Skeleton Trees for the Efficient Decoding of Huffman Encoded Texts*

Shmuel T. Klein

Department of Mathematics and Computer Science
Bar Ilan University, Ramat-Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 535 3325
tomi@cs.biu.ac.il

Abstract: A new data structure is investigated, which allows fast decoding of texts encoded by canonical Huffman codes. The storage requirements are much lower than for conventional Huffman trees, $O(\log^2 n)$ for trees of depth $O(\log n)$, and decoding is faster, because a part of the bit-comparisons necessary for the decoding may be saved. Empirical results on large real-life distributions show a reduction of up to 50% and more in the number of bit operations. The basic idea is then generalized, yielding further savings.

*This is an extended version of a paper which has been presented at the *8th Annual Symposium on Combinatorial Pattern Matching* (CPM'97), and appeared in its proceedings, pp. 65–75.

1. Introduction

The importance and usefulness of Data Compression for Information Retrieval (IR) Systems is today well-established, and many authors have commented on it [1, 17, 31, 27]. Large full-text IR Systems are indeed voracious consumers of storage space realtive to the size of the raw textual database, because not only the text has to be kept, but also various auxiliary files like dictionaries and concordances, which are usually adjoined to the system to make the retrieval process efficient. Moreover, certain data structures, such as decoding tables or trees, have to be resident in RAM, so that large systems require more and more powerful machines. It is therefore quite natural that efforts have been made to compress the text and other necessary files, thereby reducing the demand for storage or RAM, or equivalently, for a fixed machine with given resources, effectively increasing the size of the data base that can still be handled efficiently.

Most of the popular compression methods are based on the works of Lempel and Ziv [29, 30], but these are adaptive methods which are not always suitable for IR applications. In the context of full-text retrieval, a large number of small passages is accessed simultaneously, e.g., when producing a KWIC (Key-Word-In-Context) index in response to a submitted query, and all these text fragments should be decodable, regardless of their exact location. When an adaptive coding method has been used, this would then force us to start the decoding at the beginning of the text or the logical block that contains the retrieved passage. So we would either decode much more than needed, which may imply increased processing time, or prepare *a priori* smaller blocks, which would cost us compression efficiency. In both cases, the advantage of using adaptive methods, which often yield better compression than static ones, may be lost.

Huffman coding [14] is still one of the best known and most popular static data compression methods. While for certain applications, such as data transmission over a communication channel, both coding and decoding ought to be fast, for other applications, like the IR scenario we focus on in this paper, compression and decompression are not symmetrical tasks. Compression is done only once, while building the system, whereas decompression is needed during the processing of every query and directly affects response time. There is thus a special interest in fast decoding techniques (see e.g., [15]).

The data structures needed for the decoding of a Huffman encoded file (a Huffman tree or lookup table) are generally considered negligible overhead relative to large texts. However, not all texts are large, and if Huffman coding is applied in connection with a Markov model [2], the required Huffman forest may become itself a storage problem. Moreover, the "alphabet" to be encoded is not necessarily small, and may, e.g., consist of all the different words in the text, so that Huffman trees with thousands and even millions of nodes are not uncommon [23]. We try, in this paper, to reduce the necessary internal memory space by devising efficient ways to encode these trees. In addition, the new suggested data structure also allows a speed-up of the decompression process, by reducing the number of necessary bit comparisons.

The manipulation of individual bits is indeed the main cause for the slow decoding of Huffman encoded text. A method based on large tables constructed in a pre-processing stage is suggested in [5], with the help of which the entire decoding process can be performed using only byte oriented commands (see also [26]). However, the internal memory required for the storage of these tables may be very large. Another possibility to avoid accessing individual bits is by using 256-ary instead of the optimal binary Huffman codes. This obviously reduces the compression efficiency, but de Moura et al. [6] report that the degradation is not significant.

In the next section, we recall the necessary definitions of canonical Huffman trees as they are used below. Section 3 presents the new suggested data structure and includes experimental results. In Section 4, the main idea is then extended, yielding yet smaller trees and even faster decoding.

2. Canonical Huffman codes

0	000
1	0010
2	0011
3	0100
4	01010
5	01011
6	01100
7	01101
8	011100
9	011101
10	011110
11	011111
12	100000
13	100001
14	100010
15	100011
16	1001000
17	1001001
18	1001010
19	1001011
...	...
29	1010101
30	1010110
31	10101110
32	10101111
33	10110000
...	...
61	11001100
62	11001101
63	110011100
64	110011101
...	...
124	111011001
125	111011010
126	1110110110
127	1110110111
...	...
198	1111111110
199	1111111111

Figure 1: *Canonical Huffman code for Zipf-200*

For a given probability distribution, there might be quite a large number of different Huffman trees, since interchanging the left and right subtrees of any internal node will result in a different tree whenever the two subtrees are different in structure, but the weighted average path length is not affected by such an interchange. There are often also other optimal trees, which cannot be obtained via Huffman's algorithm. One may thus choose one of the trees that has some additional properties. The preferred choice for many applications is the *canonical* tree, defined by Schwartz and Kallick [25], and recommended by many others (see, e.g., [15, 27]).

Denote by $(p_1, \ldots, p_n)$ the given probability distribution, where we assume that $p_1 \geq p_2 \geq \cdots \geq p_n$, and let ℓ_i be the length in bits of the codeword assigned by Huffman's procedure to the element with probability p_i, i.e., ℓ_i is the depth of the leaf corresponding to p_i in the Huffman tree. A tree is called canonical if, when scanning its leaves from left to right, they appear in non-decreasing order of their depth (or equivalently, in non-increasing order, as in [22]). The idea is that Huffman's algorithm is only used to generate the lengths $\{\ell_i\}$ of the codewords, rather than the codewords themselves; the latter are easily obtained as follows: the i-th codeword consists of the first ℓ_i bits immediately to the right of the "binary point" in the infinite binary expansion of $\sum_{j=1}^{i-1} 2^{-\ell_j}$, for $i = 1, \ldots, n$ [12]. Many properties of canonical codes are mentioned in [15, 3].

The following will be used as a running example in this paper. Consider the probability distribution implied by Zipf's law, defined by the weights $p_i = 1/(i\,H_n)$, for $1 \leq i \leq n$, where $H_n = \sum_{j=1}^{n}(1/j)$ is the n-th harmonic number. This law is believed to govern the distribution of the most common words in a large natural language text [28]. A canonical code can be represented by the string $\langle n_1, n_2, \ldots, n_k \rangle$, called a *source*, where k denotes, here and below, the length of the longest codeword (the depth of the tree), and n_i is the number of codewords of length i, $i = 1, \ldots, k$. The source corresponding to Zipf's distribution for $n = 200$ is $\langle 0, 0, 1, 3, 4, 8, 15, 32, 63, 74 \rangle$. The code is depicted in Figure 1.

We shall assume, for the ease of description, that the source has no "holes", i.e., there are no three integers $i < j < \ell$ such that $n_i \neq 0, n_\ell \neq 0$, but $n_j = 0$. This is true for many real-life distributions, and in particular for all the examples below. On the other hand, the distribution of one of the alphabets used for compressing a set of sparse bitmaps in [8] is $\langle 1,0,0,1,7,0,1,28,0,46,59,114 \rangle$. All the techniques suggested herein can be easily adapted to the general case using a vector $succ(i)$, giving for each codeword length i, the next larger codeword length j for which $n_j > 0$. But to make the exposition clearer, we shall suppress reference to $succ(i)$, since for all distributions without holes, $succ(i) = i + 1$.

One of the properties of canonical codes is that the set of codewords having the same length comprises the binary representations of consecutive integers. For example, in our case, the codewords of length 9 bits are the binary integers in the range from 110011100 to 111011010. This fact can be exploited to enable efficient decoding with relatively small overhead: once a codeword of ℓ bits is detected, one can get its relative index within the sequence of codewords of length ℓ by simple subtraction.

The following information is thus needed: let $m = \min\{i \mid n_i > 0\}$ be the length of the shortest codeword, and let $base(i)$ be the integer value of the first codeword of length i. We then have

$$\begin{aligned} base(m) &= 0 \\ base(i) &= 2\,(base(i-1) + n_{i-1}) \qquad \text{for } m < i \leq k. \end{aligned}$$

Let $B_s(k)$ denote the standard s-bit binary representation of the integer k (with leading zeros, if necessary). Then the j-th codeword of length i, for $j = 0, 1, \ldots, n_i - 1$, is $B_i(base(i) + j)$. Let $seq(i)$ be the sequential index of the first codeword of length i:

$$\begin{aligned} seq(m) &= 0 \\ seq(i) &= seq(i-1) + n_{i-1} \qquad \text{for } m < i \leq k. \end{aligned}$$

Suppose now that we have detected a codeword w of length ℓ. If $I(w)$ is the integer value of the binary string w (i.e., $w = B_\ell(I(w))$), then $I(w) - base(\ell)$ is the relative index of w within the block of codewords of length ℓ. Thus $seq(\ell) + I(w) - base(\ell)$ is the relative index of w within the full list of codewords. This can be rewritten as $I(w) - diff(\ell)$, for $diff(\ell) = base(\ell) - seq(\ell)$. Thus all one needs is the list of integers $diff(\ell)$. Table 1 gives the values of n_i, $base(i)$, $seq(i)$ and $diff(i)$ for our example.

i	n_i	$base(i)$	$seq(i)$	$diff(i)$
3	1	0	0	0
4	3	2	1	1
5	4	10	4	6
6	8	28	8	20
7	15	72	16	56
8	32	174	31	143
9	63	412	63	349
10	74	950	126	824

Table 1: *Decode values for canonical Huffman code for Zipf-200*

We suggest in the next section a new representation of canonical Huffman codes, which not only is space-efficient, but may also speed up the decoding process, by permitting, at times, the decoding of more than a single bit in one iteration.

3. Skeleton trees for fast decoding

The following small example, using the data above, shows how such savings are possible. Suppose that while decoding, we detect that the next codeword starts with 1101. This information should be enough to decide that the following codeword ought to be of length 9 bits. We should thus be able, after having detected the first 4 bits of this codeword, to read the following 5 bits as a block, without having to check after each bit if the end of a codeword has been reached. Our goal is to construct an efficient data-structure, that permits similar decisions as soon as they are possible. The fourth bit was the earliest possible in the above example, since there are also codewords of length 8 starting with 110.

3.1 Decoding with sk-trees

The suggested solution is a binary tree, called below an *sk-tree* (for skeleton-tree), the structure of which is induced by the underlying Huffman tree, but which has generally significantly fewer nodes. The tree will be traversed like a regular Huffman tree. That is, we start with a pointer to the root of the tree, and another pointer to the first bit of the encoded binary sequence. This sequence is scanned, and after having read a zero (resp., a 1), we proceed to the left (resp., right) child of the current node. In a regular Huffman tree, the leaves correspond to full codewords that have been scanned, so the decoding algorithm just outputs the corresponding item, resets the tree-pointer to the root and proceeds with scanning the binary string. In our case, however, we visit the tree only up to the depth necessary to identify the length of the current codeword. The leaves of the sk-tree then contain the lengths of the corresponding codewords.

{
  $tree\_pointer \longleftarrow root$
  $i \longleftarrow 1$
  $start \longleftarrow 1$
  while $i < length\_of\_string$
  {
    if $string[i] = 0$ $\quad tree\_pointer \longleftarrow left(tree\_pointer)$
    else $\quad tree\_pointer \longleftarrow right(tree\_pointer)$
    if $value(tree\_pointer) > 0$
    {
      $codeword \longleftarrow string[start \cdots (start + value(tree\_pointer) - 1)]$
      output $\longleftarrow table[I(codeword) - diff[value(tree\_pointer)]]$
      $tree\_pointer \longleftarrow root$
      $start \longleftarrow start + value(tree\_pointer)$
      $i \longleftarrow start$
    }
    else $\quad i \longleftarrow i + 1$
  }
}

Figure 2: *Decoding procedure using sk-tree*

The formal decoding process using an sk-tree is depicted in Figure 2. The variable *start* points to the index of the bit at the beginning of the current codeword in the encoded string, which is stored in the vector *string*[ ]. Each node of the sk-tree consists of three fields: a *left* and a *right* pointer, which are not null if the node is not a leaf, and a *value*-field, which is zero for internal nodes, but contains the length in bits of the current codeword, if the node is a leaf. In an actual implementation, we can use the fact that any internal node has either zero or two children, and store the *value*-field and the *right*-field in the same space, with *left* = *null* serving as flag for the use of the *right* pointer. The procedure also uses two tables: *table*[j], $0 \le j < n$, giving the j-th element (in non-increasing order of frequency) of the encoded alphabet; and *diff*[i] defined above, for i varying from m to k, that is from the length of the shortest to the length of the longest codeword.

The procedure passes from one level in the tree to the one below according to the bits of the encoded string. Once a leaf is reached, the rest of the current *codeword* can be read in one operation. Note that not all the bits of the input vector are individually scanned, which yields possible time savings.

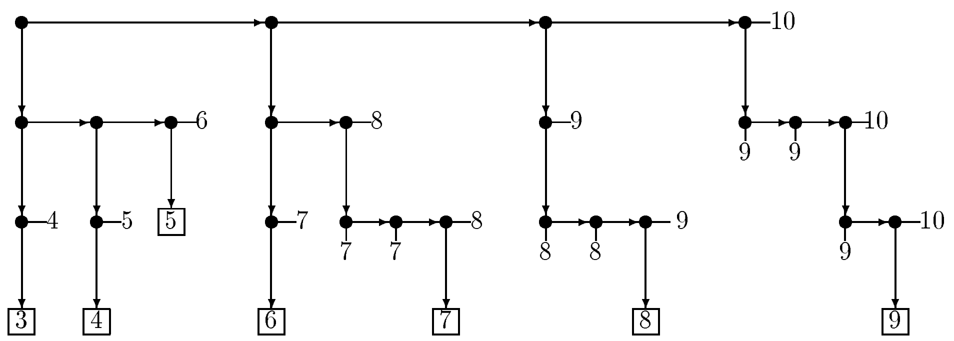


Figure 3: *sk-tree for Zipf-200 distribution*

Figure 3 shows the sk-tree corresponding to Zipf's distribution for $n = 200$. The tree is tilted by 45°, so that left (right) children are indicated by arrows pointing down (to the right). The framed leaves correspond to the last codewords of the indicated length. The sk-tree of our example consists of only 49 nodes, as opposed to 399 nodes of the original Huffman tree.

An idea similar to the sk-tree, but based on tables rather than on trees, has been suggested by Moffat and Turpin [22]. Instead of identifying roots of subtrees in which all codewords have the same depth, they essentially form a complete tree to a fixed depth no less than the depth of the code tree (by extending any shorter branches), and examine the code tree nodes at that depth to determine the minimum codeword length in each subsidiary subtree. To find the length of a codeword, a fixed-sized window of the compressed bitstream, considered as a binary value, is compared with left-justified base values in a sequence of hard-coded cascading if-statements. Each such comparison is equivalent to a transition to a left or right child of the sk-tree, and the replacement of bit comparisons by equivalent byte or word based comparisons is reminiscent of a mechanism suggested in [5].

3.2 Construction of sk-trees

While traversing a standard canonical Huffman tree to decode a given codeword, one may stop as soon as one gets to the root of any full subtree of depth h, for $h \geq 1$, i.e., a subtree of depth h that has 2^h leaves, since at this stage it is known that exactly h more bits are needed to complete the codeword. One way to look at sk-trees is therefore as standard Huffman trees from which all full subtrees of depth $h \geq 1$ have been pruned. A more direct and much more efficient construction is as follows.

The one-to-one correspondence between the codewords and the paths from the root to the leaves in a Huffman tree can be extended to define, for any binary string $S = s_1 \cdots s_e$, the path $P(S)$ induced by it in a tree with given root r_0. This path will consist of $e+1$ nodes r_i, $0 \leq i \leq e$, where for $i > 0$, r_i is the left (resp. right) child of r_{i-1}, if $s_i = 0$ (resp. if $s_i = 1$). For example, in Figure 3, $P(111)$ consists of the four nodes represented as bullets in the top line. The skeleton of the sk-tree will consist of the paths corresponding to the last codeword of every length. Let these codewords be denoted by L_i, $m \leq i \leq k$; they are, for our example, 000, 0100, 01101, 100011, etc. The idea is that $P(L_i)$ serves as "demarcation line": any node to the left (resp. right) of $P(L_i)$, i.e., a left (resp. right) child of one of the nodes in $P(L_i)$, corresponds to a prefix of a codeword with length $\leq i$ (resp. $> i$).

As a first approximation, the construction procedure thus takes the tree obtained by $\bigcup_{i=m}^{k-1} P(L_i)$ (there is clearly no need to include the longest codeword L_k, which is always a string of k 1's), and adjoins the missing children to turn it into a complete tree in which each internal node has both a left and a right child. The label on such a new leaf is set equal to the label of the closest leaf following it in an inorder traversal. In other words, when creating the path for L_i, one first follows a few nodes in the already existing tree, then one branches off creating new nodes; as to the labeling, the missing right child of any node in the path will be labeled $i+1$ (basing ourselves on the assumption that there are no holes), but only the missing left children of any *new* node in the path will be labeled i.

A closer look then implies the following refinement. Suppose a codeword L_i has a zero in its rightmost position, i.e., $L_i = \alpha 0$ for some string α of length $i-1$. Then the first codeword of length $i+1$ is $\alpha 10$. It follows that only when getting to the i-th bit one can decide if the length of the current codeword is i or $i+1$. But if L_i terminates in a string of 1's, $L_i = \beta 01^a$, with $a > 0$ and $|\beta| + a = i-1$, then the first codeword of length $i+1$ is $\beta 10^{a+1}$, so the length of the codeword can be deduced already after having read the bit following β. It follows that one does not always need the full string L_i in the sk-tree, but only its prefix up to and not including the rightmost zero. Let $L_i^* = \beta$ denote this prefix. The revised version of the above procedure starts with the tree obtained by $\bigcup_{i=m}^{k-1} P(L_i^*)$. The nodes of this tree are depicted as bullets in Figure 3. For each path $P(L_i^*)$ there is a leaf in the tree, and the left child of this leaf is the new terminal node, represented in Figure 3 by a box containing the number i. The additional leaves are then filled in as explained above.

3.3 Space complexity

To evaluate the size of the sk-tree, we count the number of nodes added by path $P(L_i^*)$, for $m \leq i < k$. Since the codewords in a canonical code, when ordered by their corresponding frequencies, are also alphabetically sorted, it suffices to compare L_i to L_{i-1}. Let $\gamma(m) =$

empty string, and for $i > m$, let $\gamma(i)$ be the longest common prefix of L_i and L_{i-1}, e.g., $\gamma(7)$ is the string 10 in our example. Then the number of nodes in the sk-tree is given by:

$$size = 2\left(\sum_{i=m}^{k-1} \max(0, |L_i^*| - |\gamma(i)|)\right) - 1, \tag{1}$$

since the summation alone is the number of internal nodes (the bullets in Figure 3).

The maximum function comes to prevent an extreme case in which the difference might be negative. For example, if $L_6 = 010001$ and $L_7 = 0101111$, then the longest common prefix is $\gamma(7) = 010$, but since we consider only the bits up to and not including the rightmost zero, we have $L_7^* = 01$. In this case, indeed, no new nodes are added for $P(L_7^*)$.

An immediate bound on the number of nodes in the sk-tree is $O(\min(n, k^2))$, since on the one hand, there are up to $k-1$ paths $P(L_i^*)$ of lengths $\leq k-2$, but on the other hand, it cannot exceed the number of nodes in the underlying Huffman tree, which is $2n-1$. To get a tighter bound, consider the nodes in the upper levels of the sk-tree belonging to the full binary tree F with $k-1$ leaves and having the same root as the sk-tree. The depth of F is $d = \lceil \log_2(k-1) \rceil$, and all its leaves are at level d or $d-1$. The tree F is the part of the sk-tree where some of the paths $P(L_i^*)$ must be overlapping, so we account for the nodes in F and for those below separately. There are at most $2k-1$ nodes in F; there are at most $k-1$ disjoint paths below it, with path $P(L_i^*)$ extending at most $i - 2 - \lfloor \log_2(k-1) \rfloor$ nodes below F, for $\log_2(k-1) < i \leq k$. This yields as bound for the number of nodes in the sk-tree:

$$2k + 2\left(\sum_{i=1}^{k-2-\lfloor \log_2(k-1) \rfloor} i\right) = 2k + (k - 2 - \lfloor \log_2(k-1) \rfloor)(k - 1 - \lfloor \log_2(k-1) \rfloor.$$

There are no savings in the worst case, e.g., when there is only one codeword of each length (except for the longest, for which there are always at least two). More generally, if the depth of the Huffman tree is $\Omega(n)$, the savings might not be significant. But such trees are optimal only for some very skewed distributions. In many applications, like for most distributions of characters or character pairs or words in most natural languages, the depth of the Huffman tree is $O(\log n)$, and for large n, even the constant c, if the depth is $c \log_2 n$, must be quite small. For suppose the Huffman tree has a leaf on depth d. Then by [16, Theorem 1], the probability of the element corresponding to this leaf is $p < 1/F_{d+1}$, where F_j is the j-th Fibonacci number, and we get from [18, Exercise 1.2.1–4], that $p < (1/\phi)^{d-1}$, where $\phi = (1+\sqrt{5})/2$ is the golden ratio. Thus if $d > c \log_2 n$, we have

$$p < \left(\frac{1}{\phi}\right)^{c \log_2 n} = n^{-c \log_2(1/\phi)} = n^{-0.693c}.$$

To give a numeric example, in Section 4 below one of the Huffman trees corresponds to the different words in English, with $n = 289,101$ leaves. The probability for a tree of this size to have a leaf at level $3 \log_2 n$ is less than 4.4×10^{-12}, meaning that such a word occurs only once every 4400 billion words; the existence of such a rare word then puts a lower limit on the size of the text, which in our case must be large enough to fill about 35,000 CD-Roms! For all the distributions given in Table 2 in the experiments below, the ratio of the depth of the Huffman tree to $\log_2 n$ is between 1.31 and 2.61. But even if the original Huffman tree would be deeper, it is sometimes convenient to impose an upper limit of $B = O(\log n)$ on the depth, which often implies only a negligible loss in compression efficiency [10]. In any case, given a logarithmic bound on the depth, the size of the sk-tree is about

$$\log n\ (\log n - \log\log n).$$

3.4 Time complexity

When decoding is based on a standard Huffman tree, the average number of comparisons per codeword is the sum, taken over all the leaves i, of the depth of i in the tree times the probability to get to i. A similar sum holds for sk-trees, with the difference that a leaf does not correspond to a single element, but to several consecutive codewords of the same length. Let w be the prefix of a codeword corresponding to a leaf of the sk-tree labeled ℓ, $\ell \geq |w|$, and denote $t = \ell - |w|$. Then the 2^t codewords corresponding to this leaf of the sk-tree are $w0^t, \ldots, w1^t$, and correspond, using the notations of Section 2, to indices in the range from $I(w0^t) - \mathit{diff}(\ell)$ to $I(w1^t) - \mathit{diff}(\ell)$. The average number of comparisons per codeword using the sk-tree can thus be evaluated as:

$$\sum_{i \in \{\text{leaves in sk-tree}\}} \left(d_i \sum_{j=I(w_i 0^{\ell_i - d_i}) - \mathit{diff}(\ell_i)}^{I(w_i 1^{\ell_i - d_i}) - \mathit{diff}(\ell_i)} \mathrm{Prob}(j) \right), \tag{2}$$

where w_i is the binary string corresponding to the leaf i, $d_i = |w_i|$ is the depth of i in the tree, ℓ_i is a shortcut for $\mathit{label}(i)$, and $\mathrm{Prob}(j)$ is the probability of the element with index j.

As an approximation, we assume that the probability of an element on level i in the tree is 2^{-i}. This corresponds to a *dyadic* probability distribution, where all the probabilities are integral powers of $\frac{1}{2}$. There cannot be too great a difference between the actual probability distribution and this dyadic one, since they both yield the same Huffman tree (see [20] for bounds on the "distance" between such distributions). Given this model, eqn. (2) becomes

$$\sum_{i \in \{\text{leaves in sk-tree}\}} \left(d_i\ 2^{-d_i} \right).$$

A similar sum, but taken over all the leaves of the original Huffman tree gives the average codeword length for a dyadic distribution. There are therefore large savings whenever the number of nodes in the sk-tree is much smaller than in the underlying full Huffman tree.

3.5 Experimental Results

To test the effectiveness of the use of sk-trees, the following real-life distributions were used. The data for French was collected from the *Trésor de la Langue Française*, a database of 680 MB of French language texts (115 million words) of the 17^{th}–20^{th} centuries [4]; for English, the source are 500 MB (87 million words) of the *Wall Street Journal* [24]; and for Hebrew, a part of the *Responsa Retrieval Project*, 100 MB of Hebrew and Aramaic texts (15 million words) written over the past ten centuries [7]. The first set of alphabets consists of the bigrams in the three languages (the source for English for this distribution was [13]); for the next set, the elements to be encoded are the different *words*, which yields very large "alphabets"; and the final set contains the distribution of trigrams in French. For completeness, the Zipf-200 distribution used in the above examples was also added.

Source		k	total number of elements	average codeword length	number of nodes in sk-tree	average number of comparisons	relative savings in # comparisons
Zipf–200		10	200	6.024	49	3.990	33.7%
bigrams	English	13	371	7.445	67	4.200	43.6%
	French	29	2192	7.784	285	4.620	40.6%
	Hebrew	24	743	8.037	127	4.183	48.0%
words	English	26	289101	11.202	425	5.726	48.9%
	French	27	439191	10.473	443	5.581	46.7%
	Hebrew	24	296933	13.033	345	5.694	56.3%
trigrams	French	28	25781	10.546	381	5.026	52.3%

Table 2: *Time and Space requirements for real-life distributions*

Table 2 displays the results. The first three columns give some statistics about the various distributions: the depth k of the Huffman tree, the size n of the encoded alphabet, and the weighted average length of a codeword, measured in bits, which equals the average number of comparisons if the standard Huffman tree is used. The next two columns bring the number of nodes in the sk-tree, as given in eqn. (1), and the average number of comparisons per codeword when decoding is based on the sk-tree, as given in eqn. (2). The final column shows the relative savings in the number of comparisons, measured in percent. We see that for large distributions, roughly half of the comparisons may be saved. Note that these savings are in spite of the fact that the high-probability symbols with short codewords have relatively few bits in common. The weighted average takes this into account: few bits are saved for the shorter codewords, and the savings are multiplied by higher probabilities; more bits are saved for the longer codewords, but even if their probabilities are very small, their large number have a cumulative effect. Note also that the cost of storing the sk-tree is only several percent of the cost for the full Huffman tree.

4. Reduced sk-trees

We now wish to explore what might be gained by pruning the sk-tree at some internal node: one would thus get to leaves at which it is not yet possible to deduce the length of the current codeword, but at which some partial information is already available. For example, in Figure 3, if the bits already processed were 111 (corresponding to the internal node in the rightmost upper corner), we know already that the length of the current codeword is either 9 or 10. We therefore need only one more comparison to know the exact length: concatenate the following seven bits with the three already processed to get a 10-bit string w; if the binary value of w is smaller than $base(10)$, the next codeword must be of length 9, otherwise it is of length 10. If we had used the original sk-tree as explained in the previous section, we would have had at least one more comparison, possibly even more, e.g., if the bits after 111 were 0110, we would have performed four more comparisons and still not know if the length is 9 or 10.

This reflection leads to the idea of a *reduced sk-tree*, which is obtained from the sk-tree by pruning some of its branches. On the one hand, this reduced tree is obviously smaller, on the other, as we saw, it may also decrease the number of comparisons. More formally, define for each node v of the sk-tree two values $lower(v)$ and $upper(v)$ by:

$$\begin{array}{ll} \text{if } v \text{ is a leaf} & lower(v) = upper(v) = value(v) \\ \text{if } v \text{ is an internal node} & lower(v) = lower(left(v)) \\ & upper(v) = upper(right(v)), \end{array}$$

that is, for each node v, the codewords corresponding to leaves of the sub-tree rooted by v have their lengths in the interval $[lower(v), upper(v)]$. In terms of our earlier notation we have $lower(root) = m$ and $upper(root) = k$. We define the reduced sk-tree as the smallest sub-tree of the sk-tree for which all the leaves w correspond to a range of at most two consecutive codeword lengths, i.e.,

$$upper(w) \leq lower(w) + 1. \tag{3}$$

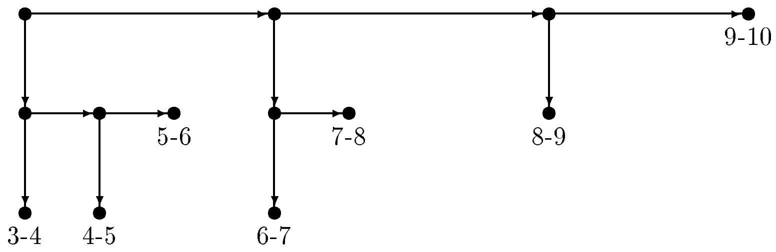


Figure 4: *Reduced sk-tree for Zipf-200 distribution*

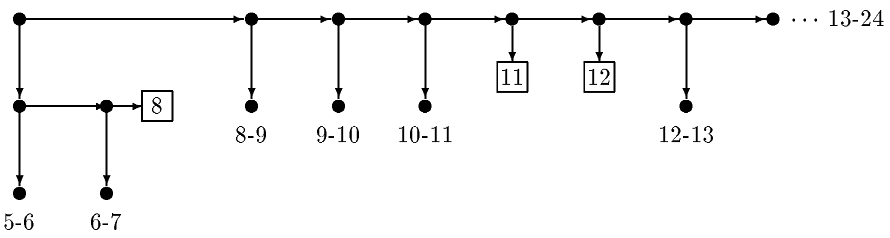
Figure 4 is the reduced sk-tree obtained from the sk-tree of Figure 3. Leaves are now also indicated as bullets, with the corresponding range underneath. Recall that the original Huffman tree had 399 nodes, and the sk-tree 49; in the reduced sk-tree we are left with only 13. Note that all the leaves of the original sk-tree are deleted, but also entire sub-trees. The nodes corresponding to the part of $P(L_i^*)$ which is not overlapping with $P(L_{i-1}^*)$ do all satisfy eqn. (3), but since we seek the minimal tree, for each such path, only the node highest up in the tree need be kept, so the rest of this branch and its offsprings are pruned.

A generalized view of both regular and reduced sk-trees would be as follows: consider a full canonical Huffman tree and assign to each node the values *lower* and *upper*. Delete now some of the nodes, starting at any leaf and proceeding to the parent nodes, until you get to the smallest tree for which every leaf w satisfies $lower(w) = upper(w)$, i.e., all the corresponding codewords have the same length; this is the sk-tree. If the process is continued and more nodes are deleted until the codewords corresponding to the new leaves have lengths i or $i+1$ for some i, we get the reduced sk-tree. We henceforth adopt the notation sk_1-tree and sk_2-tree for the original and the reduced sk-trees, respectively, the subscript referring to the maximal size of the set of codeword-lengths associated with each of the leaves of the tree.

We cannot use equality in eqn. (3), which would impose a range of exactly two codeword lengths for each leaf of the sk_2-tree. In the example of Figure 4 all the leaves do have equality

5	00000
6	010001
7	0101111
8	10011101
9	110010011
10	1110100101
11	11110111111
12	111111010101
13	1111111011111
	...

Last elements of codeword blocks



Corresponding sk_2-tree

Figure 5: *Example of sk_2-tree with special leaves*

in eqn. (3), but for other examples, leaves may exist, the parent nodes of which correspond already to ranges of 3 or more codeword lengths. In this case, the original leaf of the sk_1-tree must be kept. Let us call such leaves in the sk_2-tree *special* leaves. For the example distributions above, special leaves exist only for the distribution of the Hebrew bigrams, the first few elements of the source being $\langle 0, 0, 0, 0, 1, 16, 12, 62, 88, 126, 116, 86, \ldots \rangle$. In the left part of Figure 5, the last codewords L_i for the codeword lengths up to 13 are listed, and the right part of Figure 5 is the corresponding section of the sk_2-tree. The special leaves w are indicated as rectangles, containing the value $lower(w)$ (which is equal to $upper(w)$), the other leaves are depicted as bullets as above.

For example, we see that only codewords of length 8 can have the prefix 011, but the parent node of the corresponding leaf is associated with the prefix 01, which may be extended to codewords of lengths 6, 7 or 8. Similarly, a prefix 11110 implies the codeword length 11, but 1111 is the prefix of codewords of lengths 11 to 24.

The decoding procedure for sk_2-trees is similar to that of the sk_1-trees given in Figure 2, and only the if-block has to be replaced by the one in Figure 6. We now use a $flag$ field for each leaf w, with $flag(w) = 0$ if w is a special leaf, and $flag(w) = 1$ otherwise. The *value* field of w stores $lower(w)$ if w is a leaf, and 0 if w is an internal node.

```
...
    if value (tree_pointer) > 0
    {
        len  ⟵  value (tree_pointer)
        codeword  ⟵  string [start ··· (start + len − 1)]
        if  flag (tree_pointer) = 1   AND   2 I(codeword) ≥ base(len + 1)
        {
            codeword  ⟵  string [start ··· (start + len)]
            len  ⟵  len + 1
        }
        output  ⟵  table [I(codeword) − diff [len]]
        tree_pointer  ⟵  root
        start  ⟵  start + len
        i  ⟵  start
    }
...
```

Figure 6: *Decoding using sk_2-tree*

When a leaf w is reached, the current codeword is initialized as having length $lower(w)$. This is the correct setting if w is a special leaf or if the next codeword has indeed length $lower(w)$. When w is not a special leaf ($flag(w) = 1$), we check if by appending a zero at the right end of the codeword, we get an integer value larger or equal to that of the first codeword of length $lower(w) + 1$. If so, we update the current codeword to include also the following bit.

The construction of sk$_2$-trees is similar to that of the underlying sk$_1$-tree. We again consider the paths of nodes $\bigcup_{i=m}^{k-1} P(L_i^*)$, but keep only those nodes that appear in at least two different paths; these are the internal nodes of the sk$_2$-tree. The leaves are then added by filling in the missing left and right children, some of which may be special leaves.

As to the space complexity of sk$_2$-trees, note that in principle, several special leaves may emanate from a single branch $P(L_i^*)$, which leaves the upper bound for the number of nodes at $O(\min(n, k^2))$ as for sk$_1$-trees. But in practice, special leaves are rare, because they appear only in the very particular case when $P(L_i^*)$ is entirely contained in either $P(L_{i-1}^*)$ or $P(L_{i+1}^*)$. In the former case, the special leaves are right children of nodes in $P(L_i^*)$, in the latter they are left children. For example, referring to the tree in Figure 5, $L_7^* = 01$ is a prefix of $L_6^* = 0100$ and generates a special leaf as a right child, whereas $L_{12}^* = 1111110101$ contains $L_{11}^* = 1111$ as a prefix and generates special leaves as left children. If for a given Huffman tree, there are no special leaves in its associated sk$_2$-tree, as was the case in all our examples beside the one of Figure 5, then the number of nodes is clearly $2(k - m) - 1$, because there is exactly one leaf for each range $[i, i + 1]$, $m \le i < k$ and the sk$_2$-tree is a complete tree, i.e., each internal node has exactly two children.

The sizes of the sk$_2$-trees for our earlier example distributions are listed in Table 3. As can be seen, even for huge Huffman trees with hundreds of thousands of nodes, this size is reduced to several tens, and there is a 70–90% reduction even relative to the sizes of the sk$_1$-trees.

Source		number of nodes in sk$_2$-tree	Savings rel. to sk$_1$-tree	average number of comparisons	Savings rel. to sk$_1$-tree
Zipf–200		13	73%	3.688	7.6%
bigrams	English	15	78%	3.444	18.0%
	French	47	84%	3.757	18.7%
	Hebrew	45	65%	3.537	15.4%
words	English	41	90%	4.842	15.4%
	French	41	91%	4.725	15.3%
	Hebrew	33	90%	4.715	17.2%
trigrams	French	43	89%	4.157	17.3%

Table 3: *Time and Space requirements for sk$_2$-trees*

To evaluate the average number of comparisons, we take a sum similar to eqn. (2) over all the leaves of the sk$_2$-tree. For the special leaves, the formula of eqn. (2) applies. For the others, let w be the prefix of the corresponding codeword, assume the leaf is labeled ℓ, $\ell > |w|$, and denote $t = \ell - |w|$. Then the codewords corresponding to this leaf of the sk$_2$-tree are $w0^t, \ldots, w1^{t+1}$. The first few of them are of length ℓ and the following ones of length

$\ell+1$. The exact cutoff point is not important, as the codewords correspond to the consecutive indices in the range from $I(w0^t) - \mathit{diff}(\ell)$ to $I(w1^{t+1}) - \mathit{diff}(\ell+1)$. The probability of each of these codewords should be multiplied by the number of necessary comparisons to detect them, which is $|w|+1$, since we need an additional comparison to decide if the length is ℓ or $\ell+1$. This yields, using the same notations as for eqn. (2), the following formula:

$$\sum_{i \in \{\text{leaves in sk-tree}\}} \left(d_i\,(1 - flag(i)) \sum_{j=I(w_i 0^{\ell_i - d_i}) - \mathit{diff}(\ell_i)}^{I(w_i 1^{\ell_i - d_i}) - \mathit{diff}(\ell_i)} \mathrm{Prob}(j) \right.$$
$$\left. + \quad (d_i+1) flag(i) \sum_{j=I(w_i 0^{\ell_i - d_i}) - \mathit{diff}(\ell_i)}^{I(w_i 1^{\ell_i - d_i + 1}) - \mathit{diff}(\ell_i + 1)} \mathrm{Prob}(j) \right).$$

Table 3 gives the resulting averages for our examples. For the real-life examples, they give a reduction of 50–64% relative to the regular Huffman decoding algorithm, and of 15–19% relative to the algorithm using sk_1-trees.

5. Final remarks

If pruning the skeleton tree turned out to be profitable in terms of both time and space, shouldn't we climb up even higher and define sk_d-trees accordingly, for $d > 2$?

We can associate a value *range-size* with each node v of the sk_1-tree, giving the size of the set of the corresponding codeword lengths. The leaves of the sk_1-tree have all *range-size* $= 1$, those of the sk_2-tree have *range-size* ≤ 2. Consider a path that starts at any leaf of the sk_1-tree and moves through parent pointers towards the root. The *range-size* values of the nodes in this path form a non-decreasing sequence, the first value being 1, followed possibly by several 2's, etc. Fixing, for all such paths, the last node with value 2 (if it exists) as a new leaf yields the sk_2-tree. Similarly, proceeding even further up to the last node with value 3 would yield an sk_3-tree, etc.

However, the savings incurred by passing from the sk_1-tree to the sk_2-tree were caused by the fact that several consecutive nodes on these paths had the *range-size* value 2, so that the new leaves were several levels higher, and accordingly several comparisons could be saved. But if the parent node of a node with *range-size* value 3 has also *range-size* value 3, the other child of this parent node must have *range-size* value 1, which means that it is a special leaf. We argued earlier that such cases are rare. Therefore, whenever no special leaf is involved, passing from the lowest node with *range-size* value 2 to the lowest node with *range-size* value 3 would just let us climb one level and save one comparison. On the other hand, we need now an additional comparison within the range of 3 values, so that in all these cases, nothing is gained.

Of course, for the price of two additional comparisons, we could process, using binary search, ranges of size 4 and not just 3. More generally, we need only r additional comparisons after reaching a leaf of an sk_{2^r}-tree. Pushing this idea to its extreme with $r = \lceil \log_2 k \rceil$, there would be no skeleton-tree at all, and we would find the correct length of a codeword using a sequence of binary search steps in the list of first (or last) codewords for every codeword length, as suggested in [22]. But with standard binary search, the search in a code with maximal codeword length k takes exactly $\lceil \log_2 k \rceil$ comparisons, which would be 4 or 5 for our example distributions. Note that while the average number of comparisons with sk_1-trees is above that threshold for all the examples (Table 2), all the corresponding values for the sk_2-trees are below it (Table 3). It does therefore not seem necessarily worthwhile to pass to sk_d-trees, for $d > 2$.

Moffat and Turpin [22] further suggest to use a *biased* binary search, since the probability distribution of the codeword lengths is itself very skewed. For the first few bits of any codeword, this approaches then a linear search. The skeleton-trees introduced in this paper are a convenient data structure to perform a similar search efficiently.

References

[1] Bell T.C., Moffat A., Nevill-Manning C.G., Witten I.H., Zobel J., Data compression in full-text retrieval systems, *Journal ASIS* **44** (1993) 508–531.

[2] Bookstein A., Klein S.T., Compression, Information Theory and Grammars: A Unified Approach, *ACM Trans. on Information Systems* **8** (1990) 27–49.

[3] Bookstein A., Klein S.T., Is Huffman coding dead?, *Computing* **50** (1993) 279–296.

[4] Bookstein A., Klein S.T., Ziff D.A., A systematic approach to compressing a full text retrieval system, *Information Processing & Management* **28** (1992) 795–806.

[5] Choueka Y., Klein S.T., Perl Y., Efficient Variants of Huffman Codes in High Level Languages, *Proc. 8-th ACM-SIGIR Conf.,* Montreal (1985) 122–130.

[6] de Moura E.S., Navarro G., Ziviani N., Baeza-Yates R., Fast searching on compressed text allowing errors, *Proc. 21-st ACM-SIGIR Conf.,* Melbourne, Australia (1998) 295–306.

[7] Fraenkel A.S., All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, *Jurimetrics J.* **16** (1976) 149–156.

[8] Fraenkel A.S., Klein S.T., Novel Compression of Sparse Bit-Strings, in *Combinatorial Algorithms on Words,* NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 169–183.

[9] Fraenkel A.S., Klein S.T., Bidirectional Huffman Coding, *The Computer Journal* **33** (1990) 296–307.

[10] Fraenkel A.S., Klein S.T., Bounding the Depth of Search Trees, *The Computer Journal* **36** (1993) 668–678.

[11] Ferguson T.J., Rabinowitz J.H., Self-synchronizing Huffman codes, *IEEE Trans. on Information Theory,* **IT-30** (1984) 687–693.

[12] Gilbert E.N., Moore E.F., Variable-length binary encodings, *The Bell System Technical Journal* **38** (1959) 933–968.

[13] Heaps H.S., *Information Retrieval, Computational and Theoretical Aspects*, Academic Press, New York (1978).

[14] Huffman D., A method for the construction of minimum redundancy codes, *Proc. of the IRE* **40** (1952) 1098–1101.

[15] Hirschberg D.S., Lelewer D.A., Efficient decoding of prefix codes, *Comm. of the ACM* **33** (1990) 449–459.

[16] Katona G.H.O., Nemetz T.O.H., Huffman codes and self-information, *IEEE Trans. on Inf. Th.* **IT–11** (1965) 284–292.

[17] Klein S.T., Bookstein A., Deerwester S., Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations, *ACM Trans. on Information Systems* **7** (1989) 230–245.

[18] Knuth D.E., *The Art of Computer Programming, Vol* **I**, *Fundamental Algorithms,* Addison-Wesley, Reading, MA (1973).

[19] Lelewer D.A., Hirschberg D.S., Data compression, *ACM Computing Surveys* **19** (1987) 261–296.

[20] Longo G., Galasso G., An application of informational divergence to Huffman codes, *IEEE Trans. on Inf. Th.* **IT–28** (1982) 36–43.

[21] Moffat A., Bell T., In-situ generation of compressed inverted files, *J. ASIS* **46** (1995) 537–550.

[22] Moffat A., Turpin A., On the implementation of minimum redundancy prefix codes, *IEEE Trans. on Communications* **45** (1997) 1200–1207.

[23] Moffat A., Turpin A., Katajainen J., Space-efficient construction of optimal prefix codes, *Proc. Data Compression Conference DCC–95*, Snowbird, Utah (1995) 192–201.

[24] Moffat A., Zobel J., Sharman N., Text compression for dynamic document databases, *IEEE Transactions on Knowledge and Data Engineering* **9** (1997) 302–313.

[25] Schwartz E.S., Kallick B., Generating a canonical prefix encoding, *Comm. of the ACM* **7** (1964) 166–169.

[26] Sieminski, A., Fast decoding of the Huffman codes, *Information Processing Letters* **26** (1988) 237–241.

[27] Witten I.H., Moffat A., Bell T.C., *Managing Gigabytes: Compressing and Indexing Documents and Images,* Van Nostrand Reinhold, New York (1994).

[28] Zipf G.K., *The Psycho-Biology of Language*, Boston, Houghton (1935).

[29] Ziv J., Lempel A., A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th.* **IT–23** (1977) 337–343.

[30] Ziv J., Lempel A., Compression of individual sequences via variable-rate coding, *IEEE Trans. on Inf. Th.* **IT–24** (1978) 530–536.

[31] Zobel J., Moffat A., Adding compression to a full-text retrieval system, *Software — Practice & Experience*, **26** (1995) 891–903.