

Efficient and Compact Representations of Some Non-Canonical Prefix-Free Codes^{*}

Antonio Fariña¹, Travis Gagie², Giovanni Manzini³,
Gonzalo Navarro⁴, and Alberto Ordóñez⁵

¹ Database Laboratory, University of A Coruña, Spain

² Helsinki Institute for Information Technology (HIIT)

Department of Computer Science, University of Helsinki, Finland

³ Department of Computer Science, University of Eastern Piedmont, Italy

⁴ Department of Computer Science, University of Chile, Chile

⁵ Yoop SL, Spain

Abstract. For many kinds of prefix-free codes there are efficient and compact alternatives to the traditional tree-based representation. Since these put the codes into canonical form, however, they can only be used when we can choose the order in which codewords are assigned to characters. In this paper we first show how, given a probability distribution over an alphabet of σ characters, we can store a nearly optimal alphabetic prefix-free code in $o(\sigma)$ bits such that we can encode and decode any character in constant time. We then consider a kind of code introduced recently to reduce the space usage of wavelet matrices (Claude, Navarro, and Ordóñez, *Information Systems*, 2015). They showed how to build an optimal prefix-free code such that the codewords' lengths are non-decreasing when they are arranged such that their reverses are in lexicographic order. We show how to store such a code in $\mathcal{O}(\sigma \log L + 2^{\epsilon L})$ bits, where L is the maximum codeword length and ϵ is any positive constant, such that we can encode and decode any character in constant time under reasonable assumptions. Otherwise, we can always encode and decode a codeword of ℓ bits in time $\mathcal{O}(\ell)$ using $\mathcal{O}(\sigma \log L)$ bits of space.

1 Introduction

Binary prefix-free codes can be represented as binary trees whose leaves are labelled with the characters of the source alphabet, so that the ancestor at

^{*} Funded in part by European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 690941 (project BIRDS). The first author was supported by: MINECO (PGE and FEDER) grants TIN2013-47090-C3-3-P and TIN2015-69951-R; MINECO and CDTI grant ITC-20151305; ICT COST Action IC1302; and Xunta de Galicia (co-founded with FEDER) grant GRC2013/053. The second author was supported by Academy of Finland grant 268324. The fourth author was supported by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile.

depth d of the leaf labelled x is a left child if the d th bit of the codeword for x is a 0, and a right child if it is a 1. To encode a character, we start at the root and descend to the leaf labelled with that character, at each step writing a 0 if we go left and a 1 if we go right. To decode an encoded string, we start at the root and descend according to the bits of the encoding until we reach a leaf, at each step going left if the next bit is a 0 and right if it is a 1. Then we output the character associated with the leaf and return to the root to continue decoding. Therefore, a codeword of length ℓ is encoded/decoded in time $\mathcal{O}(\ell)$. This all generalizes to larger code alphabets, but for simplicity we consider only binary codes in this paper.

There are, however, faster and smaller representations of many kinds of prefix-free codes. If we can choose the order in which codewords are assigned to characters then, by the Kraft Inequality [8], we can put any prefix-free code into canonical form [13] — i.e., such that the codewords' lexicographic order is the same as their order by length, with ties broken by the lexicographic order of their characters — without increasing any codeword's length. If we store the first codeword of each length as a binary number then, given a codeword's length and its rank among the codewords of that length, we can compute the codeword via a simple addition. Given a string prefixed by a codeword, we can compute that codeword's length and its rank among codewords of that length via a predecessor search. If the alphabet consists of σ characters and the maximum codeword length is L , then we can build an $\mathcal{O}(\sigma \log L)$ -bit data structure with $\mathcal{O}(\log L)$ query time that, given a character, returns its codeword's length and rank among codewords of that length, or vice versa. If L is at most a constant times the size of a machine word (which it is when we are considering, e.g., Huffman codes for strings in the RAM model) then in theory we can make the predecessor search and the data structure's queries constant-time, meaning we can encode and decode in constant time [5].

There are applications for which there are restrictions on the codewords' order, however. For example, in alphabetic codes the lexicographic order of the codewords must be the same as that of the characters. Such codes are useful when we want to be able to sort encoded strings without decoding them (because the lexicographic order of two encodings is always the same as that of the encoded strings) or when we are using data structures that represent point sets as sequences of coordinates [10], for example. Interestingly, since the mapping between symbols and leaves is fixed, alphabetic codes need only to store the tree topology, which can be represented more succinctly than optimal prefix-free codes, in $2\sigma + o(\sigma)$ bits [9], so that encoding and decoding can still be done in time $\mathcal{O}(\ell)$. There are no, however, equivalents to the faster encoding/decoding methods used on canonical codes [5].

In Section 2 we show how, given a probability distribution over the alphabet, we can store a nearly optimal alphabetic prefix-free code in $o(\sigma)$ bits such that we can encode and decode any character in constant time. We note that we can still use our construction even if the codewords must be assigned to the

characters according to some non-trivial permutation of the alphabet, but then we must store that permutation such that we can evaluate and invert it quickly.

In Section 3 we consider another kind of non-canonical prefix-free code, which Claude, Navarro, and Ordóñez [1] introduced recently to reduce the space usage of their wavelet matrices. (Wavelet matrices are alternatives to wavelet trees [6, 10] that are more space efficient when the alphabet is large.) They showed how to build an optimal prefix-free code such that the codewords' lengths are non-decreasing when they are arranged such that their reverses are in lexicographic order. They represent the code in $\mathcal{O}(\sigma L)$ bits, and encode and decode a codeword of length ℓ in time $\mathcal{O}(\ell)$. We show how to store such a code in $\mathcal{O}(\sigma \log L)$ bits, and still encode and decode any character in $\mathcal{O}(\ell)$ time. We also show that, by using $\mathcal{O}(\sigma \log L + 2^{\epsilon L})$ bits, where ϵ is any positive constant, we can encode and decode any character in constant time when L is at most a constant times the size of a machine word. Our first variant is simple enough to be implementable. We show experimentally that it uses 23–30 times less space than a classical implementation, at the price of being 10–21 times slower at encoding and 11–30 at decoding.

2 Alphabetic Codes

Evans and Kirkpatrick [2] showed how, given a binary tree on n leaves, we can build a new binary tree of height at most $\lceil \lg n \rceil + 1$ on the same leaves in the same left-to-right order, such that the depth of each leaf in the new tree is at most 1 greater than its depth in the original tree. We can use their result to restrict the maximum codeword length of an optimal alphabetic prefix code, for an alphabet of σ characters, to be at most $\lg \sigma + \sqrt{\lg \sigma} + 3$, while forcing its expected codeword length to increase by at most a factor of $1 + \mathcal{O}(1/\sqrt{\log \sigma})$. To do so, we build the tree T_{opt} for an optimal alphabetic prefix code and then rebuild, according to Evans and Kirkpatrick's construction, each subtree rooted at depth $\lceil \sqrt{\lg \sigma} \rceil$. The resulting tree, T_{lim} , has height at most $\lceil \sqrt{\lg \sigma} \rceil + \lceil \lg \sigma \rceil + 1$ and any leaf whose depth increases was already at depth at least $\lceil \sqrt{\lg \sigma} \rceil$.

There are better ways to build a tree T_{lim} with such a height limit. Itai [7] and Wessner [14] independently showed how, given a probability distribution over an alphabet of σ characters, we can build an alphabetic prefix code T_{lim} that has maximum codeword length at most $\lg \sigma + \sqrt{\lg \sigma} + 3$ and is optimal among all such codes. Our construction in the previous paragraph, even if not optimal, shows that the expected codeword length of T_{lim} is at most $1 + \mathcal{O}(1/\sqrt{\log \sigma})$ times times that of an optimal code with no length restriction.

Further, let us take T_{lim} and completely balance each subtree rooted at depth $\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil$. The height remains at most $\lg \sigma + \sqrt{\lg \sigma} + 3$ and any leaf whose depth increases was already at depth at least $\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil$, so the expected codeword length increases by at most a factor of

$$\frac{\lg \sigma + \sqrt{\lg \sigma} + 3}{\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil} = 1 + \mathcal{O}\left(\frac{1}{\sqrt{\log \sigma}}\right).$$

Let T_{bal} be the resulting tree. Since the expected codeword length of T_{lim} is in turn at most a factor of $1 + \mathcal{O}(1/\sqrt{\log n})$ larger than that of T_{opt} , the expected codeword length of T_{bal} is also at most a factor of $(1 + \mathcal{O}(1/\sqrt{\log n}))^2 = 1 + \mathcal{O}(1/\sqrt{\log n})$ larger than the optimal. T_{bal} then describes our suboptimal code.

To represent T_{bal} , we store a bitvector $B[1..\sigma]$ in which $B[i] = 1$ if and only if the codeword for the i th character in the alphabet has length at most $\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil$, or if the i th leaf in T is the leftmost leaf in a subtree rooted at depth $\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil$. With Pătraşcu's implementation [12] for B this takes a total of $\mathcal{O}(2^{\lg \sigma - \sqrt{\lg \sigma}} \log \sigma + \sigma / \log^c \sigma) = \mathcal{O}(\sigma / \log^c \sigma)$ bits for any constant c , and allows us to perform in constant time $\mathcal{O}(c)$ the following operations on B : (1) access, that is, inspecting any $B[i]$; (2) rank, that is, $rank(B, i)$ counts the number of 1s in any prefix $B[1..i]$; and select, that is, $select(B, j)$ is the position of the j th 1 in B , for any j .

Let us for simplicity assume that the alphabet is $[1..\sigma]$. For encoding in constant time we store an array $S[1..2^{\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil}]$, which stores the explicit code assigned to the leaves of T_{bal} where $B[i] = 1$, in the same order of B . That is, if $B[i] = 1$, then the code assigned to the character i is stored at $S[rank(B, i)]$, using $\lg \sigma + \sqrt{\lg \sigma} + 3 = \mathcal{O}(\log \sigma)$ bits. Therefore S requires $\mathcal{O}(2^{\lg \sigma - \sqrt{\lg \sigma}} \log \sigma) = o(\sigma / \log^c \sigma)$ bits of space, for any constant c . We can also store the length of the code within the same asymptotic space.

To encode the character i , we check whether $B[i] = 1$ and, if so, we simply look up the codeword in S as explained. If $B[i] = 0$, we find the preceding 1 at $i' = select(B, rank(B, i))$, which marks the leftmost leaf in the subtree rooted at depth $\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil$ that contains the i th leaf in T . Since the subtree is completely balanced, we can compute the code for the character i in constant time from that of the character i' : The size of the balanced subtree is $r = i'' - i'$, where $i'' = select(B, rank(B, i') + 1)$, and its height is $h = \lceil \lg r \rceil$. Then the first $2r - 2^h$ codewords are of the same length of the codeword for i' , and the last $2^h - r$ have one bit less. Thus, if $i - i' < 2r - 2^h$, the codeword for i is $S[rank(B, i')] + i - i'$, of the same length of that of i ; otherwise it is one bit shorter, $(S[rank(B, i')] + 2r - 2^h) / 2 + i - i' - (2r - 2^h) = S[rank(B, i')] / 2 + i - i' - (r - 2^{h-1})$.

To be able to decode quickly, we store an array $A[1..2^{\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil}]$ such that, for $1 \leq j \leq 2^{\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil}$, if the $\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil$ -bit binary representation of $j - 1$ is prefixed by the i th codeword, then $A[j]$ stores i and the length of that codeword. If, instead, the $\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil$ -bit binary representation of j is the path label to the root of a subtree of T_{bal} with size more than 1, then $A[j]$ stores the position i' in B of the leftmost leaf in that subtree (thus $B[i'] = 1$). Again, A takes $\mathcal{O}(2^{\lg \sigma - \sqrt{\lg \sigma}} \log \sigma) = o(\sigma / \log^c \sigma)$ bits, for any constant c .

Given a string prefixed by the i th codeword, we take the prefix of length $\lceil \lg \sigma - \sqrt{\lg \sigma} \rceil$ of that string (padding with 0s on the right if necessary), view it as the binary representation of a number j , and check $A[j]$. This either tells us immediately i and the length of the i th codeword, or tells us the position i' in B of the leftmost leaf in the subtree containing the desired leaf. In the latter case,

since the subtree is completely balanced, we can compute i in constant time: We find i' , r , and h as done for encoding. We then take the first h bits of the string (including the prefix we had already read, and padding with a 0 if necessary), and interpret it as the number j' . Then, if $d = j' - S[\text{rank}(B, i')] < 2r - 2^h$, it holds $i = i' + d$. Otherwise, the code is of length $h - 1$ and the decoded symbol is $i = i' + 2r - 2^h + \lfloor (d - (2r - 2^h))/2 \rfloor = i' + r - 2^{h-1} + \lfloor d/2 \rfloor$.

Theorem 1. *Given a probability distribution over an alphabet of σ characters, we can build an alphabetic prefix code whose expected codeword length is at most a factor of $1 + \mathcal{O}(1/\sqrt{\log \sigma})$ more than optimal and store it in $\mathcal{O}(\sigma/\log^c \sigma)$ bits, for any constant c , such that we can encode and decode any character in constant time $\mathcal{O}(c)$.*

3 Codes for Wavelet Matrices

As we mentioned in Section 1, in order to reduce the space usage of their wavelet matrices, Claude, Navarro, and Ordóñez [1] recently showed how to build an optimal prefix code such that the codewords' lengths are non-decreasing when they are arranged such that their reverses are in lexicographic order. Specifically, they first build a normal Huffman code and then use the Kraft Inequality to build another code with the same codeword lengths with the desired property. They store an $\mathcal{O}(\sigma L)$ -bit mapping between characters and their codewords, where again σ is the alphabet size and L is the maximum length of any codeword, which allows them to encode and decode codewords of length ℓ in time $\mathcal{O}(\ell)$. (In the wavelet matrices, they already spend $\mathcal{O}(\ell)$ time in the operations associated with encoding and decoding.)

Assume we are given a code produced by Claude et al.'s construction. We reassign the codewords of the same length such that the lexicographic order of the reversed codewords of that length is the same as that of their characters. This preserves the property that codeword lengths are non-decreasing with their reverse lexicographic order. The positive aspect of this reassignment is that all the information on the code can be represented in $\sigma \lg L$ bits as a sequence $D = d_1, \dots, d_\sigma$, where d_i is the depth of the leaf encoding character i in the code-tree T . We can then represent D using a wavelet tree [6], which uses $\mathcal{O}(\sigma \log L)$ bits and supports the following operations on D in time $\mathcal{O}(\log L)$: (1) access any $D[i]$, which gives the length ℓ of the codeword of character i ; (2) compute $r = \text{rank}_\ell(D, i)$, which gives the number of occurrences of ℓ in $D[1..i]$, which if $D[i] = \ell$ gives the position (in reverse lexicographic order) of the leaf representing character i among those of codeword length ℓ ; and (3) compute $i = \text{select}_\ell(D, r)$, which gives the position in D of the r th occurrence of ℓ , or which is the same, the character i corresponding to the r th codeword of length ℓ (in reverse lexicographic order).

If, instead of $\mathcal{O}(\log L)$ time, we wish to perform the operations in time $\mathcal{O}(\ell)$, where ℓ is the length of the codeword involved in the operation, we can simply give the wavelet tree of D the same shape of the tree T . We can even perform

the operations in time $\mathcal{O}(\log \ell)$ by using a wavelet tree shaped like the trie for the first σ codewords represented with Elias γ - or δ -codes [4, Observation 1]. The size stays $\mathcal{O}(\sigma \log L)$ if we use compressed bitmaps at the nodes [6, 10].

We are left with two subproblems. For decoding the first character encoded in a binary string, we need to find the length ℓ of the first codeword and the lexicographic rank r of its reverse among the reversed codewords of that length, since then we can decode $i = \text{select}_\ell(D, r)$. For encoding a character i , we find its length $\ell = D[i]$ and the lexicographic rank $r = \text{rank}_\ell(D, i)$ of its reverse among the reversed codewords of length ℓ , and then we must find the codeword given ℓ and r . We first present a solution that takes $\mathcal{O}(L \log \sigma) = \mathcal{O}(\sigma \log L)$ further bits⁶ and works in $\mathcal{O}(\ell)$ time. We then present a solution that takes $\mathcal{O}(2^{\varepsilon L})$ further bits and works in constant time.

Let T be the code-tree and, for each depth d between 0 and L , let $\text{nodes}(d)$ be the total number of nodes at depth d in T and let $\text{leaves}(d)$ be the number of leaves at depth d . Let v be a node other than the root, let u be v 's parent, let r_v be the lexicographic rank (counting from 1) of v 's reversed path label among all the reversed path labels of nodes at v 's depth, and let r_u be defined analogously for u . Notice that since T is optimal it is strictly binary, so half the nodes at each positive depth are left children and half are right children. Moreover, the reversed path labels of all the left children at any depth are lexicographically less than the reversed path labels of all the right children at the same depth (or, indeed, at any depth). Finally, the reversed path labels of all the leaves at any depth are lexicographically less than the reversed path labels of all the internal nodes at that depth. It follows that

- v is u 's left child if and only if $r_v \leq \text{nodes}(\text{depth}(v))/2$,
- if v is u 's left child then $r_v = r_u - \text{leaves}(\text{depth}(u))$,
- if v is u 's right child then $r_v = r_u - \text{leaves}(\text{depth}(u)) + \text{nodes}(\text{depth}(v))/2$.

Of course, by rearranging terms we can also compute r_u in terms of r_v .

Suppose we store $\text{nodes}(d)$ and $\text{leaves}(d)$ for d between 0 and L . With the three observations above, given a codeword of length ℓ , we can start at the root and in $\mathcal{O}(\ell)$ time descend in T until we reach the leaf v whose path label is that codeword, then return its depth ℓ and the lexicographic rank $r = r_v$ of its reverse path label among all the reversed path labels of nodes at that depth.⁷ Then we compute i from ℓ and r as described, in further $\mathcal{O}(\log \ell)$ time. For encoding i , we obtain as explained its length ℓ and the rank $r = r_v$ of its reversed codeword among the reversed codewords of that length. Then we use the formulas to walk up towards the root, finding in each step the rank r_u of the parent u of v , and determining if v is a left or right child of u . This yields the ℓ bits of the codeword of i in reverse order (0 when v is a left child of u and 1 otherwise), in overall time $\mathcal{O}(\ell)$. This completes our first solution, which we evaluate experimentally in Section 4.

⁶ Since the code tree has height L and σ leaves, it follows that $L < \sigma$.

⁷ This descent is conceptual; we do not have a concrete node v at each level, but we do know r_v .

Theorem 2. *Suppose we are given an optimal prefix code in which the codewords' lengths are non-decreasing when they are arranged such that their reverses are in lexicographic order. We can store such a code in $\mathcal{O}(\sigma \log L)$ bits — possibly after swapping characters' codewords of the same length — where L is the maximum codeword length, such that we can encode and decode any character in $\mathcal{O}(\ell)$ time, where ℓ is the corresponding codeword length.*

If we want to speed up descents, we can build a table that takes as arguments a depth and several bits, and returns the difference between r_u and r_v for any node u at that depth and its descendant v reached by following edges corresponding to those bits. Notice that this difference depends only on the bits and the numbers of nodes and leaves at the intervening levels. If the table accepts t bits as arguments at once, then it takes $L2^t \log \sigma$ bits and we can descend in $\mathcal{O}(L/t)$ time. Setting $t = \epsilon L/2$, and since $L \geq \lg \sigma$, we use $\mathcal{O}(2^{\epsilon L})$ space and descend from the root to any leaf in constant time.

Speeding up ascents is slightly more challenging. Consider all the path labels of a particular length that end with a particular suffix of length t : the lexicographic ranks of their reverses form a consecutive interval. Therefore, we can partition the nodes at any level by their r values, such that knowing which part a node's r value falls into tells us the last t bits of that node's path label, and the difference between that node's r value and the r value of its ancestor at depth t less. For each depth, we store the first r value in each interval in a predecessor data structure, implemented as a trie with degree $\sigma^{\epsilon/3}$; since there are at most 2^t intervals in the partition for each depth and $L \geq \lg \sigma$, setting $t = \epsilon L/2$ again we use a total of $\mathcal{O}(L2^{\epsilon L/2} \sigma^{\epsilon/3} \log \sigma) \subset \mathcal{O}(2^{\epsilon L})$ bits and ascend from any leaf to the root in constant time.

Finally, the operations on the wavelet tree can be made constant-time by using a balanced multiary variant [3].

Theorem 3. *Suppose we are given an optimal prefix code in which the codewords' lengths are non-decreasing when they are arranged such that their reverses are in lexicographic order. Let L be the maximum codeword length, so that it is at most a constant times the size of the machine word. Then we can store such a code in $\mathcal{O}(\sigma \log L + 2^{\epsilon L})$ bits — possibly after swapping characters' codewords of the same length — where ϵ is any positive constant, such that we can encode and decode any character in constant time.*

4 Experiments

We have run experiments to compare the solution of Theorem 2 (referred to as WMM in the sequel, for Wavelet Matrix Model) with the only previous encoding, that is, the one used by Claude et al. [1] (denoted by TABLE). Note that our codes are not canonical, so other solutions [5] do not apply.

Claude et al. [1] use for encoding a single table of σL bits storing the code of each symbol, and thus they easily encode in constant time. For decoding, they have tables separated by codeword length ℓ . In each such table, they store

Collection	Length (n)	Alphabet size (σ)	Entropy ($\mathcal{H}(P)$)	max code length(L)	Entropy of level entries ($\mathcal{H}_0(D)$)
EsWiki	200,000,000	1,634,145	11.12	28	2.24
EsInv	300,000,000	1,005,702	5.88	28	2.60
Indo	120,000,000	3,715,187	16.29	27	2.51

Table 1. Main statistics of the texts used.

the codewords of that length and the associated character, sorted by codeword. This requires $\sigma(L + \lg \sigma)$ further bits, and permits decoding binary searching the codeword found in the wavelet matrix. Since there are at most 2^ℓ codewords of length ℓ , the binary search takes time $\mathcal{O}(\ell)$.

For the sequence D used in our WMM, we use binary Huffman-shaped wavelet trees with plain bitmaps. The structures for supporting rank/select efficiently require 37.5% space overhead, so the total space is $1.37 \sigma \mathcal{H}_0(D)$, where $\mathcal{H}_0(D) \leq \lg L$ is the per-symbol zero-order entropy of the sequence D . We also add a small index to speed up select queries [11] (that is, decoding), which can be parameterized with a sampling value that we set to $\{16, 32, 64, 128\}$. Finally, we store the values leaves and nodes, which add an insignificant L^2 bits in total.

We used a prefix of three datasets in <http://lbd.udc.es/research/ECRPC>. The first one, **EsWiki**, contains a sequence of word identifiers generated by using the Snowball algorithm to apply stemming to the Spanish Wikipedia. The second one, **EsInv**, contains a concatenation of differentially encoded inverted lists extracted from a random sample of the Spanish Wikipedia. The third dataset, **Indo** was created with the concatenation of the adjacency lists of Web graph **Indochina-2004** available at <http://law.di.unimi.it/datasets.php>. In Table 1 we provide some statistics about the datasets. We include the the number of symbols in the dataset (n) and the alphabet size (σ). Assuming P is the relative frequency of the alphabet symbols, $\mathcal{H}(P)$ indicates (in bits per symbol) the empirical entropy of the sequence. This is approximates the average ℓ value of queries. Finally we show L , the maximum code length, and the zero-order entropy of the sequence D , $\mathcal{H}_0(D)$, in bits per symbol. The last column is then a good approximation of the size of our Huffman-shaped wavelet tree for D .

Our test machine has a Intel(R) Core(tm) i7-3820@3.60GHz CPU (4 cores/8 siblings) and 64GB of DDR3 RAM. It runs Ubuntu Linux 12.04 (Kernel 3.2.0-99-generic). The compiler used was g++ version 4.6.4 and we set compiler optimization flags to $-O9$. All our experiments run in a single core and time measures refer to CPU *user-time*.

Figure 1 compares the space required by both code representations and their compression and decompression times. As expected, the space per character of our new code representation, WMM, is close to $1.37 \mathcal{H}_0(D)$, whereas that of TABLE is close to $2L + \lg \sigma$. This explains the large difference in space between both representations, a factor of 23–30 times. For decoding we show the mild effect of adding the structure that speeds up select queries.

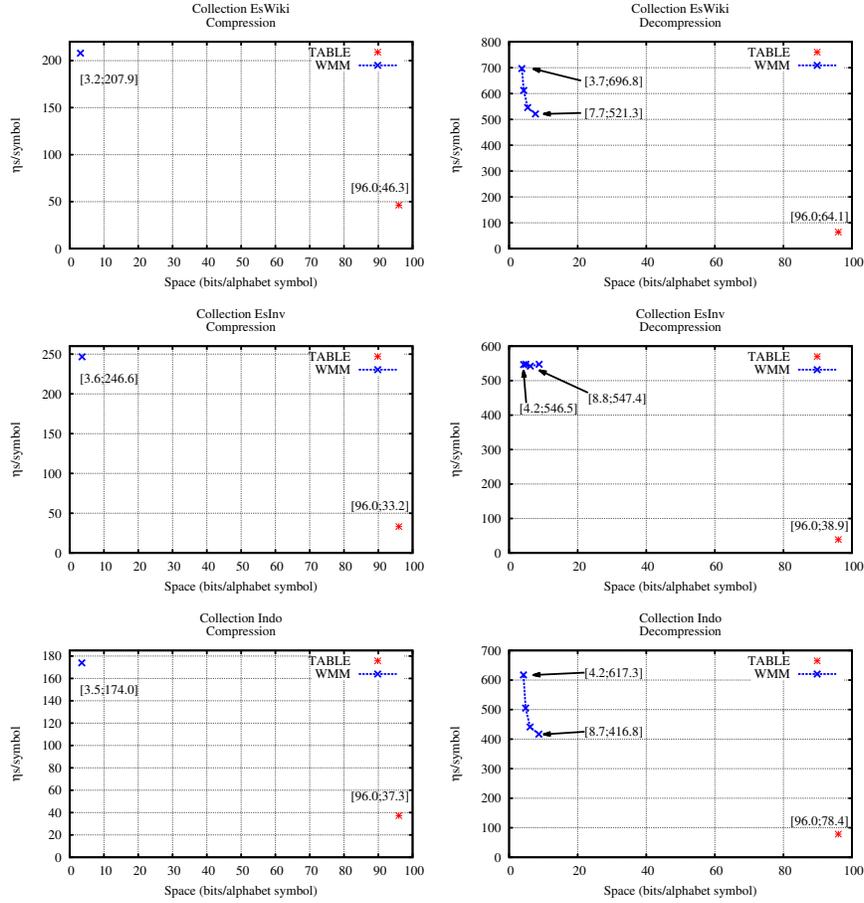


Fig. 1. Size of code representations versus either compression time (left) or decompression time (right). Time is measured in nanoseconds per symbol.

The price of our representation is the encoding and decoding time. While the **TABLE** approach encodes using a single table access, in 8–18 nanoseconds, our representation needs 130–230, which is 10 to 21 times slower. For decoding, the binary search performed by **TABLE** takes 20–50 nanoseconds, whereas our **WMM** representation requires 510–700 in the slowest and smallest variant (i.e., 11–30 times slower). Our faster variants require 300–510 nanoseconds, which is still several times slower.

5 Conclusions

A classical prefix code representation uses $\mathcal{O}(\sigma L)$ bits, where σ is the alphabet size and L the maximum codeword length, and encodes in constant time and decodes a codeword of length ℓ in time $\mathcal{O}(\ell)$. Canonical prefix codes can be represented in $\mathcal{O}(\sigma \log L)$ bits, so that one can encode and decode in constant time under reasonable assumptions. In this paper we have considered two families of codes that cannot be put in canonical form. Alphabetic codes can be represented in $\mathcal{O}(\sigma)$ bits, but encoding and decoding takes time $\mathcal{O}(\ell)$. We gave an approximation that worsens the average code length by a factor of $1 + \mathcal{O}(1/\sqrt{\log \sigma})$, but in exchange requires $o(\sigma)$ bits and encodes and decodes in constant time. We then consider a family of codes that are canonical when read right to left. For those we obtain a representation using $\mathcal{O}(\sigma \log L)$ bits and encoding and decoding in time $\mathcal{O}(\ell)$, or even in $\mathcal{O}(1)$ time under reasonable assumptions if we use $\mathcal{O}(2^{\epsilon L})$ further bits, for any constant $\epsilon > 0$.

We have implemented the simple version of these right-to-left codes, which are used for compressing wavelet matrices, and shown that our encodings are significantly smaller than classical ones in practice (up to 30 times), albeit also slower (up to 30 times). For the final version of the paper, we plan to implement the wavelet tree of D with a shape that lets it operate in time $\mathcal{O}(\ell)$ or $\mathcal{O}(\log \ell)$, as used to prove Theorem 2; currently we gave it Huffman shape in order to minimize space. Since there are generally more longer than shorter codewords, the Huffman shape puts them higher in the wavelet tree of D , so the longer codewords perform faster and the shorter codewords perform slower. This is the opposite effect as the one sought in Theorem 2. Therefore, a faithful implementation may lead to a slightly larger but also faster representation.

An interesting challenge is to find optimal alphabetic encodings that can encode and decode faster than in time $\mathcal{O}(\ell)$, even if they use more than $\mathcal{O}(\sigma)$ bits of space. Extending our results to other non-canonical prefix codes is also an interesting line of future work.

Acknowledgements

This research was carried out in part at University of A Coruña, Spain, while the second author was visiting and the fifth author was a PhD student there. It started at a StringMasters workshop at the Research Center on Information and Communication Technologies (CITIC) of the university. The workshop was

partly funded by EU RISE project BIRDS (Bioinformatics and Information Retrieval Data Structures). The authors thank Nieves Brisaboa and Susana Ladra for organizational support.

References

1. F. Claude, G. Navarro, and A. Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Systems*, 47:15–32, 2015.
2. W. Evans and D. G. Kirkpatrick. Restructuring ordered binary trees. *J. Algorithms*, 50:168–193, 2004.
3. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.
4. T. Gagie, M. He, J. I. Munro, and P. K. Nicholson. Finding frequent elements in compressed 2d arrays and strings. In *Proc. SPIRE*, pages 295–300, 2011.
5. T. Gagie, G. Navarro, Y. Nekrich, and A. Ordóñez. Efficient and compact representations of prefix codes. *IEEE Trans. Inf. Theory*, 61(9):4999–5011, 2015.
6. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.
7. A. Itai. Optimal alphabetic trees. *SIAM J. Comp.*, 5:9–18, 1976.
8. L. G. Kraft. *A device for quantizing, grouping, and coding amplitude modulated pulses*. M.Sc. thesis, EE Dept., MIT, 1949.
9. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comp.*, 31(3):762–776, 2001.
10. G. Navarro. Wavelet trees for all. *J. Discr. Alg.*, 25:2–20, 2014.
11. G. Navarro and E. Provedel. Fast, small, simple rank/select on bitmaps. In *Proc. SEA*, LNCS 7276, pages 295–306, 2012.
12. M. Pătraşcu. Succincter. In *Proc. FOCS*, pages 305–313, 2008.
13. E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Comm. of the ACM*, 7:166–169, 1964.
14. R. L. Wessner. Optimal alphabetic search trees with restricted maximal height. *Inf. Proc. Letters*, 4:90–94, 1976.