Optimal Skeleton and Reduced Huffman Trees*

Shmuel T. Klein[a], Jakub Radoszewski[b], Tamar C. Serebro[c], Dana Shapira[c]

[a]*Department of Computer Science, Bar Ilan University, Ramat Gan, Israel*
tomi@cs.biu.ac.il

[b]*Institute of Informatics, University of Warsaw, Poland*
jrad@mimuw.edu.pl

[c]*Computer Science Department, Ariel University, Ariel, Israel*
ytserebro@gmail.com, shapird@ariel.ac.il

Abstract

A skeleton Huffman tree is a Huffman tree from which all full subtrees of depth $h \geq 1$ have been pruned. Skeleton Huffman trees are used to save storage and enhance processing time in several applications such as decoding, compressed pattern matching and wavelet trees for random access. A reduced skeleton tree prunes the skeleton Huffman tree further to an even smaller tree. The resulting more compact trees can be used to further enhance the time and space complexities of the corresponding algorithms. However, it is shown that the straightforward ways of basing the constructions of a skeleton tree as well as that of a reduced skeleton tree on a canonical Huffman tree does not necessarily yield the least number of nodes. New algorithms for achieving such trees are given.

Keywords: Data compression, Huffman tree, skeleton tree, reduced skeleton.

1. Introduction

One of the most popular static data compression methods is still Huffman coding [10], even more than sixty years after its invention. A Huffman code is a minimum redundancy code, subject to the constraint that each codeword is composed of an integral number of bits. Given are an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ and a probability distribution $P = \{p_1, \ldots, p_n\}$ for the occurrences of its characters. Huffman's algorithm assigns lengths $\{\ell_1, \ldots, \ell_n\}$

*This is an extended version of a paper that has been presented at the 24th International Symposium on String Processing and Information Retrieval (SPIRE) in 2017, and appeared in its proceedings [14].

to the codewords, so that the average codeword length $\sum_{i=1}^{n} p_i \ell_i$ is minimized. The algorithm for the construction of the code repeatedly combines the two smallest probabilities and may be implemented in time $O(n \log n)$. A useful way to represent the code is by means of a binary tree called a *Huffman tree*. The leaves of the tree are associated with the elements of the alphabet. Edges in the tree pointing to a left or right child are labeled by 0 or 1, respectively, and the concatenation of the labels on the path from the root to a given leaf yields the corresponding codeword. In a more general setting, integer frequencies or even arbitrary positive numbers called *weights* $W = \{w_1, \ldots, w_n\}$ are used instead of probabilities, and it is the weighted average $\sum_{i=1}^{n} w_i \ell_i$ that is minimized. The algorithm remains the same. A tree minimizing this sum is called *optimal*; Huffman's method produces optimal trees, but not all optimal trees can be obtained directly by the Huffman algorithm.

It should be noted that the set of lengths $\{\ell_1, \ldots, \ell_n\}$ produced by Huffman's algorithm for a given distribution $\{w_1, \ldots, w_n\}$ is not necessarily unique. In fact, there are distributions for which the number of different such lengths-sets might be exponential [8].

A data structure called a *skeleton tree*, or *sk-tree* for short, has been introduced in [12], which is especially suited for fast decoding of Huffman encoded texts. An sk-tree is a Huffman tree in which all disjoint full subtrees have been replaced by their respective roots (precise definitions are given below). The storage requirements of sk-trees are much lower than those of traditional Huffman trees. The latter have $2n-1$ nodes, whereas the former need only $O(\log^2 n)$ nodes for trees of depth $O(\log n)$. The motivation for the definition of an sk-tree is to accelerate the decoding of a file compressed by means of a Huffman code. This is achieved by allowing the processing of the compressed file, one bit at a time, until a leaf of the sk-tree is reached, where the length of the current codeword w is already determined. This will often be the case before having read all the bits of w. Then several bits, from the one following the current position to the end of the codeword w, are processed in a single operation. Decoding may be faster since a part of the bit-comparisons and manipulations necessary for the conventional Huffman decoding may be saved. Empirical results on large real-life distributions show an average reduction of up to half and more in the number of bit operations [12].

There are several applications for which Huffman trees may be replaced by sk-trees in order to speed up processing time and/or save space, for example, to accelerate compressed pattern matching, as shown in [22]. Another application for which sk-trees are used to improve the time and space com-

plexities is *wavelet trees*. A wavelet tree (WT), suggested by Grossi et al. [9], is a data structure which reorders the bits of the compressed file into an alternative form, thereby enabling direct access, as well as other efficient operations. WTs can be defined for any prefix code, and the tree structure associated with this code is inherited by the WT.

The internal nodes of the WT are annotated with bitmaps. The root of the WT holds the bitmap obtained by concatenating the first bit of each of the sequence of codewords in the order they appear in the compressed text. The left and right children of the root hold, respectively, the bitmaps obtained by concatenating, again in the given order, the second bit of each of the codewords starting with 0 and starting with 1, respectively. This process is repeated similarly on the next levels: the grand-children of the root hold the bitmaps obtained by concatenating the third bit of the sequence of codewords starting, respectively, with 00, 01, 10 or 11, if they exist at all, etc.

Various manipulations on the bitmaps of the WT are based on fast implementations of operations known as rank and select. These are defined for any bit $b \in \{0, 1\}$ as

$\mathsf{rank}_b(B, i)$ – number of occurrences of b in B up to and including position i; and

$\mathsf{select}_b(B, i)$ – position of the ith occurrence of b in B.

Efficient implementations for rank and select are due to Jacobson [11], Raman et al. [19], Okanohara and Sadakane [18], Barbay et al. [1] and Navarro and Providel [17], to list only a few. WTs can be seen as extensions of rank and select operations to a general alphabet.

Recently, Baruch et al. [2] suggested to replace a Huffman shaped WT by a skeleton tree shaped WT in order to support faster random access and save storage, at the price of less effective rank and select operations. The general idea is to apply some pruning strategy on the internal nodes of the WTs, so that the overhead of the additional storage, used by the data structures for processing the stored bitmaps, is reduced. Moreover, the average path length corresponding to the codewords is also decreased, and so is also the average time spent for traversing the paths from the root to the desired leaf, which is the basic processing component used to evaluate random access. The suggestion of [2], combining wavelet with skeleton trees has been extended in [5], where it was empirically shown that reordering the sk-tree may enhance the direct access via WTs. The pruning idea was

also applied on WTs corresponding to Fibonacci codes [15], rather than to Huffman codes based WTs.

The current paper is organized as follows. We recall the details of sk-trees in Section 2. In Section 3, we develop our method for designing enhanced sk-trees with a minimal number of nodes, and prove its optimality. Section 4 deals with reduced skeleton trees. Finally, Section 5 presents some experimental results.

2. Skeleton Trees

Since the codes we consider herein consist of codewords corresponding to the leaves of trees, all the codes are prefix-free. The trees are binary trees, although one could easily extend the ideas to more general k-ary trees with $k > 2$. The *level* (also called *depth*) of a node v in a given tree is the number of edges one has to traverse to get from the root to v. The *height* of a (sub)tree is the largest depth of one of its nodes. A *full* tree is a tree all of whose leaves are on the same level, as in Figure 3(a). A *complete* tree is a tree in which every internal node has exactly two children, a left and a right one[1]. A tree all of whose leaves are on two adjacent levels will be called *almost full*. The trees in Figure 1 are complete; they are not full, but they are almost full. The tree in Figure 2(a) is not almost full, but its subtree rooted at the node labeled 3 is almost full. A compact way to describe a complete tree is by means of its *quantized source* $\langle n_1, n_2, \ldots, n_k \rangle$ or *q-source* for short, as defined in [6], where n_i is the number of codewords of length i, for $1 \leq i \leq k$, and k is the longest codeword length. Note that $\sum_{i=1}^{k} n_i = n$. The q-source does not uniquely identify a given tree, for example, the q-source of both trees in Figure 1 is $\langle 0, 2, 4 \rangle$, as for both there are no codewords of length 1, two codewords of length 2, and four codewords of length 3. Nevertheless, it is convenient to use the q-source for Huffman trees, since their shape is generally of no matter, and all trees belonging to same q-source share the same codeword lengths.

A well-known property of complete trees is that they satisfy the *Kraft equality*, see, e.g., [13, Chapter 4]: if $\ell_1, \ell_2, \ldots, \ell_n$ are the lengths of the

[1] We are aware of the fact that there is, unfortunately, no consensus for these definitions, and that some authors prefer to invert them, e.g., [4]. We advocate, however, our definition, which can be found, among others, in [20], because of the obvious connection to complete codes.

codewords, or equivalently, the depths of the leaves in the tree, then

$$\sum_{j=1}^{n} 2^{-\ell_j} = \sum_{i=1}^{k} n_i 2^{-i} = 1. \tag{1}$$

In fact, the Kraft equality is often used as a characteristic of a complete code, in the sense that if a sequence of numbers $\ell_1, \ell_2, \ldots, \ell_n$ satisfies eq. (1), then a complete tree can be constructed whose leaves are at the given depths.

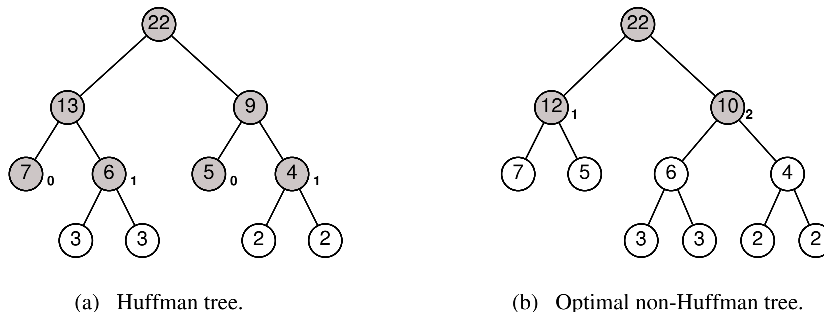
Given a complete binary tree T, *pruning* a subtree T' of T is a process that can be applied if T' is a full subtree. It consists of eliminating all the nodes of T' except its root. For example, in Figure 1(b), the rightmost subtree of height 2 could be pruned, leaving only its root (labeled 9) in the tree.

LEMMA 1: *Pruning a subtree from a complete binary tree results in a complete binary tree itself.*

PROOF: Actually, an even stronger property could be claimed, namely, that the replacement by its root of *any* subtree of a complete tree, not just for full subtrees, does not change the fact that all internal nodes still have two children. Thus the resulting tree is also complete. ∎

We may thus repeatedly prune subtrees from a given Huffman tree, and this will not affect the completeness of the remaining tree. Our goal is to prune several disjoint subtrees from optimal trees for some given weight distribution, so that the number of nodes remaining in the tree is minimal. An sk-tree is what remains after having pruned all the possible full subtrees of a complete binary tree. The size and shape of an sk-tree does, however, depend on the shape of the complete tree we started from. The trees in Figure 1 show different sk-trees derived from trees with different shapes, yet both optimal. As mentioned, Huffman trees are optimal, however, not all optimal trees can be attained directly via Huffman's algorithm. Consider for example the sequence of frequencies $\{7, 5, 3, 3, 2, 2\}$, yielding the Huffman tree in Figure 1(a). The tree in Figure 1(b) is still optimal as the codeword lengths remain the same as the ones in Figure 1(a), but it is not a Huffman tree: Huffman's algorithm would not combine the weights 6 and 4, since there is a weight 5 between them, and the algorithm adds the two smallest weights in each iteration.

For a given set of weights, there may be many equivalent Huffman trees, as it is possible to build up to 2^{n-1} different Huffman trees by interchanging the left and right subtrees of some internal node. The number of different



(a) Huffman tree.　(b) Optimal non-Huffman tree.

Figure 1: *Optimal trees for weights* $\{7, 5, 3, 3, 2, 2\}$.

Huffman trees can be even larger in case the set of weights W contains ties, or even when the sequence of weight sums, that are considered during Huffman's algorithm, contains ties.

A tree is called *canonical* [21] if, when scanning its leaves from left to right, they appear in non-decreasing order of their depth. Thus the tree in Figure 1(b) is canonical, but that in Figure 1(a) is not. Another way for defining canonicity is that when the codewords are sorted in decreasing order by the frequencies of their corresponding symbols, they are ordered lexicographically. This second definition is stronger, but the difference does not affect the discussion below. To build a canonical tree, Huffman's algorithm is only used for generating the optimal lengths ℓ_i of the codewords, and then the i-th codeword is defined as 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 $1 \le i \le n$ [7]. Turpin and Moffat [23] use canonical codes, with a symmetrically equivalent definition, to enhance decoding in Huffman encoded texts, so that more than a single bit can be processed in one machine operation.

Canonical trees gather all codewords of the same length consecutively, motivating the idea of pruning such trees. Although canonical trees reduce the number of different Huffman trees dramatically, there are still weight distributions for which even the canonical tree is not unique. For example, consider the frequencies $\{2, 1, 1, 1\}$, yielding the Huffman trees in Figure 2. Huffman's algorithm does not impose any strict order on the nodes in each level, nor any preference on connections between equal values and specific nodes. In the second step of the construction of the Huffman tree for our example, the tree has 3 leaves, with weights 1, 2 and 2. The value 2 thus appears both on level 1 (the level of the root being defined as 0) and on level 2. The third and last step of the construction is then to create two new

nodes with weight 1 each, and define them as being the children of one of the leaves with weight 2. Choosing the leaf on the lowest level yields the tree in Figure 2(a), choosing the leaf on level 1 yields the tree in Figure 2(b). Both choices give the weighted sum $2{\cdot}1+1{\cdot}2+1{\cdot}3+1{\cdot}3 = 1{\cdot}2+1{\cdot}2+1{\cdot}2+2{\cdot}2 = 10$, so both trees are Huffman trees and thus optimal.

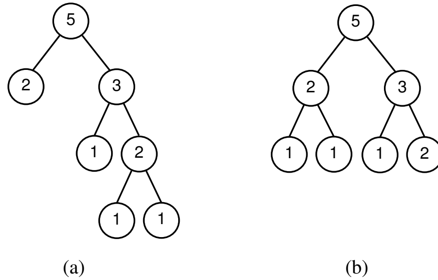


Figure 2: *Different optimal canonical trees for the frequencies* $\{2, 1, 1, 1\}$.

Figure 3 generalizes this example to show that weight distributions giving more than a single canonical Huffman tree may be found for every alphabet size. Consider the set of $n = 2^h$ frequencies $\{2, 1, \ldots, 1\}$, for $h \geq 2$. As in the previous example, there are two choices for splitting a node with weight 2 in the last step of the construction. While Figure 3(a) chooses to locate this node on level $h - 1$ of the tree, Figure 3(b) selects the only node with value 2 on level h.

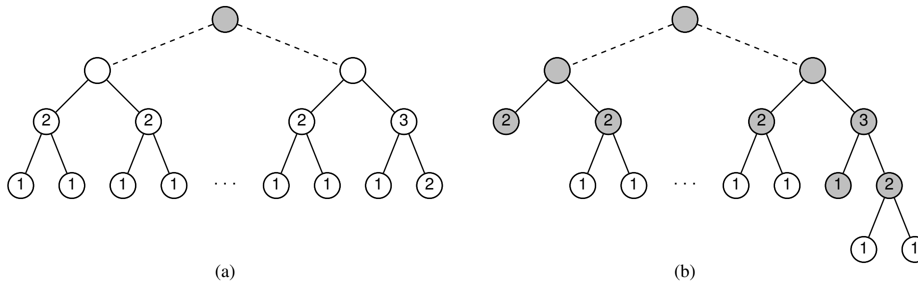


Figure 3: *Canonical trees are not unique.*

The original definition of the skeleton tree in [12] uses an underlying canonical Huffman tree, which here and below, refers to a canonical tree built for optimal codeword lengths for a given probability distribution, even if the specific canonical layout cannot be obtained directly by Huffman's algorithm, as, for example, the tree in Figure 1(b). Formally, an sk-tree is a

canonical Huffman tree from which all full subtrees of depth $h \geq 1$ have been pruned. Thus, a path from the root to a leaf of an sk-tree may correspond to a prefix of several codewords of the original Huffman tree. The prefix is the shortest necessary in order to identify the length of the current codeword. A leaf, v, of the sk-tree contains the height, $h(v)$, of the subtree that has been pruned ($h(v) = 0$ for leaves that were also leaves in the original Huffman tree), as well as a list of symbols belonging to that subtree. In the examples in Figure 1, as well as in the subsequent ones, we shall follow the convention that the nodes of the sk-trees appear in gray or black (the use of black nodes will be explained later). The values $h(v)$ appear in boldface to the right of the leaves of the sk-trees in Figure 1.

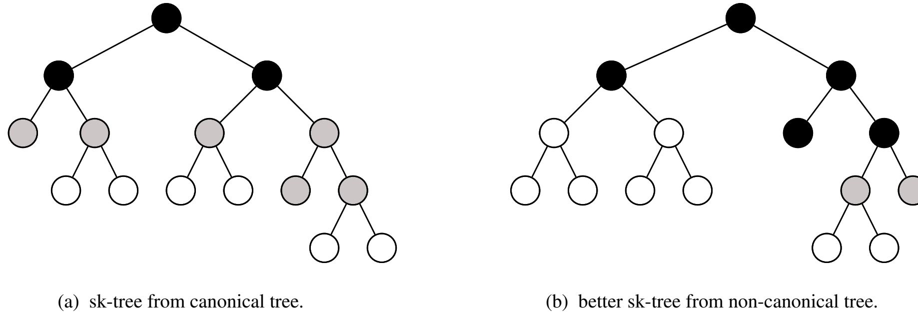
Figure 3 shows that different canonical trees constructed for the same set of weights may result in different sk-trees, as can be seen by inspecting the nodes highlighted in gray. Moreover, the example also shows that the difference in the number of nodes of different sk-trees for the same set of weights may not be bounded by a constant: the number of nodes in the sk-tree of Figure 3(b) is $2 + \sum_{i=0}^{h-1} 2^i = 2^h + 1 = n + 1$, whereas it is just 1 in the sk-tree of Figure 3(a), as the entire tree, except the root, may be pruned.

Since one of the goals of using sk-trees is saving space, it makes sense not to restrict the trees to be pruned only to those generated by Huffman's algorithm, but to consider the larger set of optimal trees for a given weight distribution. Figure 1(b) is an example that such a strategy may reduce the number of nodes in the sk-tree, from 7 to 3 in this example. Intuitively, canonical Huffman trees seem then to be a good choice in order to achieve smaller sk-trees, because the canonical structure collects all the leaves appearing on the same level together. However, we show in the following section that this intuition may be misleading.

3. Optimal Pruned Trees

The challenge is to find a way to produce the most compact pruned tree possible. An *optimal sk-tree* is defined as an sk-tree having the minimum number of nodes among all sk-trees obtained by pruning an optimal tree for a given weight distribution. If the canonical tree in Figure 4(a) is optimal for some given distribution, then so is the non-canonical tree in Figure 4(b); yet the sk-tree of the latter is smaller, by 2 nodes, than that of the former.

For a general example of a non-optimal sk-tree based on a canonical tree, consider the canonical Huffman tree for $n = 2^h$ codewords given in Figure 5, with $h \geq 3$. This tree is of height $h+1$, has its two rightmost leaves on level



(a) sk-tree from canonical tree. (b) better sk-tree from non-canonical tree.

Figure 4: *Optimal pruned tree.*

$h+1$, a single leftmost leaf on level $h-1$ and the remaining $n-3$ leaves on level h. Figure 4(a) is the particular case $h=3$. As a result, every node (except the two lowest) on the path from the root to the rightmost leaf, is the root of an asymmetric subtree which is not full: its right subtree is one level deeper than its left one. Similarly, the same is true also for the nodes on the path from the root to the leftmost leaf. In particular, the roots of the two largest full subtrees, which appear in the center of the figure, are not children of the same node. The number of nodes in the corresponding sk-tree is $4h-3$: four nodes on each level, except for that of the root (with a single node), and the first and the lowest levels, having two nodes each.

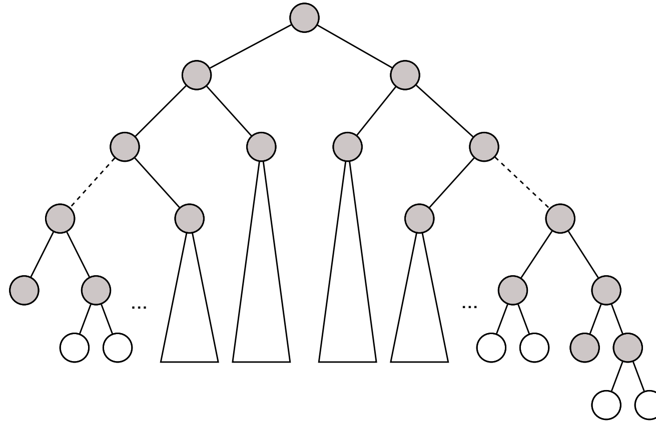


Figure 5: *A non-optimal sk-tree for $n \geq 8$ codewords.*

On the other hand, the tree given in Figure 6 has the same codeword lengths as that in Figure 5, but the locations of the nodes are different, resulting in a non-canonical tree. Nevertheless, there are fewer nodes in the

corresponding pruned tree. There are now two nodes on each level, except that of the root, which has only one node, for a total of $1+2(h-1)$ nodes. The difference between the number of nodes in the two sk-trees is thus $2(h-1)$, therefore this example shows that sk-trees of canonical Huffman trees might produce $\Omega(\log n)$ extra nodes as compared to pruning some non-canonical optimal tree. Therefore, not only does a canonical tree fail to provide the best possible sk-tree for n leaves, but moreover, the difference in the number of nodes between an sk-tree based on pruning a canonical tree and the best possible sk-tree might not even be bounded by a constant.

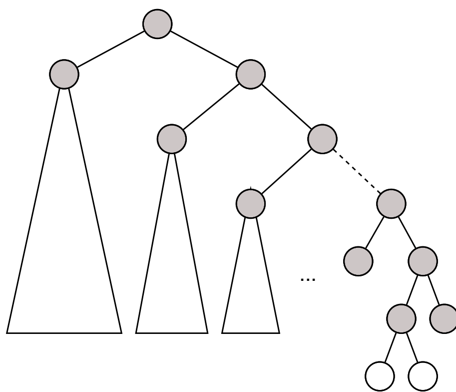


Figure 6: *Optimal pruned tree for the tree of Figure 5.*

Our search for an optimal sk-tree will be guided by the following reflexions. Since there might be no obvious connection between the shape of the optimal tree to start with and the fact that the corresponding sk-tree has a minimal number of nodes, as we have seen in the examples above, we shall circumvent the problem of finding the best possible optimal tree to be pruned by generating directly the requested sk-tree. This can be done by working with the q-source $N = \langle n_1, n_2, \ldots, n_k \rangle$ of an optimal code for the given weight distribution, rather than with one of its many possible corresponding optimal trees. If a small number of different q-sources are possible for a given frequency distribution and an optimal sk-tree for the distribution is sought, all corresponding q-sources can be processed. If, on the other hand, the number of possible q-sources is large, a polynomial algorithm to find the optimal sk-tree for the given distribution has recently been published in [16].

Starting with N, we shall produce the q-source $M = \langle m_1, m_2, \ldots, m_{k'} \rangle$ of the optimal sk-tree, where $k' \leq k$. Any complete binary tree having M as

q-source will be an optimal sk-tree, and we could, for example, choose the canonical tree as representative. We shall also indicate how to get from the optimal sk-tree to the optimal tree for the original n weights.

LEMMA 2: *Full subtrees having their leaves on different levels can be pruned independently.*

PROOF: Full subtrees involve only leaves appearing on the same level. In other words, if a subtree has leaves on different levels, it cannot possibly be a full subtree and is therefore not a candidate for being pruned. Any pruning may thus be applied to the leaves of a given level, without taking leaves on other levels into consideration. ∎

Consider level i and suppose there are n_i leaves on this level. The largest possible savings for this level can obviously be attained when n_i is a power of 2, say, $n_i = 2^h$, in which case, an entire full subtree of height h, having its leaves on level i, may be pruned. That is, it seems at first sight that an additional constraint has to be fulfilled, namely that the 2^h leaves all belong to the same subtree of height h, or in other words, they should all be adjacent. Referring to Figure 1(a), there are $n_3 = 4 = 2^2$ leaves on level 3, but they do not belong to a single subtree of height 2. Nevertheless, we show that the additional constraint is not needed.

LEMMA 3: *Given the number n_i of leaves on level i, let 2^h be the largest power of 2 not larger than n_i, that is, $h = \lfloor \log_2 n_i \rfloor$. Then an entire subtree of height h may be pruned.*

PROOF: Consider the Kraft sum $\sum_{j=1}^{k} n_j 2^{-j}$. According to Lemma 1, removing the 2^h leaves on level i and adding a leaf on level $i - h$ yields a new q-source that also satisfies the Kraft equality. The new q-source thus corresponds to a complete binary tree R. One can therefore choose any leaf on level $i - h$ of R and turn it to the root of a subtree of height h. The resulting tree has n_i leaves on level i belonging all to the same subtree. ∎

It follows from Lemma 3 that even though the 2^h leaves do not always belong to the same subtree, as in Figure 1(a), it is still true that *there exists* an optimal tree for which these nodes are clumped together, as in the example of Figure 1(b). As another example for which n_i is not a power of 2, the tree in Figure 4(a) has $n_3 = 5$, but does not allow the pruning of a subtree of height 2; but there exists an equivalent tree with leaves on the same levels, e.g., the tree in Figure 4(b), for which four of the leaves on level

3 are consecutive and part of the same subtree rooted at level 1.

The effect of the pruning on the q-source is materialized by updating the value of n_i to $n_i - 2^h$ and incrementing n_{i-h} by 1, reflecting the fact that 2^h leaves have been removed from the tree and a new leaf has been added. According to Lemma 1, the current q-source is again one of a complete tree, so the same argument as above can be repeated for the new value of n_i. Ultimately, what one gets is a decomposition of n_i into a sum of powers of 2, that is, the standard binary representation of n_i. For example, if $n_i = 47$, one could prune consecutively subtrees with 32, 8, 4 and 2 leaves on level i, after which, a single leaf will remain on this level.

While the different levels can be treated independently, the order by which to process them should not be arbitrary. Care has to be taken that only original leaves are considered when looking for a subtree to prune, and not newly added leaves resulting from a previous pruning action. This suggests to consider the levels top down, from level 1 to level k. Since when treating level i, nodes are only added at levels $i-h$, for $h \geq 1$, the additional nodes are inserted at levels that have been treated in previous iterations and will thus not be processed any more.

Summarizing, we propose a greedy algorithm that considers, in order for every $1 \leq i \leq k$, the n_i leaves corresponding to each codeword length i individually, and repeatedly prunes full trees having their number of leaves equal to 2^h, for the largest possible $h \geq 1$. The construction in [5] is similar, but presented as a heuristic improving the use of wavelet trees.

Algorithm 1 gets as input parameter the q-source $\langle n_1, n_2, \ldots, n_k \rangle$ of an optimal code for a given weight distribution and constructs a corresponding optimal sk-tree after having generated its q-source $\langle m_1, m_2, \ldots, m_{k'} \rangle$. We have chosen here the canonical form for this optimal sk-tree, but any other form could be used. The algorithm maintains a list $\mathcal{L}$ in which the pairs (i, h) are inserted, each identifying a pruned subtree T, with i being the index of the level of the leaves of T, and h being its height, implying that the number of its leaves is 2^h. Once the optimal sk-tree is constructed, the elements in $\mathcal{L}$ are used to assign the correct values $h(v)$ to its leaves v. The list $\mathcal{L}$ can be implemented as queue or stack or any other way, as long as it permits to process all of its elements in some order.

The algorithm can also be adapted to produce an optimal tree for the given weight distribution, whose corresponding sk-tree is optimal. All one needs to do is to replace the last line by

replace the leaf v by the root of a full subtree of height h

Algorithm 1: *Optimal Pruning Algorithm*

```
OptimalPruning(⟨n_1, n_2, ..., n_k⟩)
  for i ← 1 to k do
      m_i ← n_i
      while m_i ≥ 2 do
          h ← ⌊log_2 m_i⌋
          m_i ← m_i − 2^h
          m_{i−h} ← m_{i−h} + 1
          add the pair (i, h) to the list L
  k' ← max{i | 1 ≤ i ≤ k, m_i > 0}
  build canonical tree for ⟨m_1, m_2, ..., m_{k'}⟩ and set h(v) ← 0 to all its
    leaves v
  for each pair (i, h) ∈ L do
      choose a leaf v on level i − h for which h(v) = 0
      h(v) ← h
```

Theorem 1: *The sk-tree constructed by Algorithm 1 is optimal for the given q-source.*

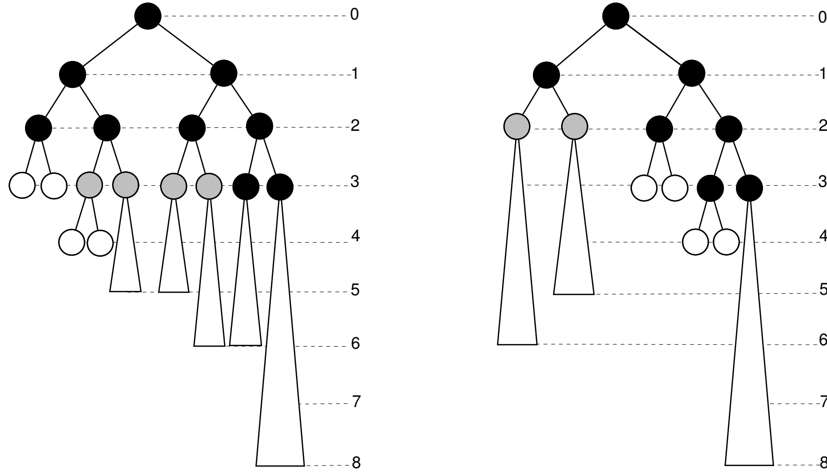
Proof: The claim follows from the above discussion. Lemma 1 implies that the structure of a complete tree may be maintained after each pruning action. Lemma 2 justifies that each level is treated separately and Lemma 3 suggests the greedy approach. Since at each step, the number of eliminated nodes is the largest possible, the size of the remaining tree at the end of the process is minimal. ∎

Applying Algorithm 1 on the q-source $\langle 0, 1, 5, 2\rangle$ results in the q-source $\langle 1, 1, 2\rangle$; a possible optimal tree yielding this optimal sk-tree is the one in Figure 4. Applying Algorithm 1 on the h-tuple q-source $\langle 0, \ldots, 0, 1, n-3, 2\rangle$ corresponding to Figure 5 results in the $(h-1)$-tuple q-source $\langle 1, \ldots, 1, 2\rangle$; a possible optimal tree yielding this optimal sk-tree is the one in Figure 6.

4. Reduced Skeleton Trees

The pruning of Huffman trees can be extended even further to a reduced tree that prunes the skeleton Huffman tree at some internal node at which the length of the current codeword may only be partially determined. Specifically, when getting to a leaf of a reduced skeleton tree, it is not necessarily

possible to deduce the exact length of the current codeword, but some partial information is already available: the length of the codeword belongs to a set of size at most 2. We shall refer to a subtree of the Huffman tree that is rooted by a leaf of the reduced tree as being *almost full.* More formally, each node v of the skeleton tree stores two values $lower(v)$ and $upper(v)$, which are computed recursively as follows. If v is a leaf, $lower(v)$ and $upper(v)$ are given the depth of v. Otherwise, $lower(v)$ is given the minimum of *lower* values of its children, and $upper(v)$ is given the maximum of *upper* values of its children. The reduced tree is defined as the smallest subtree of the skeleton tree for which all the leaves correspond to a range of at most two consecutive codeword lengths, i.e., $upper(w) \leq lower(w) + 1$. Figure 7(a) presents a canonical Huffman tree where nodes belonging to the corresponding reduced tree are highlighted in black, while those of the skeleton tree include also the gray nodes in addition to the black ones.



(a) *Reduced tree derived from a canonical Huffman tree.*

(b) *Reduced tree with fewer nodes, derived from a non-canonical tree.*

Figure 7: *Reduced canonical tree for the q-source* $\langle 0, 0, 2, 2, 8, 16, 0, 32\rangle$ ($m = 2$).

The decoding for reduced trees is similar to that of skeleton Huffman trees. When a leaf w is reached during the traversal of the reduced tree, the current codeword is initialized as having length $lower(w)$. As the codewords assigned to the leaves on a given level of the reduced tree are consecutive binary numbers, verifying whether the codeword for w is of length $lower(w)$

or $lower(w) + 1$ just needs one more comparison[2].

4.1. Greedy attempts to find optimal reduced skeleton trees

Extending the work of Section 3, we consider the problem of finding the reduced tree with a minimum number of nodes. An exhaustive search, generating all possible Huffman trees and then constructing the corresponding reduced skeleton trees to choose the one with a minimal number of nodes, is obviously impractical. A first attempt that comes to mind is to start the pruning process from a canonical tree, as suggested in [12] for sk-trees, but this is not always optimal: Figure 7(b) presents a Huffman tree with the same codeword lengths as in Figure 7(a) in which the resulting reduced tree has fewer nodes, 7 instead of 9. Note that there is also a difference in the number of nodes of the corresponding skeleton trees in this case, which is 13 and 9, respectively, depicted by the black and gray nodes.

As a second attempt, we try pruning the optimal skeleton tree derived from Algorithm 1. However, referring to Figure 4, a reduced tree obtained by pruning the optimal skeleton tree in Figure 4(b) yields five nodes, while the non-optimal skeleton tree given in Figure 4(a) yields only three nodes (as usual, colored in black).

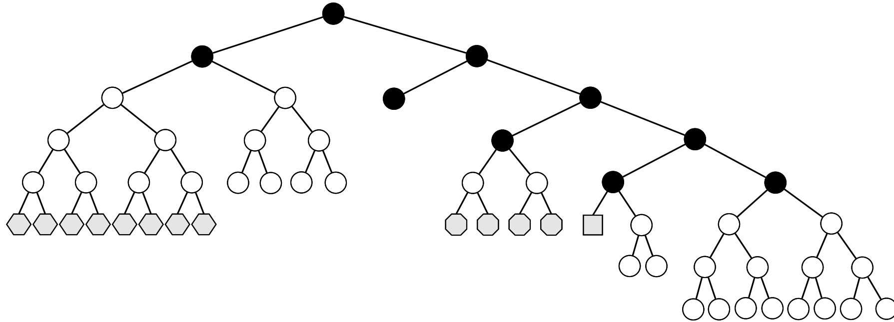


Figure 8: *Example of the partition of leaves on a given level into three different sets.*

The construction of optimal reduced trees faces a more involved challenge than that of skeleton trees, for which one could operate on a single level at a time, according to Lemma 2. In reduced trees, one has to consider two adjacent levels, and choose, among their leaves, those that will be joined

[2]Actually, in the special case when $lower(w) = upper(w)$, even this comparison may be saved, see [12].

into the same almost full subtrees. For example, Figure 8 gives a specific tree in which the set of leaves on level 5, colored in light gray, is partitioned into three subsets of leaves belonging to different almost full subtrees: the square leaf is adjoined to level 6, the almost full subtree including the eight hexagon leaves has its leaves on levels 4 and 5, and the four octagon leaves belong to a full, rather than only almost full, subtree rooted by a leaf of the reduced skeleton tree.

Consider the possibility of extending the optimal algorithm developed in the previous section for sk-trees. We would then process pairs of adjacent levels whose numbers of leaves are n_i and n_{i+1}, and try to generate repeatedly the largest almost full subtrees having their leaves on levels i and $i+1$. Formally, we are looking for

$$\max\{x + y \ \mid\ x \le n_i,\ \ y \le n_{i+1},\ \ \exists h\ 2x + y = 2^h\}, \tag{2}$$

to prune a subtree with x leaves on level i and y leaves on level $i+1$. This raises then the question of the order in which the levels are to be processed: contrarily to Algorithm 1, in which each level is treated independently, the pair of levels $(i, i+1)$ affects both pairs $(i-1, i)$ and $(i+1, i+2)$.

Greedy algorithms, that try to identify the largest almost full subtrees for every pair of adjacent levels in our quest for optimal reduced skeleton trees, do not necessarily yield optimal solutions. In particular:

1. joining pairs of adjacent levels in a top-down scan of the tree;
2. joining pairs of adjacent levels in a bottom-up scan of the tree; and
3. joining, at each step, the levels yielding the largest possible subtree to be pruned.

Counterexamples showing the non-optimality of these approaches are the q-sources $\langle 0,0,6,2,4\rangle$, $\langle 0,1,1,5,8,2,0,8\rangle$ and $\langle 0,0,2,8,2,7,9,2\rangle$, respectively, and the corresponding tree pairs can be seen in Appendix A.

4.2. Dynamic Programming Solution

This section presents a dynamic programming (DP) solution for finding the optimal reduced skeleton tree, based on finding an optimal partition of the leaves of the tree according to equation (2). We first extend the notion of a q-source as follows: a string $\langle m_1, m_2, \ldots, m_i\rangle$ is called a *sub-q-source*, or sq-source for short, if there exists a q-source $\langle n_1, n_2, \ldots, n_k\rangle$ such that

$$i \le k, \qquad \forall j < i \quad m_j = n_j \qquad \text{and} \qquad m_i \le n_i, \tag{3}$$

that is, the elements of an sq-source are a prefix of some q-source, the first $i-1$ numbers being identical, and the last one being smaller or equal. For example $\langle 0,0,6,1\rangle$ is an sq-source, because $\langle 0,0,6,2,4\rangle$ is a q-source. Note that while any q-source satisfies the Kraft equality of eq. (1), an sq-source satisfies $\sum_{j=1}^{i} m_j 2^{-j} \leq 1$, with equality only if the sq-source is a q-source, that is, $i = k$ and $m_k = n_k$.

This extension to sq-sources is needed because the DP process incrementally builds an optimal solution basing itself on previously calculated optimal solutions of sub-problems. However, in our case, the sub-problems correspond to prefixes of the given q-source in the sense defined by equation (3), but these sq-sources do not describe full binary trees.

The idea is therefore to consider the problem formally as finding an optimal partition of the elements described by some sq-source, according to some criteria. It is only at the final stage of the DP, when the input sq-source is in fact the original q-source, that the optimal partition corresponds to an optimal reduced skeleton tree.

The criterion for the partition is the following. Consider the number m_j in the given sq-source $\langle m_1, m_2, \ldots, m_i\rangle$ as the number of leaves on level j in some part of a binary tree. In a process we shall call *partitioning the q-source*, we try to group adjacent leaves together when they can be the leaves of full or almost full binary trees, and seek such a grouping which yields the smallest possible number of trees. More precisely, given that there are $m_1 + m_2 + \cdots + m_i$ leaves in total, we try to partition them into the smallest possible number h of disjoint subsets $G_1 \cup G_2 \cup \ldots \cup G_h$, called *classes*, such that each G_j consists of leaves satisfying one of the following constraints:

1. either all the w leaves in G_j belong to the same level, and w is a power of 2;
2. or the leaves in G_j belong to two adjacent levels, say x leaves from the m_t on level t and y leaves from the m_{t+1} on level $t+1$, and $2x+y$ is a power of 2.

For example, Figure 9 gives the optimal partitions for the sq-sources $\langle 0,0,3,3\rangle$, $\langle 0,0,3,4\rangle$ and $\langle 0,0,3,5\rangle$. The black squared nodes represent the roots of full or almost full binary trees of the possible optimal partitions. For the rightmost tree, G_1 includes all the $m_3 = 3 = x$ leaves, and $y = 2$ of the $m_4 = 5$ leaves, and indeed $2x + y = 8 = 2^3$; G_2 includes $w = 2$ of the remaining $m_4 - 2$ leaves and G_3 is the remaining singleton. There is another optimal partition for $\langle 0,0,3,5\rangle$, in which the number of leaves in

G_1, G_2 and G_3 are 1 (from level 3), 6 (2 from level 3 and 4 from level 4) and 1 (from level 4), respectively.

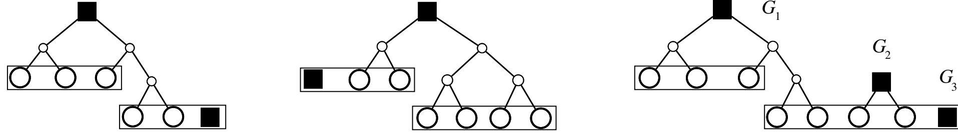


Figure 9: *The optimal partitions for the sq-sources* $\langle 0,0,3,3\rangle$, $\langle 0,0,3,4\rangle$ *and* $\langle 0,0,3,5\rangle$.

A DP solution uses a table of size $k \times \max\{n_i\}_{i=1}^{k}$ named PT. The value $PT[i,y]$ denotes the minimum number of classes G_j into which the sq-source $\langle n_1, n_2, \ldots, n_{i-1}, y\rangle$ can be partitioned. For example, if the input is the q-source $\langle 0,0,3,7,5,2\rangle$, the calculated values of $PT[4,3], PT[4,4]$ and $PT[4,5]$ are 2, 2 and 3, respectively.

The value of $PT[i,y]$ is obtained by inspecting the currently last levels $i-1$ and i and splitting the n_{i-1} leaves of level $i-1$ into two parts of x and $n_{i-1}-x$ leaves, for all possible values of x. For the sq-source $\langle n_1, n_2, \ldots, n_{i-2}, x\rangle$, the optimal solution can then be found in $PT[i-1,x]$. What is left is to partition the $n_{i-1}-x$ remaining leaves of level $i-1$ and the y leaves of level i into the minimum number of classes.

In Algorithm 1, for a given number z of leaves, the minimal number M of full subtrees in the decomposition was found by repeatedly decreasing z by the largest possible power of two; equivalently, M is $\mathsf{N1B}(z)$, the number of 1-bits in the standard binary representation of z, which is often called *popcount*(z). Similarly, in our case dealing with almost full subtrees with z leaves on level $i-1$ and y leaves on level i, the minimal number of classes will be $\mathsf{N1B}(2z+y)$, according to eq. (2).

We can thus summarize the DP construction in the formal algorithm below. Recall that a full binary tree with x leaves has $2x-1$ nodes, thus the number of nodes in an optimal reduced tree is calculated in line 6 using the answer retrieved from cell $PT[k,n_k]$ for the entire q-source. From the above discussion, we conclude the following.

THEOREM 2: *The size of the reduced sk-tree computed by Algorithm 2 is optimal for the given q-source.*

An example of the execution of the DP algorithm can be found in Figure 10. The time complexity of the algorithm is clearly bounded by $O(kn^2)$, where $n = \max\{n_1, \ldots, n_k\}$. The necessary space for storing the full DP

Algorithm 2: *Optimal Reduced Skeleton Trees*

SizeOfOptimalReduced($\langle n_1, n_2, \ldots, n_k\rangle$)

1 $PT[0,0] \leftarrow 0$
2 $PT[i,y] \leftarrow \infty \qquad 0 \le i \le k, \quad 0 \le y \le n_i$
3 **for** $i \leftarrow 1$ **to** k **do**
4 | **for** $y \leftarrow 0$ **to** n_i **do**
5 | | $PT[i,y] \leftarrow \min_{0\le x\le n_{i-1}} \big(PT[i-1,x] + \mathsf{N1B}(2n_{i-1} - 2x + y)\big)$
6 **return** $2 \cdot PT[k, n_k] - 1$

table is $O(kn)$. However, if we are only interested in the size of the reduced tree, it suffices to store only two last rows of the table (i.e., $PT[i-1,y]$ and $PT[i,y]$ for all y), which decreases the space complexity to $O(n)$.

Reconstructing an optimal reduced tree together with the underlying Huffman tree requires some more effort. First of all, a standard technique of storing parents of states of the DP should be used. More precisely, in addition to $PT[i,y]$, we store $parent[i,y]$ that is equal to an index x that yields the minimum in line 5 of the algorithm. Then the algorithm can go back from $PT[k,n_k]$ to $PT[0,0]$ using these values, as shown by arrows in Figure 10. On the way it restores, for every pair of levels $i-1$ and i, the numbers of leaves at these levels that are to be combined together into classes, that is, full or almost full subtrees; when going from $PT[i,y]$ to $PT[i-1,x]$ with $x = parent[i,y]$, we get y leaves at level i and $z = n_{i-1} - x$ leaves at level $i-1$.

Assume that z leaves at level $i-1$ and y leaves at level i are to be partitioned into subtrees. Let $u = 2z + y$. From the above discussion we know that these leaves can be partitioned into $\mathsf{N1B}(u)$ classes G_j which correspond to full or almost full binary trees. For every $p = \lfloor \log_2 u \rfloor, \ldots, 0$, if the standard binary representation of the integer u has the bit corresponding to 2^p set, then the reduced tree has a leaf at level $i-p$. In the underlying Huffman tree there is a full or an almost full subtree with root at level $i-p$. We can create the subtree by using $w_1 = \min(2^{p-1}, z)$ leaves at level $i-1$ and $w_2 = 2^p - 2w_1$ leaves at level i and then decrease z by w_1 and y by w_2. For example, if there are $z = 3$ leaves at level 3 and $y = 5$ leaves at level 4 to be partitioned into subtrees, we have $u = 2 \times 3 + 5 = 11 = (1011)_2$, so the values for p are 3, 1 and 0. This corresponds to subtrees with roots at levels 1, 3 and 4, as shown on the rightmost example of Figure 9. For the subtree that corresponds to $p = 3$, we take all the $w_1 = 3$ leaves from level

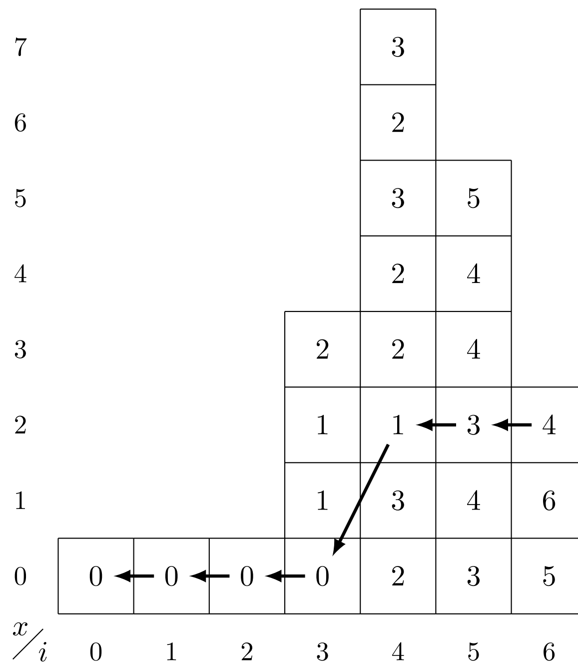


Figure 10: *An example of the execution of the DP algorithm for the q-source* $\langle 0, 0, 3, 7, 5, 2 \rangle$*. The values in the cells represent* $PT[i, x]$.

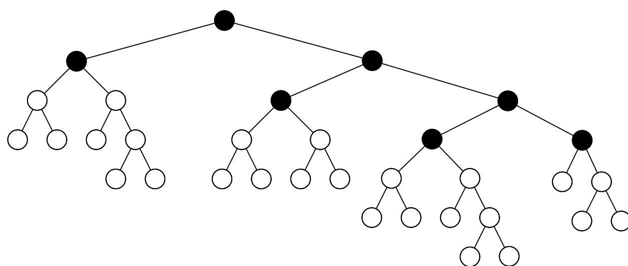


Figure 11: *An optimal reduced tree for the q-source* $\langle 0, 0, 3, 7, 5, 2 \rangle$*; see also Figure 10.*

3 and $w_2 = 2$ leaves from level 4.

Finally, all the leaves of the reduced tree need to be joined together to form the tree. To this end we can construct a canonical full binary tree with

this set of leaves. Note that the leaves of subtrees can be sorted by depths in $O(k+n)$ time using counting sort.

As an example, an optimal reduced tree and the corresponding Huffman tree for the q-source $\langle 0,0,3,7,5,2\rangle$ are shown in Figure 11. They are constructed according to the path of parents (see Figure 10):

- $parent[6,2]=2$, $z=3$, $y=2$ and $u=8$ imply an almost full subtree in the Huffman tree that contains 3 leaves at depth 5 and 2 leaves at depth 6; the root of this subtree has depth 3;
- $parent[5,2]=2$, $z=5$, $y=2$ and $u=12$ imply a full subtree that contains 4 leaves at depth 4 and an almost full subtree that contains 1 leaf at depth 4 and 2 leaves at depth 5; the roots of these subtrees have depths 2 and 3, respectively;
- $parent[4,2]=0$, $z=3$, $y=2$ and $u=8$ imply an almost full subtree that contains 3 leaves at depth 3 and 2 leaves at depth 4; the root of this subtree has depth 1.

When the reduced tree is constructed, the subtrees are ordered by the depths of their roots so that the reduced tree has a canonical form. Let us note that in this case the underlying Huffman tree is not necessarily in a canonical form. Let us also note that the reduced tree is generated according to the *set* rather than the *sequence* of depths. For example, if one would try to adhere to the order of depths as they were recovered above (i.e., with depths of roots 3, 2, 3, 1), no corresponding tree could be constructed.

4.3. Reduced trees derived from canonical Huffman trees

We saw already in Section 4.1 that starting the pruning process from a canonical Huffman tree does not necessarily lead to a smallest possible reduced tree. Next we show that not only does this approach not yield an optimum, but the difference in the number of nodes between an optimal reduced tree and a reduced tree that has been derived from a canonical Huffman tree may even not be bounded by a constant.

We start with a special case seen at the beginning of this section: Figure 7(a) presents the eight nodes reduced tree derived from a canonical Huffman tree for the q-source $\langle 0,0,2,2,8,16,0,32\rangle$, whereas Figure 7(b) is an improved reduced tree for the same q-source with only six nodes. The q-source for the generalisation of this example to a tree of height $3m+2$, with $m\geq 2$, is given in Figure 12. The upper line lists the indices of the

1	2	$\cdots$	m	m+1	m+2	m+3	m+4	m+5	m+6	m+7	$\cdots$	3m–2	3m–1	3m	3m+1	3m+2
0	0	$\cdots$	0	2	2	2^3	2^4	2^6	2^7	2^9	$\cdots$	2^{3m-5}	2^{3m-3}	2^{3m-2}	0	2^{2m+1}

Figure 12: *q-source for the general case of the trees in Figure 13 and 14.*

levels i above the corresponding n_i values for the number of leaves. The corresponding generalized canonical Huffman tree is presented in Figure 13.

The canonical tree is constructed so that the lowest level, indexed $3m+2$, is two levels deeper than the adjacent level with leaves, indexed $3m$, so that these levels cannot be combined in the reduced tree; their common ancestor x, on level $m-1$, as well as all ancestors of x itself, should thus also belong to the reduced tree. The remaining levels of the tree can only be partially combined: The leaves on levels $i \in \{m+3, m+4, \ldots, 3m\}$ are arranged so that the $n_i = 2^\ell$ leaves on level i are partitioned between an increasing number of subtrees. Specifically, the 8 nodes on level $m+3$ are divided into two subtrees with number of leaves 4 and 4, and the 16 nodes on level $m+4$ are divided into two subtrees with number of leaves 8 and 8. The leaves on the following two levels (64 leaves on level $m+5$ and 128 leaves on level $m+6$) are each partitioned into three subtrees, with number of leaves 16, 32, 16 and 32, 64, 32, respectively. The leaves on the following two levels (512 leaves on level $m+7$ and 1024 leaves on level $m+8$) are partitioned into four subtrees with number of leaves 64, 256, 128, 64 and 128, 512, 256, 128, and so on. Figure 7 is the special case for $m=2$.

Counting the number of nodes in the reduced tree, there are: a single node, the root, on level 0; two nodes on level 1; four nodes on level 2; and six nodes on each of the levels $3, 4, \ldots, m-1$. There are two additional nodes on level m, so the total number of nodes in the reduced tree is $6m-3$, and they appear in black in Figure 13.

A better reduced tree for the same q-source is presented in Figure 14. There are two nodes on each of the levels 1 to $m+1$, for a total of $2(m+1)+1 = 2m+3$ nodes, including the root. The difference in the number of (black) nodes in the reduced trees of Figure 13 and Figure 14 is, thus, $4m-6$, which is a function of their height, and bounded below by $\Omega(\log n)$.

Note that the number of (gray or black) nodes in the skeleton tree of Figure 13 is $2m(m+1)+1$, for $m>1$, which also shows that the number of nodes in the reduced tree can improve on that of the corresponding skeleton tree by a factor of m.

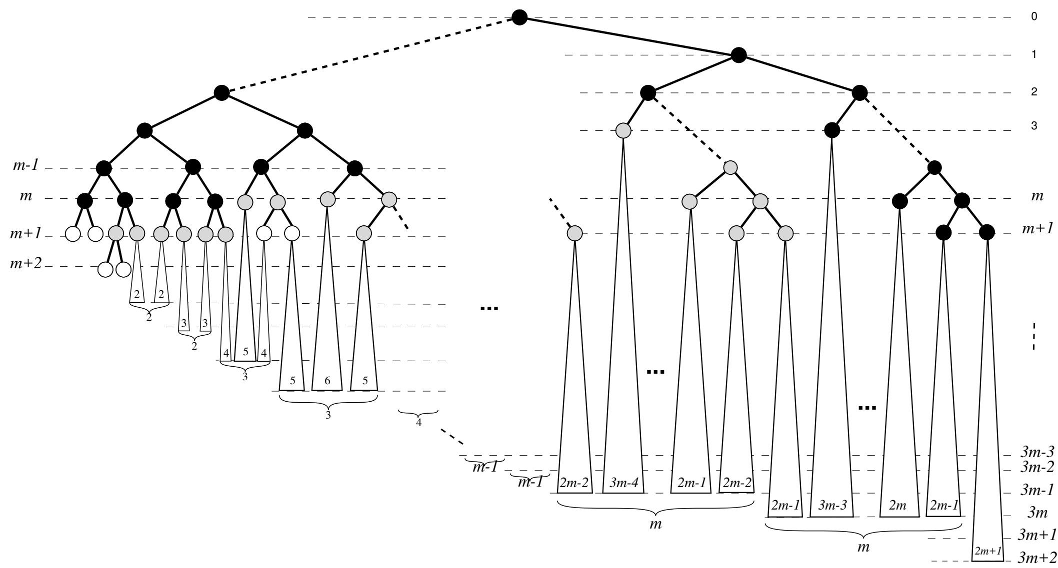


Figure 13: *General Canonical Tree*

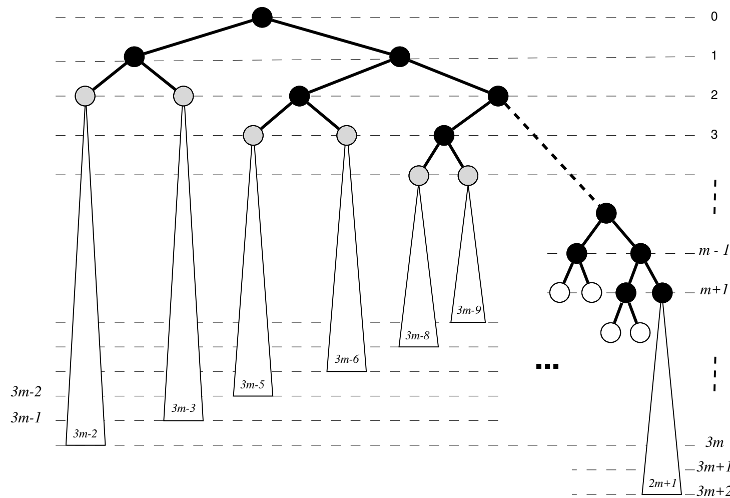


Figure 14: *Non-constant improvement of reduced skeleton Huffman trees.*

5. Experimental Results

We considered four texts of different languages and alphabet sizes. *ebib* is the Bible (King James version) in English, in which the text was stripped

of all punctuation signs; *ftxt* is the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [24]; *sources* is formed by C/Java source codes obtained by concatenating all the .c, .h and .java files of the linux-2.6.11.6 distributions; and *English* is the concatenation of English text files selected from etext02 to etext05 collections of the Gutenberg Project, from which the headers related to the project were deleted so as to leave just the real text. In addition, we considered also Zipf's law [25] on $|\Sigma| = 200$ elements, which is a theoretical distribution rather than a real-life dataset. The law is believed to govern the distribution of the words in a large natural language text, and is defined by the weights $p_i = 1/(i\, H_{|\Sigma|})$, for $1 \leq i \leq |\Sigma|$, where $H_n = \sum_{j=1}^{n}(1/j)$ is the n-th harmonic number.

Table 1 presents some information on the data files involved. The second and third columns present the original file sizes in MB and millions of words, and the fourth column gives the size of the encoded alphabet, $|\Sigma|$.

File	size (MB)	# of words (in millions)	$\lvert\Sigma\rvert$
ebib	3.5	0.6	53
ftxt	7.6	1.2	127
sources	200.0	25.8	208
English	200.0	37.0	217
Zipf	–	–	200

TABLE 1: *Information about the used datasets.*

The experimental results are summarized in Table 2. Columns 2, 3 and 4 list, respectively, the number of nodes in a Huffman tree (Huff), in an sk-tree based on pruning a canonical Huffman tree (can), as advocated in [12], and in an optimal sk-tree (opt), according to Algorithm 1. The column headed rdcd is the number of nodes in an optimal reduced sk-tree obtained by further pruning the canonical sk-tree using the proposed DP solution presented in Algorithm 2. There is a gain of 12–24% for the given example files.

The last four columns of the table give the average number of necessary bit comparisons for the decoding of a single codeword. Averages are evaluated by using a model that assigns a probability of 2^{-m} to a leaf of depth m. Note that such a dyadic distribution is a good approximation of the actual probabilities, as it yields the same Huffman tree. For a Huffman tree, the number of comparisons is the codeword length, and for the skeleton trees,

these numbers are smaller since no additional bit comparisons are needed once the codeword length is known, that is, a leaf of the sk-tree has been reached. Since decoding time should be roughly proportional to the number of processed bits, these averages can be seen as estimates for the average decoding times.

The improvement in the average number of bit comparisons of the optimal over the canonical sk-tree is, for our examples, of about 4–11%. We see that, in spite of the already significant savings in both time and space of the canonical sk-tree versus Huffman trees, passing to the non-intuitive optimal sk-trees may still yield some additional gain. In all cases, using reduced trees rather than only skeleton trees decreases the number of nodes by additional 38% to 65%, implying an improvement of 13-33% on the average depth of the leaves.

File	**number of nodes**				**avrg # bit comparisons**			
	Huff	can	opt	rdcd	Huff	can	opt	rdcd
ebib	105	57	47	29	4.22	3.35	2.97	2.00
ftxt	253	89	77	45	4.59	3.14	3.02	2.19
English	433	129	113	57	4.48	3.22	3.00	2.00
sources	415	93	71	45	5.55	3.42	3.17	2.25
Zipf	399	49	37	13	6.15	4.09	3.61	2.75

TABLE 2: *Comparing tree sizes and average number of bit comparisons.*

6. Conclusion

Skeleton and reduced skeleton trees have been introduced as data structures improving both the space and time complexities of the decoding of texts encoded by optimal prefix codes such as Huffman's. The originally suggested construction of skeleton and reduced trees is based on canonical Huffman trees, clustering leaves on each level together, according to the assumption that this should increase the number of nodes in the pruned subtrees. This paper shows, however, that this intuition is misleading, as pruning a canonical tree does not always yield a tree with a minimal number of nodes. Algorithms for creating such an optimal skeleton tree and an optimal reduced tree are presented. We also showed that several simple approaches to construct reduced trees do not always result in optimal ones. Our algorithms work for a given q-source. If the number of different q-sources for the same distribution is too large to allow processing each on its own, one can use the polynomial-time solution proposed in [16].

Note that sk-trees and reduced trees are just one of the possibilities to enhance decoding: while some prefix of each codeword is processed bit by bit, several bits forming its suffix may be dealt with as a single unit. Alternatively, other methods use lookup tables prepared in a preprocessing stage, to decode prefixes or other substrings of codewords, or even several codewords together, as a bulk [3].

Acknowledgements. Jakub Radoszewski was supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

References

[1] Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. In: Algorithms and Computation - 21st International Symposium (ISAAC 2010). pp. 315–326 (2010)

[2] Baruch, G., Klein, S.T., Shapira, D.: A space efficient direct access data structure. Journal of Discrete Algorithms 43, 26–37 (2017)

[3] Bergman, E., Klein, S.T.: Fast decoding of prefix encoded texts. In: 2005 Data Compression Conference (DCC 2005), 29-31 March 2005, Snowbird, UT, USA. pp. 143–152 (2005)

[4] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009)

[5] Dubé, D.: Leaner skeleton trees for direct-access compressed files. Proc. IEEE International Symposium on Information Theory and Its Applications (ISITA), Monterey pp. 122–130 (2016)

[6] Ferguson, T.J., Rabinowitz, J.H.: Self-synchronizing Huffman codes. IEEE Trans. Information Theory 30(4), 687–693 (1984)

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

[8] Golomb, S.W.: Sources which maximize the choice of a Huffman coding tree. Information and Control 45(3), 263–272 (1980)

[9] Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA) pp. 841–850 (2003)

[10] Huffman, D.: A method for the construction of minimum redundancy codes. Proc. of the IRE pp. 1098–1101 (1952)

[11] Jacobson, G.: Space efficient static trees and graphs. Proc. Foundations of Computer Science (FOCS) pp. 549–554 (1989)

[12] Klein, S.T.: Skeleton trees for the efficient decoding of Huffman encoded texts. Kluwer Journal of Information Retrieval 3, 7–23 (2000)

[13] Klein, S.T.: Basic Concepts in Data Structures. Cambridge University Press (2016)

[14] Klein, S.T., Serebro, T.C., Shapira, D.: Optimal skeleton Huffman trees. In: String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings. pp. 241–253 (2017)

[15] Klein, S.T., Shapira, D.: Random access to Fibonacci encoded files. Discrete Applied Mathematics 212, 115–128 (2016)

[16] Kosolobov, D., Merkurev, O.: Optimal skeleton Huffman trees revisited. CoRR abs/2001.05239 (2020), https://arxiv.org/abs/2001.05239

[17] Navarro, G., Providel, E.: Fast, small, simple rank/select on bitmaps. In: Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings. pp. 295–306 (2012)

[18] Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. 2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM pp. 60–70 (2007)

[19] Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms 3(4), 43 (2007)

[20] Salomon, D.: Data Compression: The Complete Reference. Springer (1998)

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

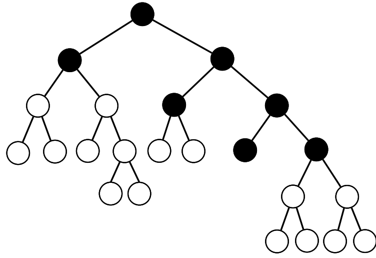
[22] Shapira, D., Daptardar, A.: Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts. Information Processing and Management, IP & M 42(2), 429–439 (2006)

[23] Turpin, A., Moffat, A.: Fast file search using text compression. In: Proc. 20th Australasian Computer Science Conference, Sydney, Australia. pp. 1–8 (1997)

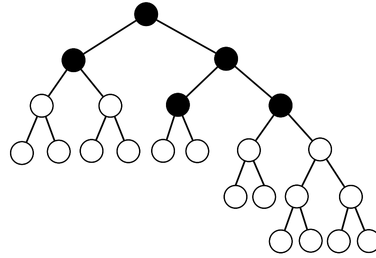
[24] Véronis, J., Langlais, P.: Evaluation of parallel text alignment systems: the ARCADE project. Parallel Text Processing pp. 369–388 (2000)

[25] Zipf, G.K.: The Psycho-Biology of Language. Houghton-Mifflin, Boston (1935)

Appendix A

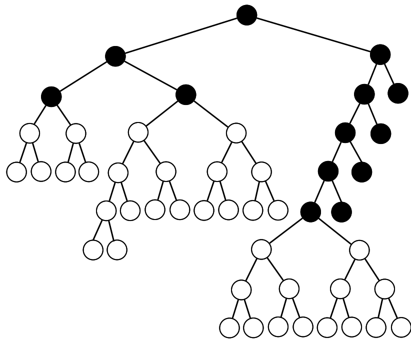


(a) *Non-optimal reduced tree.*

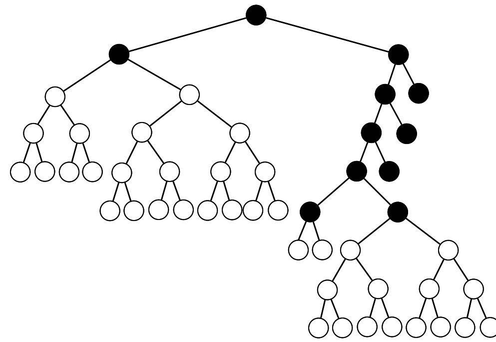


(b) *Better reduced tree.*

Figure A.1: *Top-down counterexample with q-source* $\langle 0, 0, 6, 2, 4 \rangle$.

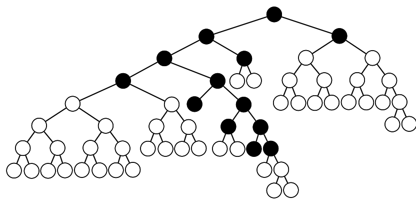


(a) *Non-optimal reduced tree.*

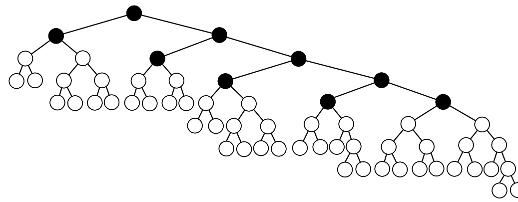


(b) *Better reduced tree.*

Figure A.2: *Bottom-up counterexample for the q-source* $\langle 0, 1, 1, 5, 8, 2, 0, 8 \rangle$.



(a) *Non-optimal reduced tree.*



(b) *Better reduced tree.*

Figure A.3: *Maximum possible counterexample using the q-source* $\langle 0, 0, 2, 8, 2, 7, 9, 2 \rangle$.