Naive (σw)	Engineered (σd)	Canonical ($\sigma \log \sigma$)	Ours ($\sigma H(Runs)$)
EsWiki	6.23 MB	5.45 MB	4.09 MB	0.49 MB
EsInv	6.07 MB	5.30 MB	3.90 MB	0.59 MB
Indo	14.17 MB	11.95 MB	9.74 MB	1.11 MB

Table 2: Estimated size of various model representations.

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 [7]. For the three sequences, we use a prefix to speed up experiments.

Table 1 gives various statistics on the collections. Apart from n and σ, we give the zero-order entropy of the sequence (in bits per symbol), the depth of the Huffman tree, the number of runs after we reorder the symbols of each level of the canonical Huffman tree (i.e., the number of different levels in the wavelet tree), and the entropy of the sequence of run sizes, which is significantly lower than $\log \rho$.

Table 2 shows how the sizes of different representations of the model can be estimated using these numbers. The first column gives σw, the size of a naive model representation using computer words of $w = 32$ bits. The second gives σd, where d is the Huffman tree depth, and corresponds to a more engineered representation where we use only the (maximum) number of bits required. In these two, more structures are needed for decoding but we ignore them. The third column gives $\sigma \log \sigma$, 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 $\sigma H(Runs)$, 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 known representations to reduce the space, and that we can expect space reductions of around one order of magnitude.

Next we evaluate the performance of our technique to compress and decompress the whole sequences. We measure the space required by the model and the compression/decompression time. We compare our technique (PERM) with a careful implementation (TABLE) using the "engineered" table for compression (σd bits) and the "canonical" table for decompression ($\sigma \log \sigma$ bits). The input and output sequences reside on disk. For decompression, we use in both cases canonical Huffman codes with tables *fst* and sR: we iteratively probe the next l bits from the compressed sequence, where l is the next available tree depth. If the relative numeric code resulting from reading l bits exceeds the number of nodes at this level, we probe the next level, and so on until finding the right length [21, 20].

For our permutation we implement the technique of Barbay and Navarro almost verbatim [1], using a Hu-Tucker tree with pointers to the children plus pointers to the leaves (note that this tree has just $O(\rho) = O(\log n)$ nodes). For the bitmaps we use a fast version posing 37.5% space overhead on top of the bitmaps [16].

We can see in Figure 3 that our compressed representations take just around 12% of the space of the table implementation for compression, and 17% for decompres-

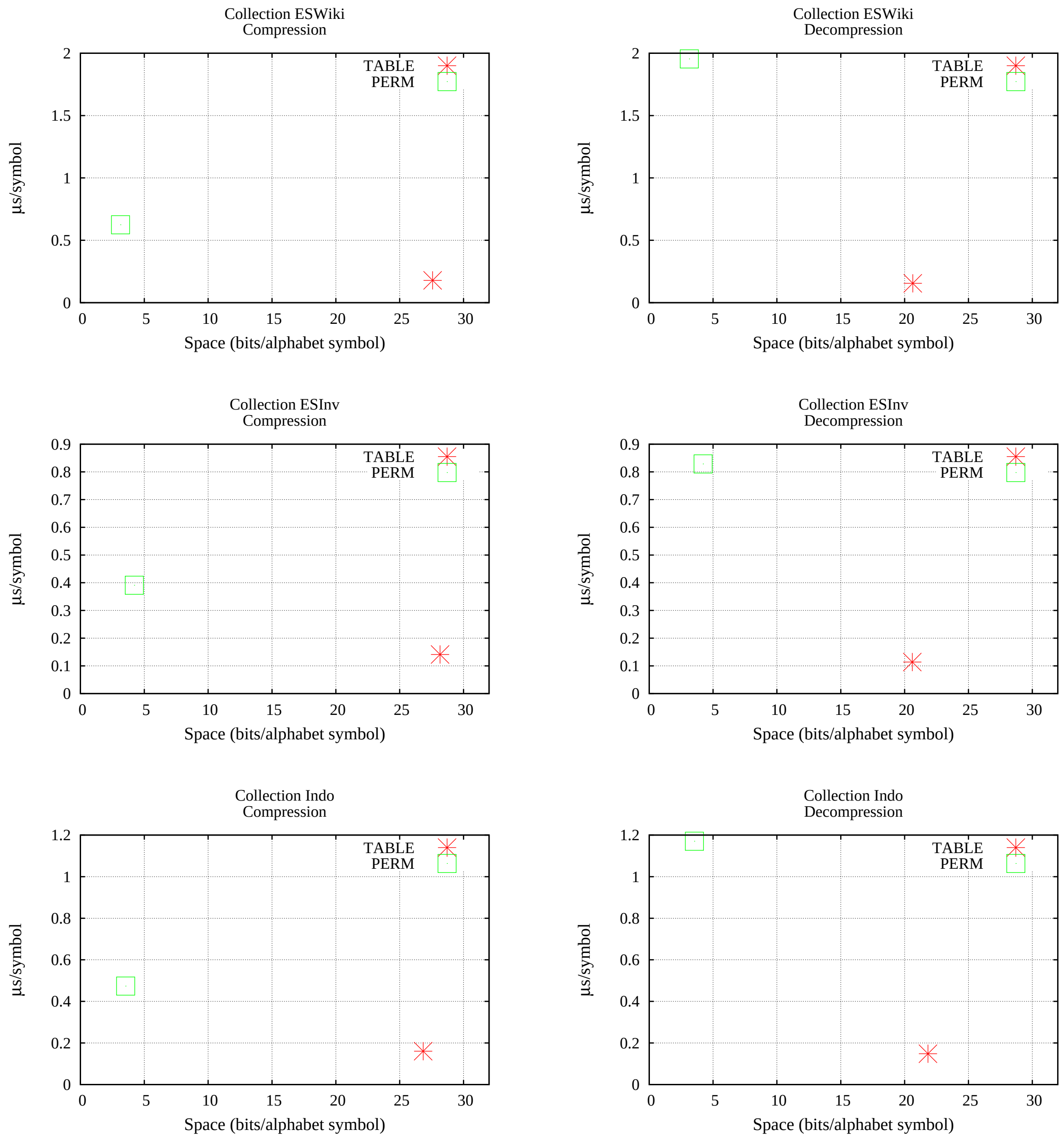

Figure 3: Model size versus compression/decompression time for the naive representation (TABLE) and ours (PERM).

sion. We can also see that our structures make the process slower, by 3-4 times in compression and around 7-13 times in decompression. The gap could be narrowed by improving the performance of the operations on the bitmaps (especially *select*). Yet, we can still compress and decompress at the reasonable rates of around 2 MB/sec and 0.5–1 MB/sec, respectively.

5 Conclusions

We have presented a technique to sharply reduce the size of the model representation in Huffman coding, to about 12% (at compression) and 17% (at decompression) of its original size, processing the data at 2 MB/sec (at compression) and 0.5–1 MB/sec (at decompression). This result is relevant in various applications where large alphabets must be handled, such as when compressing natural language texts using word-based models. The drastic reduction in the model space enables compression and decompression within very limited main memory, for example on mobile devices.

An interesting challenge, precisely considering a computationally strong server sending compressed data to a weaker client (e.g., a mobile device), is whether a dynamic code can be maintained with the minimum possible burden for the client. This asymmetry has been studied on the word-based model using suboptimal codes [4], showing that the server can maintain the statistics, update the code, and inform the receiver only of the code changes, trying to minimize them. Is it possible for a server to maintain a dynamic canonical Huffman code and inform the receiver of the changes, so that it can now handle a dynamic compressed representation of the permutation?

References

[1] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. 26th STACS*, pages 111–122, 2009.

[2] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, 2012. To appear.

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

[4] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Dynamic lightweight text compression. *ACM Transactions on Information Systems*, 28(3):article 10, 2010.

[5] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[6] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th SPIRE*, LNCS 5280, pages 176–187, 2008.

[7] F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Transactions on the Web*, 4(4):article 16, 2010.

[8] 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.

[9] T. Gagie, G. Navarro, and Y. Nekrich. Fast and compact prefix codes. In *Proc. 36th SOFSEM*, LNCS 5901, pages 419–427, 2010.

[10] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 13th SODA*, pages 841–850, 2003.

[11] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. I.R.E.*, volume 40, pages 1098–1101, 1952.

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

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

[14] 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.

[15] I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.

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

[17] G. Navarro. Wavelet trees for all. In *Proc. 23rd CPM*, LNCS 7354, pages 2–26, 2012.

[18] M. Pătraşcu. Succincter. In *Proc. 49th FOCS*, pages 305–313, 2008.

[19] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 12th SODA*, pages 233–242, 2002.

[20] D. Salomon. *Data Compression.* Springer, 2007.

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

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