Information Processing Letters 179 (2023) 106274

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Space-efficient Huffman codes revisited

Szymon Grabowski[a], Dominik Köppl[b,*]

[a] Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, 90–924, Łódź, Poland
[b] Tokyo Medical and Dental University, M&D Data Science Center, 113-8510, Tokyo, Japan

ARTICLE INFO

Article history:
Received 13 August 2021
Received in revised form 11 April 2022
Accepted 24 April 2022
Available online 29 April 2022
Communicated by Leah Epstein

Keywords:
Data structures
Data compression
Huffman code
Canonical code
Compact representation

ABSTRACT

A canonical Huffman code is an optimal prefix-free compression code whose codewords enumerated in the lexicographical order form a list of binary words in non-decreasing lengths. Gagie et al. (2015) gave a representation of this coding capable of encoding and decoding a symbol in constant worst-case time. It uses $\sigma \lg \ell_{\max} + o(\sigma) + \mathcal{O}(\ell_{\max}^2)$ bits of space, where σ and $\ell_{\max}$ are the alphabet size and maximum codeword length, respectively. We refine their representation to reduce the space complexity to $\sigma \lg \ell_{\max}(1 + o(1))$ bits while preserving the constant encode and decode times. Our algorithmic idea can be applied to any canonical code.

1. Introduction

Huffman coding [16,25], whose compressed output closely approaches the zeroth-order empirical entropy, is nowadays perceived as a standard textbook encoder, being the most prevalent option when it comes to statistical compression. Unlike arithmetic coding [31], its produced code is *instantaneous*, meaning that a character can be decoded as soon as the last bit of its codeword is read from a compressed input stream. Thanks to this property, instantaneous codes tend to be swiftly decodable. While most research efforts focus on the achievable decompression speeds, not much has been considered for actually representing the dictionary of codewords, which is usually done with a binary code tree, called the Huffman tree.

Given that the characters of our text are drawn from an alphabet Σ of size σ, the *Huffman tree* is a full binary tree, where each leaf represents a character of Σ. The tree stored in a plain pointer-based representation takes $\mathcal{O}(\sigma \lg \sigma)$ bits of space. Here we assume that $\sigma \leq n$ and $\Sigma = \{1, 2, \ldots, \sigma\}$, where n is the number of characters of our text. A naive encoding algorithm for a character c traverses the Huffman tree top-down to the leaf representing c while writing the bits stored as labels on the traversed edges to the output. Similarly, a decoding algorithm traverses the tree top-down based on the read bits until reaching a leaf. Hence, the decoding and encoding need $\mathcal{O}(\ell_{\max})$ time, where $\ell_{\max}$ is the height of the tree. If σ is constant, then this representation is optimal with respect to space and time (since $\ell_{\max}$ becomes constant). Therefore, the problem we study becomes interesting if $\sigma = \omega(1)$, which is the setting of this article. If space is not of concern, we can create a lookup table, storing for each possible bit string B of length $\ell_{\max}$ the character associated with the codeword that is a prefix of B. The table takes $\mathcal{O}(2^{\ell_{\max}} \lg \sigma)$ bits of space and allows decoding a symbol in constant time. If the table additionally stores the length of the codeword associated with the returned character, we know how many bits we can drop from the encoded input.

The alphabet size σ is often much smaller than n. Yet in some cases the Huffman tree space becomes non-negligible: For instance, let us consider that Σ represents

---

[*] Corresponding author.
E-mail addresses: sgrabow@kis.p.lodz.pl (S. Grabowski), koeppl.dsc@tmd.ac.jp (D. Köppl).

https://doi.org/10.1016/j.ipl.2022.106274

a set of distinct words of a language. Then its size can even exceed a million[1] for covering an entire natural language. In another case, we maintain a collection of texts compressed with the same Huffman codewords under the setting that a single text needs to be shipped together with the codeword dictionary, for instance across a network infrastructure. A particular variation is *canonical Huffman codes* [30], where leaves of the same depth in the Huffman tree are lexicographically ordered with respect to the characters they represent. An advantage of canonical Huffman codes is that several techniques (see [23,26] and the references therein) can compute the lengths of a codeword from the compressed stream by reading bit chunks instead of single bits, thus making it possible to decode a character faster than an approach linear in the bit-length of its codeword. Moffat and Turpin [26, Algorithm TABLE-LOOKUP] were probably the first to notice that it is enough to (essentially) perform binary search over a collection of the lexicographically smallest codewords for each possible length, to decode the current symbol. The same idea is presented perhaps more lucidly in [27, Section 2.6.3], with an explicit claim that this representation requires $\sigma \lg \sigma + \mathcal{O}(\lg^2 n)$ bits[2] and achieves $\mathcal{O}(\lg\lg n)$ time per codeword. Given $\ell_{\max}$ is the maximum length of the codewords, an improvement, both in space and time, has been achieved by Gagie et al. [13], who gave a representation of the canonical Huffman tree within

- $\sigma \lg \ell_{\max} + o(\sigma) + \mathcal{O}(\ell_{\max}^2)$ bits of space with $w \in \Omega(\ell_{\max})$ [13, Theorem 1], or
- $\sigma \lg\lg(n/\sigma) + \mathcal{O}(\sigma) + \mathcal{O}(\ell_{\max}^2)$ bits of space with $w \in \Omega(\lg n)$ [13, Corollary 1],[3]

while supporting character encoding and decoding in $\mathcal{O}(1)$ time, where w is the machine word size. Their approach consists of a wavelet tree, a predecessor data structure P, and an array First storing the lexicographically smallest codeword for each depth of the Huffman tree. The last two data structures are responsible for the last term of $\mathcal{O}(\ell_{\max}^2)$ bits in each of the space complexities above.

Our contribution Our contribution is a joint representation of the predecessor data structure P with the array First to improve their required space of $\mathcal{O}(\ell_{\max}^2)$ bits down to $o(\ell_{\max} \lg \sigma + \sigma)$ bits, and this space is in $o(\sigma \lg \ell_{\max})$ bits since $\ell_{\max} \leq \sigma - 1$. In other words, we obtain the following theorem.

Theorem 1.1. *There is a data structure using $\sigma \lg \ell_{\max}(1 + o(1))$ bits of space, which can encode a character to a canonical Huffman codeword or restore a character from a binary input stream of codewords, both in constant time per character.*

We remark that when the alphabet consists of natural language words, like in the examples shown in the beginning, we often initially map words to IDs in an arbitrary manner. For our use case, it would be convenient to assign the IDs based on their frequencies. By doing so, the needed space of canonical Huffman representation is greatly reduced as the alphabet permutation can be considered as already given. In such a case, we only consider the extra space needed for the Huffman coding, where our improvement in space from $\mathcal{O}(\ell_{\max}^2)$ down to $o(\ell_{\max} \log \sigma)$ is more pronounced (in detail, we can omit the wavelet tree of Gagie et al.'s achieved space complexities).

This article is organized as follows. After presenting related work in the next paragraphs, we review some preliminaries (Section 2), in particular the work of Gagie et al. [13] in Section 2.2. Subsequently (Section 3), we start with a small theoretical improvement of a practical solution based on a predecessor search in a list of codewords, before proposing our solution in Section 4 leading to Theorem 1.1.

Related work A lot of research effort has been spent on the practicality of decoding Huffman-encoded texts (e.g., [15,1,26,14,24]) where often a lookup table is employed for figuring out the lengths of the currently read codeword. For instance, Nekrich [28] added a lookup table for byte chunks of a codeword to figure out the length of a codeword by reading it byte-wise instead of bit-wise. Regarding different representations of the Huffman tree, Chowdhury et al. [7] came up with a solution using $\lceil 3\sigma/2 \rceil$ words of space, which can decode a codeword in $\mathcal{O}(\log \sigma)$ time.

Our theoretical results are built on the foundation laid out by Gagie et al. [13], which is heavily cited in this work, and is discussed in detail in Section 2.2. An approach with related techniques to that of Gagie et al. is due to Fariña et al. [10] (see also an extended version [11]), who showed how to represent a particular optimal prefix-free code used for compressing wavelet matrices [8] in $\mathcal{O}(\sigma \lg \ell_{\max} + 2^{\varepsilon \ell_{\max}})$ bits, allowing $\mathcal{O}(1/\varepsilon)$-time encoding and decoding, for a selectable constant $\varepsilon > 0$. Like Gagie et al. [13], they also use a wavelet tree of $\mathcal{O}(\sigma \lg \ell_{\max})$ bits to map each character to the length of its respective codeword. The tree topology is represented by counting level-wise the number of nodes and leaves, resulting in $\mathcal{O}(\ell_{\max} \lg \sigma) \subset \mathcal{O}(\sigma \lg \ell_{\max})$ bits. With these two ingredients, this structure is already operational with $\mathcal{O}(\ell_{\max})$ time per encoding or decoding operation. To obtain the aforementioned time bounds, they sample certain depths of the code tree with lookup tables to speed up top-down traversals.

Another line of research is on so-called skeleton trees [19–21]. The idea is to replace disjoint perfect subtrees[4] in the canonical Huffman tree with leaf nodes. A leaf node representing a pruned perfect subtree only needs to store the height of this tree and the characters represented by its leaves to be able to reconstruct it. Note that all leaves

---

[1] Two such examples are the Polish Scrabble dictionary with over 3.0M words (https://sjp.pl/slownik/growy/) and the Korean dictionary *Woori Mal Saem* with over 1.1M words (https://opendict.korean.go.kr/service/dicStat).

[2] By lg we mean the logarithm to base two ($\log_2$) throughout the paper. Occasional logarithms to another base b are denoted with $\log_b$.

[3] Although the space is stated as $\sigma \lg\lg(n/\sigma) + \mathcal{O}(\sigma) + \mathcal{O}(\lg^2 n)$ bits in that paper, we could verify with the authors that the last term can be improved to the bounds written here.

[4] A binary tree is *perfect* if every internal node has exactly two children and all the leaves are at the same depth.

in a perfect binary tree are on the same level, and hence due to the restriction on the order of the leaves of the canonical Huffman tree, we can restore the pruned subtree by knowing its height and the set of characters. Thus, a skeleton tree may use less space. Decompression can be accelerated whenever we hit a leaf λ during a top-down traversal, where we know that the next h bits from the input are equal to the suffix of the currently parsed codeword if the leaf λ is the representative of a perfect subtree of height h. Unfortunately, the gain depends on the shape of the Huffman tree, and all that is known about the skeleton tree size is that it has $\mathcal{O}(\lg^2 n)$ nodes for a Huffman tree of height $\mathcal{O}(\lg n)$ [20], or, more generally, at most $2h \lg n$ for the Huffman tree height of h [21, comment following Lemma 2].

Related to our problem of keeping the space for maintaining codewords up to the length $\ell_{\max}$ small is the problem of computing the smallest code under the restriction that $\ell_{\max}$ is user-given [22].

2. Preliminaries

Our computational model is the standard word RAM with machine word size $w = \Omega(\lg n)$ bits, where n denotes the length of a given input string $T[1..n]$ (called the *text*) whose characters are drawn from an integer alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$ with size σ being bounded by $\sigma \leq n$ and $\sigma = \omega(1)$. We call the elements of Σ *characters*. Given a string $S \in \Sigma^*$, we define the queries $S.rank_c(i)$ and $S.select_c(j)$ returning the number of c's in $S[1..i]$ and the position of the j-th c in S, respectively. There are data structures that replace S and use $|S| \lg \sigma + o(|S|)$ bits, and can answer each query in $\mathcal{O}(1 + \log_w \sigma)$ time [3, Theorem 4.1]. We further stipulate that $S.select_c(0) := 0$ for any $c \in \Sigma$. In what follows, we assume that we have at least $\Theta(w)$ bits of working space available for storing a constant number of pointers, and omit this space in the space analysis of this article.

2.1. Code tree

A binary tree is *full* if each node has either none or two children. In this article, a *code tree* is defined to be a full binary tree[5] whose leaves represent the characters of Σ. A left or a right child of a node v is connected to v via an edge with label `0` or `1`, respectively. We say that the *string label* of a node v is the concatenation of edge labels on the path from the root to v. Then the string label of a leaf λ is the codeword of the character represented by λ. The length of the string label of a node is equal to its depth in the tree. Let $\mathcal{C}$ denote the set of the string labels of all leaves. The bit strings of $\mathcal{C}$ are called *codewords*. A crucial property of a code tree is that $\mathcal{C}$ is prefix-free, meaning that no codeword in $\mathcal{C}$ is prefix of another. A consequence is that, when reading from a binary stream of concatenated codewords, we only have to match the longest prefix of this stream with $\mathcal{C}$ to decode the next codeword. Hence, prefix-free codes are instantaneous, where an *instantaneous code*, as described in the introduction, is a code with the property that a character can be decoded as soon as the last bit of its codeword is read from the compressed input. Instantaneous codes are also prefix-free. That is because, if a code is not prefix-free, then there can be two different codewords C_1 and C_2 with C_2 being a prefix of C_1 such that when reading C_2 we need to read the next $|C_1| - |C_2|$ bits to judge whether the read bits represent C_1 or C_2.

A code tree is called *canonical* [30] if its induced codewords read from left to right are lexicographically strictly increasing,[6] while their lengths are non-decreasing. Consequently, a codeword of a leaf λ is lexicographically smaller than the codeword of any leaf deeper than λ. We further assume that all leaves on the same depth are sorted according to the order of their respective characters.

Let $\ell_{\max}$ denote the maximum length of the codewords in $\mathcal{C}$. In the following we fix an instance of a code tree for which $\ell_{\max} \leq \min(\sigma - 1, \log_\phi n)$ holds, where $\phi = (1 + \sqrt{5})/2 \approx 1.618$ is the golden ratio. It has been shown by Buro [6, comment after Theorem 2.1] that the canonical Huffman coding exhibits this property.

Our claims in Sections 3 and 4, expressed in terms of the canonical Huffman codes and trees, are applicable to arbitrary canonical codes (resp. trees) if they obey the above bound on $\ell_{\max}$. Our focus on Huffman codes is motivated by the importance of this coding.

2.2. Former approach

We briefly review the approach of Gagie et al. [13], which consists of three data structures:

1. a multi-ary wavelet tree storing, for each character, the length of its corresponding codeword,
2. an array First storing the codeword of the leftmost leaf on each depth, and
3. a predecessor data structure P for finding the length of a codeword that is the prefix of a read bit string.

The first data structure represents an array $L[1..\sigma]$ with $L[i]$ being the codeword length of the i-th character, i.e., L is a string of length σ whose characters are drawn from the alphabet $\{1, 2, \ldots, \ell_{\max}\}$. The wavelet tree built upon L can access L, and answer $L.rank$ and $L.select$ in $\mathcal{O}(1 + \log_w \ell_{\max})$ time [3, Theorem 4.1]. While a plain representation costs $\sigma \lg \ell_{\max} + o(\sigma)$ bits, Gagie et al. [13] showed that they can use an empirical entropy-aware topology of the wavelet tree to get the space down to $\sigma \lg \lg(n/\sigma)$ bits, still accessing L, and answering *rank* and *select* in $\mathcal{O}(1 + \log_w \ell_{\max})$ time by using the entropy-compressed wavelet tree of Belazzougui and Navarro [3, Theorem 5.1]. Since $\ell_{\max} \in \mathcal{O}(\log n)$, $\mathcal{O}(\log_w \ell_{\max}) \subset \mathcal{O}(1)$, and hence the time complexity is constant.

The second data structure called First is a plain array of length $\ell_{\max}$. Each of its entries has $\ell_{\max}$ bits, therefore it takes $\mathcal{O}(\ell_{\max}^2)$ bits in total. It additionally stores $\ell_{\max}$ in

---

[5] We assume $\sigma \geq 2$. Note that there exist codes whose code trees are not full, which are, however, not in scope of this paper.

[6] This is obviously the case if we sort the children of all internal nodes to form a trie.

$\mathcal{O}(\lg \ell_{\max})$ bits (for instance, in a prefix-free code such as Elias-γ [9]).

The last data structure P is represented by a fusion tree [12] storing for each depth the lexicographically smallest codeword padded to $\ell_{\max}$ bits at its right end, where we assume the least significant bit is stored. For instance, `110` denotes the integer 6 in binary, and its padding with $\ell_{\max} = 7$ is `1100000`. A fusion tree storing m elements of a set $\mathcal{S}$, each represented in m bits, needs $\mathcal{O}(m^2)$ bits of space and can answer the query $\mathit{pred}(X)$ returning the predecessor of X in $\mathcal{S}$, i.e., $\max\{Y \in \mathcal{S} : Y \leq X\}$ in $\mathcal{O}(\log_w m)$ time. For our application, the fusion tree P built on codewords computes pred in $\mathcal{O}(\log_w \ell_{\max})$ = $\mathcal{O}(1)$ time, and needs $\mathcal{O}(\ell_{\max}^2)$ bits. Here, our query to P is a bit string having a codeword C as a prefix (since our codewords are prefix-free, there can be at most one codeword that is a prefix of an arbitrary but fixed bit string). Instead of returning C, it is sufficient for us to let P just return the length ℓ of C; we then can retrieve C with $\mathsf{First}[\ell]$. Unfortunately, it is not explicitly mentioned by Gagie et al. [13] how to obtain this length, i.e., the depth of the leaf corresponding to C, in particular when some depths may have no leaves. We address this problem with the following subsection and explain after that how the actual computation is done (Section 2.2.2).

2.2.1. *Missing leaves*

To extract the depth from a predecessor query on P, we replace each right-padded codeword C_i by the pair (C_i, ℓ_i) as the keys stored in P, where ℓ_i is the length of C_i. Such a pair of codeword and length is represented by the concatenation of the binary representations of its two components such that we can interpret this pair as a bit string of length $\ell_{\max} + \lg \ell_{\max}$. By slightly abusing the notation, we can now query $\mathit{pred}((B, \ell))$ to obtain $(\bar{B}, \bar{\ell})$, i.e., the argument for $\mathit{pred}(\cdot)$ is a pair rather than a single value and similarly the returned value is a pair, but physically the bit string $\bar{B}\bar{\ell}$ is a predecessor of $B\ell$. Then $\mathit{pred}((X, 1^{\lg \ell_{\max}})) = (C_i, \ell_i)$ for a bit string $X \in \{0,1\}^{\ell_{\max}}$ gives us not only the predecessor C_i of X, but also ℓ_i. Storing such long bit strings poses no problem as the time complexity of $\mathit{pred}(X)$ for bit string X does not change in a fusion tree when the length of X is, e.g., doubled. Appendix A outlines a different strategy augmenting the fusion tree with additional methods instead of storing the codeword lengths.

2.2.2. *Encoding and decoding*

Finally, we explain how the operational functionality is implemented, i.e., the steps to encode a character, and to decode a character from a binary stream. We can encode a character $c \in \Sigma$ by first finding the depth of the leaf λ representing c with $\ell = L[c]$, which is also the length of the codeword to compute. Given the string label of the leftmost leaf λ' on depth ℓ is $\mathsf{First}[\ell]$, then all we have to do is increment the value of this codeword by the number of leaves between λ and λ' on the same depth. For that we compute $L.\mathit{rank}_\ell(c)$ that gives us the number of leaves to the left of λ on the same depth ℓ. Hence, $\mathsf{First}[\ell] + L.\mathit{rank}_\ell(c) - 1$ is the codeword of c. For decoding a character, we assume to have a binary stream of concatenated codewords as an input. We first use the predecessor data structure to figure out the length of the first codeword in the stream. To this end, we peek the next $\ell_{\max}$ bits in the binary stream, i.e., we read $\ell_{\max}$ bits into a variable X from the stream *without* removing them. Given $(C', \ell) = \mathsf{P}.\mathit{pred}((X, 1^{\lg \ell_{\max}}))$, we know that $\mathsf{First}[\ell] = C'$ and that the next codeword has length ℓ. Hence, we can read ℓ bits from the stream into a bit string C, which must be the codeword of the next character. The leaf having C as its string label is on depth ℓ, and has the rank $r = C - \mathsf{First}[\ell] + 1$ among all other leaves on the same depth. This rank helps us to retrieve the character having the codeword C, which we can find by $L.\mathit{select}_\ell(r)$.

3. A warm-up

We start with a simple idea unrelated to the main contribution presented in the subsequent section. The point is that it might have a (mild) practical impact. Also, we admit this idea is not new. Fariña et al. [11, Section 2.4] attribute it to Moffat and Turpin [26], although, in our opinion, it is presented rather in disguise. Additionally, Karpinski and Nekrich [17, Section 7] presented similar ideas for decoding Shannon codes. However, we hope that the analysis given below is original and of some value.

Navarro, in Section 2.6.3 of his textbook [27], describes a simple solution (based on an earlier work of Moffat and Turpin [26]) which gives $\mathcal{O}(\lg \lg n)$ worst-case time for symbol decoding. With the modification presented below, this worst-case time remains, but the average time bound becomes $\mathcal{O}(\lg \lg \sigma)$. It also means we can decode the whole text in $\mathcal{O}(n \lg \lg \sigma)$ rather than $\mathcal{O}(n \lg \lg n)$ time.

The referenced solution is based on a binary search over First represented by a list (with constant time random access) storing the lexicographically smallest codeword for each possible length; this list is ordered ascendingly by the codeword length. We replace the binary search with the exponential search [5], which finds the predecessor of item key in First in $\mathcal{O}(\lg \mathit{pos}_L(\mathsf{key}))$ time, where $\mathit{pos}_L(\mathsf{key})$ is the position of key in First. Exponential search is not faster than binary search in general, but may help if key occurs close to the beginning of First.

The changed search strategy has an advantage whenever short codewords occur much more often than longer ones. Fortunately, we have such a distribution, which is expressed formally in the following lemma:

Lemma 3.1. *The number of occurrences of all characters in the text whose associated Huffman codewords have lengths exceeding $2 \log_\phi \sigma$ is $\mathcal{O}(n/\sigma)$.*

Proof. For each character $c \in \Sigma$, let f_c denote the number of occurrences of c in the text and d_c denote the depth of its associated leaf in the Huffman tree, which is also the length of its associated codeword. Here we are interested in those characters c for which $d_c > 2 \log_\phi \sigma = \log_\phi(\sigma^2)$. Since $d_c = \mathcal{O}(\log_\phi(n/f_c))$ [18, Thm. 1] and $\log_\phi$ is a strictly increasing function, we have $n/f_c = \Omega(\sigma^2)$, or, equivalently, $f_c = \mathcal{O}(n/\sigma^2)$. The number of characters we deal with is upper-bounded by σ, therefore the total number of their occurrences is $\mathcal{O}(n/\sigma)$, which ends the proof. $\square$

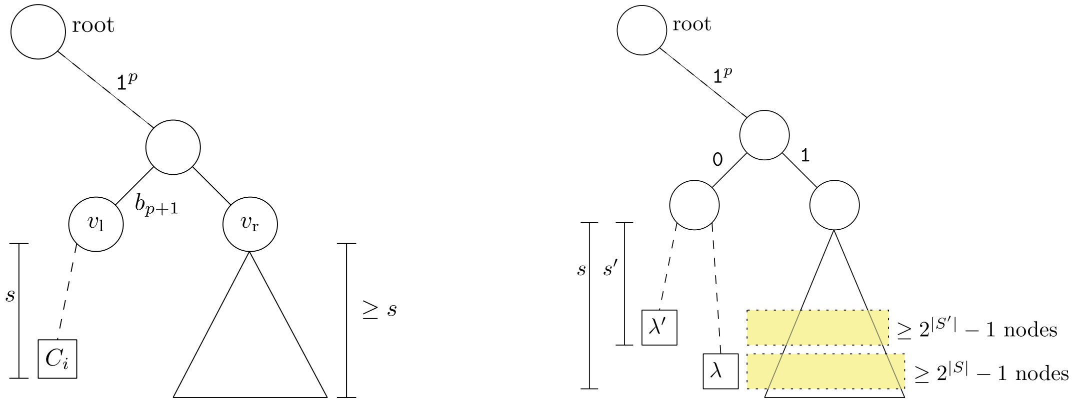

Fig. 1. Sketches of the code trees considered in the proofs of Lemma 4.1 (left) and Lemma 4.2 (right), respectively.

For the $\mathcal{O}(n/\sigma)$ characters specified in Lemma 3.1 the exponential search works in $\mathcal{O}(\min\{\lg\sigma, \lg\lg n\})$ time. However, for the remaining $\Theta(n)$ characters of the text the exponential search works in only $\mathcal{O}(\lg(2\log_\phi \sigma)) = \mathcal{O}(\lg\lg\sigma)$ time. Overall, the average time is $\mathcal{O}(\lg\lg\sigma)$, which is also the total average character decoding time, as finding the lexicographically smallest codeword with the appropriate length is, in general, more costly than all other required operations, which work in constant time.

In practice, one would rather resort to a linear scan over First, as the Huffman codewords are rather short. This is in particular interesting if we consider modern computer architectures, leveraging bit-parallelism.

4. Multiple fusion trees

In what follows, we present our space efficient representation of P and First for achieving the result claimed in Theorem 1.1. In Section 4.1, we start with the key observation that long codewords in First have a necessarily long prefix of ones. This observation lets us group together codewords of roughly the same lengths, storing only the suffixes that distinguish them from each other. Hence, we proceed with partitioning the set of codewords into codewords of different lengths, and orthogonal to that, of different distinguishing suffix lengths.

4.1. Distinguishing suffixes

Let us notice at the beginning that a long codeword has a necessarily long prefix of ones.

Lemma 4.1. *Let $\mathcal{C} = \{C_1, \ldots, C_\sigma\}$ be a canonical code. Each codeword $C_i \in \mathcal{C}$ can be represented as either a bit string of ones $C_i = \mathtt{1}^{|C_i|}$ or as a bit string $C_i = \mathtt{1}^p\mathtt{0}S$ with $S \in \{\mathtt{0}, \mathtt{1}\}^*$ and $0 \leq |S| < \lg\sigma$ for some $p \geq 0$.*

Proof. Given a codeword C_i represented as $C_i = \mathtt{1}^p\mathtt{0}S$ for a bit string $S \in \{\mathtt{0}, \mathtt{1}\}^*$ (the second case in the claim), we have to show that the length of S is less than $\lg\sigma$. Without loss of generality, let σ be a power of two, i.e., $\lg\sigma$ is an integer. Let us assume there is a codeword $C_i = \mathtt{1}^p b_{p+1} b_{p+2} \ldots b_{|C_i|}$ with $b_{p+1} = 0$. The length of the suffix $b_{p+2} \ldots b_{|C_i|}$ of C_i is $s := |S| = |C_i| - p - 1 \geq 0$. The prefix $\mathtt{1}^p b_{p+1}$ of C_i is the string label of a node v_l in the canonical tree of $\mathcal{C}$. The height of v_l is at least s since it has a leaf whose string label is C_i. Since a canonical tree is a full binary tree, v_l has a right sibling, which we call v_r. Since the depths of the leaves iterated in the left-to-right order in a canonical tree are non-decreasing, all leaves of the tree rooted at v_r must be at depth at least s. This implies that the number of leaves in the tree rooted in v_r is at least 2^s. See Fig. 1 (left) for a sketch. The subtree rooted at v_l has at least one leaf (it has exactly one leaf if v_l is a leaf itself). Moreover, the two trees rooted in v_l and v_r, respectively, are disjoint. Consequently, there are at least $2^s + 1$ leaves in the tree representing $\mathcal{C}$. For $s \geq \lg\sigma$, we obtain a contradiction since the code tree has exactly σ leaves. Hence, $|C_i| - p - 1 = s < \lg\sigma$. □

The lemma states that given a codeword perceived as a concatenation of a maximal run of set ‘1’ bits and a suffix that follows it, the length of this suffix is less than $\lg\sigma$. According to that, we can group the codewords of similar lengths together and store only their distinguishing $\Theta(\log\sigma)$-bit suffixes. Roughly speaking, this would already give a space bound of $\mathcal{O}(\ell_{\max}\log\sigma)$ bits for maintaining all codewords with multiple P and First instances. Considering only the lexicographically smallest codewords per length, we achieve a better bound with the following lemma.

Lemma 4.2. *Given two non-negative lengths p and s, the number of codewords C_i corresponding to the* leftmost *leaves on each depth of a code tree given by $C_i = \mathtt{1}^p\mathtt{0}S$ with $S \in \{\mathtt{0}, \mathtt{1}\}^*$ and $|S| \geq s$ is bounded by $\mathcal{O}(\sigma/2^s)$.*

Proof. Let S be the longest string such that $\mathtt{1}^p\mathtt{0}S$ is the string label of a leaf λ with the property that λ is the leftmost leaf on its depth. By the shape of the code tree, one of the deepest leaves whose string labels have $\mathtt{1}^p\mathtt{0}$ as a prefix is a leftmost leaf, and this leaf is λ. By the proof of Lemma 4.1, there is a node with string label $\mathtt{1}^{p+1}$ having at least $2^{|S|}$ nodes in its subtree; in particular, there are at least 2^x nodes on depth $p + x$ for $x \leq |S| - 1$. We charge the $2^{|S|-1}$ nodes on the depth of λ for λ, and more generally, we charge $2^{|S'|-1}$ nodes on the depth of λ' for each leftmost leaf λ' with a string label of the form $\mathtt{1}^p\mathtt{0}S'$ with $|S'| \le |S|$. By doing so, none of the nodes is charged twice. Since the number of nodes in the Huffman (or any other full binary) tree is $2\sigma - 1$, the overall number of such leftmost leaves on a depth of at least $p + 1 + s$ can be at most $(2\sigma - 1)/2^{s-1}$. A visualization is given in Fig. 1 (right). □

In what follows, we want to derive a partition of the codewords stored in P. Let us recall that P stores not all codewords, but the codeword of the *leftmost* leaf on each depth (and omits depth d if there is no leaf with depth d). Given a set of such codewords $C_1, \dots, C_{\ell'_{\max}}$ with $\ell'_{\max} \le \ell_{\max}$, let s_i denote the length of the shortest suffix of the representation $C_i = \mathtt{1}^p S$ with $S \in \{\mathtt{0}, \mathtt{1}\}^{s_i}$. In what follows, we call a codeword C_i *long-tailed* if $s_i \ge 2 \lg\lg\sigma = \lg\lg^2\sigma$, and otherwise *short-tailed*. A consequence of Lemma 4.2 is that $\mathcal{O}(\sigma/\lg^2\sigma)$ codewords can be long-tailed. In what follows, we manage long-tailed and short-tailed codewords separately.

4.2. Long-tailed codewords

We partition the long-tailed codewords into sets $\mathcal{C}_1, \dots, \mathcal{C}_m$ with $\mathcal{C}_k$ being the set of codewords of lengths within the range $[1 + (k-1)\lg\sigma, k\lg\sigma]$, for each $k \in [1..m]$. By Lemma 4.1, we know that codewords in $\mathcal{C}_k$ have the shape PS with $P \in \{\mathtt{1}\}^*$, $|P| \ge (k-2)\lg\sigma + 1$ and $0 \le |S| \le \lg\sigma$. We are therefore sure that the $((k-2)\lg\sigma + 1)$-length prefix of the codewords in $\mathcal{C}_k$ is a run of $\mathtt{1}$s (the case that $(k-2)\lg\sigma + 1 \le 0$ is treated below separately). Consequently, we can cut off this prefix of each codeword, and obtain trimmed codewords of length less than $2\lg\sigma$ bits. (We can restore the original codeword by prepending the cut-off $(k-2)\lg\sigma + 1$ $\mathtt{1}$s to the respective trimmed codeword.) To induce the original lexicographical order of the original codewords onto their trimmed counterparts, we pad these trimmed codewords at their left ends with zeros such that each trimmed codeword has $2\lg\sigma$ bits. To now restore the original codeword from this padded version, we additionally store its *local depth*, which is the length of the trimmed codeword before padding.

Instead of representing P with a single fusion tree storing the lexicographically smallest codeword of $\mathcal{C}$ for each depth, we maintain for each such set $\mathcal{C}_k$ a dedicated fusion tree F_k. To know which fusion tree to consult during a query with a bit string B read from a binary stream, we compute the longest prefix $P \in \{\mathtt{1}\}^*$ of B (that is, a maximal run of ones). We can do that by asking for the position $h \in [1..|B|]$ of the most significant set bit of B bit-wise negated, which can be computed in constant time [12]. Then we know that $|P| = h - 1 \ge 0$. Since $\mathcal{C}_k$ captures codewords having runs of ones of lengths in $[(k-2)\lg\sigma + 1..k\lg\sigma]$ as a prefix, we find our codeword in either $\mathsf{F}_{\lceil |P|/\lg\sigma \rceil}$ or $\mathsf{F}_{\lceil |P|/\lg\sigma \rceil + 1}$.

This works fine unless we encounter the following two border cases. Firstly, P is empty if and only if $h = 1$. In that case, our codeword starts with $\mathtt{0}\cdots$. By Lemma 4.1, its length is at most $\lg\sigma$. Hence, we can maintain all these long-tailed codewords starting with $\mathtt{0}$ in an extra fusion tree F_0 without modification (like the aforementioned trimming).

The second border case arises when the predecessor is not stored in the queried fusion tree, but in one built on shorter codewords. To treat that case, for $k \in [1..m]$, we store the dummy codeword $\mathtt{0}$ in F_k and cache the largest value (i.e., the longest codeword length) of $\mathsf{F}_0, \dots, \mathsf{F}_{k-1}$ in F_k such that F_k returns this cached value instead of $\mathtt{0}$ in the case that $\mathtt{0}$ is the returned predecessor. Finally, we treat the border case for F_0, in which we store the smallest codeword $C_1 = \mathsf{First}[1] = \mathtt{0}^{|C_1|}$ regardless of whether C_1 is long-tailed or not — in that way we are sure that a predecessor always exists.

Since the lengths of the codewords (before truncation) in the sets $\mathcal{C}_1, \dots, \mathcal{C}_m$ are pairwise disjoint, the number of elements stored in the fusion trees $\mathsf{F}_1, \dots, \mathsf{F}_m$ is the number of long-tailed codewords, which is bounded by $\mathcal{O}(\sigma/\lg^2\sigma)$. A key in a fusion tree represents a long-tailed codeword C by a pair consisting of C's suffix of length $\lg\sigma$ (thus using $\lg\sigma$ bits) and C's length packed in $\lg\ell_{\max} \le \lg\sigma$ bits. Our total space for the long-tailed codewords is $\mathcal{O}(\sigma/\lg\sigma)$ bits, which are stored in $\mathcal{O}(\ell_{\max}/\lg\sigma)$ fusion trees. (The additional cost for storing the dummy codeword $\mathtt{0}$ in all fusion trees is covered by the total space of the fusion trees, which we study later in Section 4.4.)

4.3. Short-tailed codewords

Similar to our strategy for long-tailed codewords, we partition the short-tailed codewords by their total lengths. By doing so, we obtain similarly a set of codewords $\mathcal{C}'_1, \dots, \mathcal{C}'_{m'}$ with $\mathcal{C}'_k$ being the set of codewords of lengths within the range $[1 + (k-1)\lg\lg^2\sigma, k\lg\lg^2\sigma]$. Like in Section 4.2, we build a fusion tree on each of the codeword sets $\mathcal{C}'_k$ with the same logic.

This time, however, we know that each short-tailed codeword C_i has the shape $\mathtt{1}^{|C_i| - s_i} S_i$ for $S_i \in \{\mathtt{0}, \mathtt{1}\}^{s_i}$ and $|S_i| = s_i < 2\lg\lg\sigma = \lg\lg^2\sigma$. We trim, similarly to the long-tailed codewords, the $1 + (k-2)\lg\lg^2\sigma$ long prefix of $\mathtt{1}$s from each codeword to obtain a trimmed codeword of length at most $4\lg\lg\sigma$. Additionally, we represent the length of the codeword C_i in the set $\mathcal{C}'_k$ by its local depth within $\mathcal{C}'_k$ as a $\mathcal{O}(\lg\lg\lg\sigma)$-bit integer. Hence, the key of a fusion tree is composed by the $(4\lg\lg\sigma)$-bit long suffix and the local depth with $\mathcal{O}(\lg\lg\lg\sigma)$ bits. Therefore, the total space for the short-tailed codewords is $\mathcal{O}(\ell_{\max}\lg\lg\sigma)$ bits, which are stored in $\mathcal{O}(\ell_{\max}/\lg\lg\sigma)$ fusion trees.

4.4. Complexities

Summing up the space for the short- and long-tailed codewords, we obtain $\mathcal{O}(\ell_{\max}\lg\lg\sigma + \sigma/\lg\sigma)$ bits, which are stored in $\mathcal{O}(\ell_{\max}/\lg\lg\sigma)$ fusion trees. To maintain these fusion trees in small space, we assume that we have access to a separately allocated RAM of above stated size to fit in all the data. While we have a global pointer of $\mathcal{O}(w)$ bits to point into this space, pointers inside this space take $\mathcal{O}(\lg\sigma)$ bits. Hence, we can maintain all fusion trees inside this allocated RAM with an extra space of $\mathcal{O}(\ell_{\max}\lg\sigma/\lg\lg\sigma)$ bits. In total, the space for P is $\mathcal{O}(\ell_{\max}\lg\sigma/\lg\lg\sigma + \sigma/\lg\sigma) = o(\ell_{\max}\lg\sigma + \sigma)$ bits.

To answer $\mathsf{P}.\mathit{pred}(B)$, we find the predecessor of B among the short- and long-tailed codewords, and take the maximum one. In detail, this is done as follows. Remembering the decoding process outlined in Section 2.2.2, B is a bit string of length $\ell_{\max}$, which we delegate to the fusion trees F_k and F_{k+1} (cf. the second paragraph of Section 4.2) corresponding to either $\mathcal{C}_k$, $\mathcal{C}_{k+1}$, $\mathcal{C}'_k$, or $\mathcal{C}'_{k+1}$ (we try all four possibilities) if the longest unary prefix of '`1`'s in B is in $[1+(k-1)\lg\sigma, k\lg\sigma]$ (long-tailed) or $[1+(k-1)\lg\lg^2\sigma, k\lg\lg^2\sigma]$ (short-tailed). By delegation we mean that we remove this unary prefix, take the $2\lg\sigma$ most significant bits (long-tailed) or the $2\lg\lg\sigma$ most significant bits (short-tailed) from B, and store these bits in a variable B' used as the argument for the query $\mathsf{F}_k.\mathit{pred}(B')$.

Finally, it remains to treat First for encoding a character (cf. Section 2.2.2). One way would be to partition First analogously like P. Here, we present a solution based on the already analyzed bounds for P. We create a duplicate of P, named P′, whose difference to P is that the components of the stored pairs are swapped, such that a predecessor query is of the form $\mathsf{P}'.\mathit{pred}((\ell, B))$ for a length ℓ and a bit string B of length $\ell_{\max}$. Then $\mathsf{P}'.\mathit{pred}((\ell, 1^{\ell_{\max}})) = (\ell', B')$, and the ℓ first/leftmost bits of B' are equal to $\mathsf{First}[\ell]$ if $\ell = \ell'$ (otherwise, if $\ell \neq \ell'$, then there is no leaf at depth ℓ).

5. Conclusion and open problems

Canonical codes (e.g., Huffman codes) are an interesting subclass of general prefix-free compression codes, allowing for compact representation without sacrificing character encoding and decoding times. In this work, we refined the solution presented by Gagie et al. [13] and showed how to represent a canonical code, for an alphabet of size σ and the maximum codeword length of $\ell_{\max}$, in $\sigma\lg\ell_{\max}(1+o(1))$ bits, capable to encode or decode a symbol in constant worst case time. Our main idea was to store codewords not in their plain form, but partition them by their lengths and by the length of the shortest suffix covering all '`0`' bits of a codeword such that we can discard unary prefixes of '`1`'s from all codewords.

This research spawns the following open problems: First, we wonder whether the proposed data structure works in the AC^0 model. As far as we are aware of, the fusion tree can be modeled in the AC^0 model [2], and our enhancements do not involve complicated operations except finding the most significant set bit, which can be also computed in AC^0 [4]. The missing part is the wavelet tree, for which we used the implementation of Belazzougui and Navarro [3, Thm. 4.1]. We think that most bit operations used by this implementation are portable, and the used multiplications are not of general type, but a broadword broadcast operation, i.e., storing $\lfloor w/b \rfloor$ copies of a bit string of length b in a machine word of w bits. However, they rely on a monotone minimum perfect hash function, for which we do not know whether there is an existing alternative solution (even allowing a slight increase in the space complexity but still within $\sigma\lg\ell_{\max}(1+o(1))$ bits) in AC^0.

Another open question concerns the possibility of applying the presented technique in an adaptive (sometimes also called dynamic) Huffman coding scheme. There exist efficient dynamic fusion tree and multi-ary wavelet tree implementations, but it is unclear to us whether we can meet those costs (at least in the amortized sense) involved in maintaining the code tree dynamically. Also, we are curious whether, for a small enough alphabet, a canonical code can encode and decode k ($k > 1$) symbols at a time (ideally, $k = \Theta(w/\ell_{\max})$ or $k = \Theta(\log n/\ell_{\max})$, where the denominator could be changed to $\log\sigma$, if focusing on the average case) without a major increase in the required space.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is funded by the JSPS KAKENHI Grant Numbers `JP21K17701` and `JP21H05847` (DK) and by the Faculty of EECCE, Lodz University of Technology, as a statutory activity (SG). We thank the anonymous reviewers for detailed and constructive comments, in particular for finding a flaw in the proof of Lemma 4.2.

Appendix A. Fusion tree augmentation

Here, we provide an alternative solution to Section 2.2.1 without the need to let P store pairs of codewords with their respective lengths. Instead, the idea is to augment P with additional information to allow queries needed for simulating First. For that, we follow Pătraşcu and Thorup [29, Section IV.], who presented a solution for a dynamic fusion tree that can answer the following queries, where $\mathcal{S}$ is the set of keys defined in Section 2.2.

- $\mathit{rank}(X)$ returns $|\{Y \in \mathcal{S} : Y \leq X\}|$ and
- $\mathit{select}(i)$ returns $Y \in \mathcal{S}$ with $\mathit{rank}(Y) = i$.

Since P is a static data structure, adding these operations to P is rather simple: The idea is to augment each fusion tree node with an integer array storing the prefix-sums of subtree sizes. Specifically, let v be a fusion tree node having w^c children, for a fixed (but initially selectable) positive constant $c < 1$. Then we augment v with an integer array P_v of length w^c such that $P_v[i]$ is the sum of the sizes of the subtrees rooted at the preceding siblings of v's i-th child. With P_v we can answer $\mathit{rank}(x)$ via $\mathsf{P}.\mathit{pred}(x)$, which is solved by traversing the fusion tree P in a top-down manner. Starting with a counter c of the rank initially set to zero, on visiting the i-th child of a node v, we increment c by $P_v[i]$. On finding $\mathit{pred}(x)$ we return $c+1$ (+1 because the predecessor itself needs to be counted).

To answer a *select* query, we store the content of each P_v in a (separate) fusion tree F_v to support a *rank* query on the prefix-sum values stored in P_v. Consequently, a node of P stores not only P_v but also a fusion tree built upon P_v. The algorithm works again in a top-down manner, but uses F_v for navigation: For answering *select*(i), suppose we are at a node v, and suppose that $F_v.\mathit{rank}(i)$ gives us j. Then we exchange i with $i - P_v[j]$. Now if i has been decremented to zero, we are done since the answer is the j-th child of v. Otherwise, we descend to the j-th child of v and recurse.

The fusion trees F_v as well as the prefix-sums P_v take asymptotically the same space as its respective fusion tree node v.[7] Regarding the time, F_v and P_v answer a *rank* and an access query in $\mathcal{O}(\log_w w^c) = \mathcal{O}(1)$ time, and therefore, we can answer *rank* and *select* in the same time bounds as *pred*.

It is left to deal with depths having no leaves. For that, we add a bit vector $\mathsf{B_D}$ of length $\ell_{\max}$ with $\mathsf{B_D}[\ell] = 1$ if and only if depth ℓ has at least one leaf. For the latter solution, P only needs to take care of all depths in which leaves are present, i.e., P will return a depth ℓ' during a predecessor query, which we map to the correct depth with $\mathsf{B_D}.\mathit{select}_1[\ell']$, given that we endowed $\mathsf{B_D}$ with a select-support data structure in a precomputation step. In total, $\mathsf{B_D}$ with its select-support data structure takes $\ell_{\max} + o(\ell_{\max})$ bits of space, and answers a select query in constant time. We no longer need to store First since $\mathsf{First}[\ell] = \mathsf{P}.\mathit{select}(\ell)$, given there is a leaf of the Huffman tree with depth ℓ. Consequently, our approach shown in Section 4 works with this augmented fusion tree analogously.

References

[1] M. Aggarwal, A. Narayan, Efficient Huffman decoding, in: Proc. ICIP, 2000, pp. 936–939.

[2] A. Andersson, P.B. Miltersen, M. Thorup, Fusion trees can be implemented with AC^0 instructions only, Theor. Comput. Sci. 215 (1–2) (1999) 337–344, https://doi.org/10.1016/S0304-3975(98)00172-8.

[3] D. Belazzougui, G. Navarro, Optimal lower and upper bounds for representing sequences, ACM Trans. Algorithms 11 (4) (2015) 31, https://doi.org/10.1145/2629339.

[4] O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, O. Weimann, Towards optimal packed string matching, Theor. Comput. Sci. 525 (2014) 111–129, https://doi.org/10.1016/j.tcs.2013.06.013.

[5] J.L. Bentley, A.C. Yao, An almost optimal algorithm for unbounded searching, Inf. Process. Lett. 5 (3) (1976) 82–87, https://doi.org/10.1016/0020-0190(76)90071-5.

[6] M. Buro, On the maximum length of Huffman codes, Inf. Process. Lett. 45 (5) (1993) 219–223, https://doi.org/10.1016/0020-0190(93)90207-P.

[7] R.A. Chowdhury, M. Kaykobad, I. King, An efficient decoding technique for Huffman codes, Inf. Process. Lett. 81 (6) (2002) 305–308, https://doi.org/10.1016/S0020-0190(01)00243-5.

[8] F. Claude, G. Navarro, A. Ordóñez, The wavelet matrix: an efficient wavelet tree for large alphabets, Inf. Sci. 47 (2015) 15–32, https://doi.org/10.1016/j.is.2014.06.002.

[9] P. Elias, Universal codeword sets and representations of the integers, IEEE Trans. Inf. Theory 21 (2) (1975) 194–203, https://doi.org/10.1109/TIT.1975.1055349.

[10] A. Fariña, T. Gagie, G. Manzini, G. Navarro, A. Ordóñez, Efficient and compact representations of some non-canonical prefix-free codes, in: Proc. SPIRE, in: LNCS, vol. 9954, 2016, pp. 50–60.

[11] A. Fariña, T. Gagie, S. Grabowski, G. Manzini, G. Navarro, A. Ordóñez, Efficient and compact representations of some non-canonical prefix-free codes, arXiv:1605.06615, 2021.

[12] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. Syst. Sci. 47 (3) (1993) 424–436, https://doi.org/10.1016/0022-0000(93)90040-4.

[13] T. Gagie, G. Navarro, Y. Nekrich, A. Ordóñez, Efficient and compact representations of prefix codes, IEEE Trans. Inf. Theory 61 (9) (2015) 4999–5011, https://doi.org/10.1109/TIT.2015.2452252.

[14] R. Hashemian, Memory efficient and high-speed search Huffman coding, IEEE Trans. Commun. 43 (10) (1995) 2576–2581, https://doi.org/10.1109/26.469442.

[15] D.S. Hirschberg, D.A. Lelewer, Efficient decoding of prefix codes, Commun. ACM 33 (4) (1990) 449–459, https://doi.org/10.1145/77556.77566.

[16] D.A. Huffman, A method for the construction of minimum-redundancy codes, Proc. Inst. Radio Eng. 40 (9) (1952) 1098–1101, https://doi.org/10.1109/JRPROC.1952.273898.

[17] M. Karpinski, Y. Nekrich, A fast algorithm for adaptive prefix coding, Algorithmica 55 (1) (2009) 29–41, https://doi.org/10.1007/s00453-007-9140-4.

[18] G.O.H. Katona, T.O.H. Nemetz, Huffman codes and self-information, IEEE Trans. Inf. Theory 22 (3) (1976) 337–340, https://doi.org/10.1109/TIT.1976.1055554.

[19] S.T. Klein, Space- and time-efficient decoding with canonical Huffman trees, in: Proc. CPM, in: LNCS, vol. 1264, 1997, pp. 65–75.

[20] S.T. Klein, Skeleton trees for the efficient decoding of Huffman encoded texts, Inf. Retr. 3 (1) (2000) 7–23, https://doi.org/10.1023/A:1009910017828.

[21] D. Kosolobov, O. Merkurev, Optimal skeleton Huffman trees revisited, in: Proc. CSR, in: LNCS, vol. 12159, 2020, pp. 276–288.

[22] L.L. Larmore, D.S. Hirschberg, A fast algorithm for optimal length-limited Huffman codes, J. ACM 37 (3) (1990) 464–473, https://doi.org/10.1145/79147.79150.

[23] M. Liddell, A. Moffat, Decoding prefix codes, Softw. Pract. Exp. 36 (15) (2006) 1687–1710, https://doi.org/10.1002/spe.741.

[24] M.F. Mansour, Efficient Huffman decoding with table lookup, in: Proc. ICASSP, 2007, pp. 53–56.

[25] A. Moffat, Huffman coding, ACM Comput. Surv. 52 (4) (2019) 85, https://doi.org/10.1145/3342555.

[26] A. Moffat, A. Turpin, On the implementation of minimum redundancy prefix codes, IEEE Trans. Commun. 45 (10) (1997) 1200–1207, https://doi.org/10.1109/26.634683.

[27] G. Navarro, Compact Data Structures – A Practical Approach, Cambridge University Press, ISBN 978-1-107-15238-0, 2016.

[28] Y. Nekrich, Decoding of canonical Huffman codes with look-up tables, in: Proc. DCC, IEEE Computer Society, 2000, p. 566.

[29] M. Pătraşcu, M. Thorup, Dynamic integer sets with optimal rank, select, and predecessor search, in: Proc. FOCS, 2014, pp. 166–175.

[30] E.S. Schwartz, B. Kallick, Generating a canonical prefix encoding, Commun. ACM 7 (3) (1964) 166–169, https://doi.org/10.1145/363958.363991.

[31] I.H. Witten, R.M. Neal, J.G. Cleary, Arithmetic coding for data compression, Commun. ACM 30 (6) (1987) 520–540, https://doi.org/10.1145/214762.214771.

---

[7] Since our tree P has a branching factor of w^c, there are $\mathcal{O}(m/w^c)$ leaves and $\mathcal{O}(m/w^{2c})$ internal nodes. Since an internal node takes w^{1+c} bits, we need m/w^c bits for all internal nodes. Also, a leaf λ does not need to store F_λ and P_λ since $P_\lambda[i] = i$.