Leaner Skeleton Trees  
for Direct-Access Compressed Files

Danny Dubé  
Département d’informatique et de génie logiciel, Université Laval  
Québec City, QC, Canada  
Email: Danny.Dube@ift.ulaval.ca

Abstract—A wavelet tree, as proposed by Grossi et al., allows the storage of a file in a compressed form while providing efficient direct access to its contents and other manipulation operations. A variant of such trees consists in using a Huffman code to represent the file contents. The shape of such a wavelet tree mimics the shape of the Huffman tree. The bits stored at each internal node need not be compressed since they come from a Huffman code. Additional data structures at the internal nodes must be provided to implement the ranking operation that is necessary to have a fast access to the symbols of the file. Later, Klein and Shapira proposed the concept of skeleton tree in order to relax the requirement of the ranking-related data structures for some internal nodes. Recently, Baruch et al. have proposed an improvement to the latter technique in order to increase the number of internal nodes that are spared from the requirement of the additional data structures. Here we propose to further improve the technique proposed by Baruch et al. by giving the skeleton tree the ability to fully exploit the opportunities to spare the internal nodes from the additional data structures.

I. WAVELET TREES

A. WAVELET TREES WITH COMPRESSED LABELS

Representing a large file or a large string under a compressed form is convenient to save space. For instance, a string drawn from the alphabet {A, B, C, D, E, F, G, H, I} may be compressed using a Huffman code such as the one depicted by the tree in Figure 3(a). However, since such a prefix code is a fixed-to-variable one, it is not possible to have an efficient direct access to an arbitrary symbol in the string by its position.

In order to obtain efficient direct access to the symbols of a string while still benefiting from compression, Grossi et al. proposed the wavelet trees (WTs) [1]. Instead of relying on a prefix-free code to compress the string, they use a “flat” encoding of the symbols in codewords of \( \log |\Sigma| \) bits (more exactly, either \( \lceil \log |\Sigma| \rceil \) or \( \lfloor \log |\Sigma| \rfloor \) bits), where \( \Sigma \) is the alphabet, but they shuffle the bits of the coded symbols into the shape of a WT and use compression on the bit strings that label the WT nodes. Let \( s \in \Sigma^* \) be a string we want to represent under the form of a WT. The WT separates the bits of the codewords according to their bit planes. Let \( T_s \) be this WT, where \( \epsilon \) denotes the empty string. The root of \( T_s \) is labelled by a bit string that is formed by taking the first bit of the codeword of each symbol \( s_i \) of the string \( s \). The left-hand side child \( T_0 \) of the root is the subtree that describes the remaining bits of the successive symbols whose codewords begin with 0. The right-hand side child \( T_1 \) of the root is the subtree that describes the remaining bits of the successive symbols whose codewords begin with 1.

The construction of a WT \( T_s \) for a string \( s \in \Sigma^* \) proceeds as described in Figure 1, where the construction is guided by a prefix-free code, which is both specified by the function \( C : \Sigma \rightarrow \{0, 1\}^* \) and the code tree \( t_s \).

Note that, in the WT’s proposed by Grossi et al., the various labels \( L_{w*} \) have to be stored in a compressed form. (This is not necessarily the case in other kinds of WT’s, as we see below.) This is necessary since the codewords that are assigned to the symbols of \( \Sigma \) are not selected according to the symbols frequencies. Consequently, a label \( L_{w*} \) may contain vastly different numbers of 0s and 1s, when the number of symbols whose codewords begin in \( w0 \) is vastly different from the number of symbols whose codewords begin in \( w1 \).

Many authors have described techniques to compress bit strings while offering direct access to individual bits, as well as other operations [2], [3], [4], [5], [6].

In order to decode symbol \( s_i \) and as a consequence of the organization of the WT, one starts by decoding the first bit \( b_{i,1} \) of \( C(s_i) \) from the root’s label \( L_{r*} \) and, depending if \( b_{i,1} = 0 \) or 1, the rest of \( C(s_i) \) is then decoded from \( T_0 \) or \( T_1 \), respectively. Since \( L_{r*} \) contains some numbers of 0s and 1s, in some order, then the ranking operator has to be used. The operation \( \text{rank}(w, b, j) \) returns the number of occurrences of bit \( b \) before position \( j \) in string \( w \). When \( b_{i,1} = 0 \), we continue decoding by looking for the rest of the codeword at position \( \text{rank}(L_{r*}, 0, i) \) in \( T_0 \) and, when \( b_{i,1} = 1 \), we continue decoding by looking for the rest of the codeword

---

1 From now on, we only refer to a string instead of a file and/or a string, except in the section on experiments.

2 All logarithms are taken to base 2, in this paper.
by referring to the desired symbol by its position in labels, as proposed by Grossi et al. Decoding a symbol takes the computation of a rank has to be performed on compressed method is guided by the code tree source string. Like the construction method, the decoding process. One obtains at position \( \text{rank}(L_w, 1, i) \) in \( T_1 \). Figure 2 formally describes the decoding of a symbol from the WT can now be performed in average performance of the used Huffman code. Second, the decoding in the codewords contain no or little redundancy, up to the more need to compress the labels in the WT. Indeed, the bits First, since the code compresses the symbols, there is no code. In this case, the shape of the WT simply mimics the construction of a WT can instead be based on a Huffman B. Wavelet Trees Based on Huffman Trees and the constant-time complexity of the ranking operation. Fig. 1. Construction of a wavelet tree. 

\[
\begin{align*}
\text{build}_{\text{main}}(s) &= \text{build}(t_c, \ [C(s_i) \mid 0 \leq i < |s|]) \\
\text{build}(t_w, [v_0, \ldots, v_{k-1}]) &= \left\{ \begin{array}{ll}
\text{a leaf with label } L_w = e, & \text{if } t_w \text{ is a leaf}, \\
an \text{internal node with label } L_w \text{ and children } T_w0 \text{ and } T_w1, & \text{otherwise}, \\
\end{array} \right.
\end{align*}
\]

\[
\begin{align*}
\text{where: } b_i u_i &= v_i, & \text{for } 0 \leq i < k, \\
b_i &= \{0, 1\}, & \text{for } 0 \leq i < k, \\
L_w &= [b_0, \ldots, b_{k-1}], \\
T_w0 &= \text{build}(t_w0, [u_i \mid 0 \leq i < k \text{ and } b_i = 0]), \\
T_w1 &= \text{build}(t_w1, [u_i \mid 0 \leq i < k \text{ and } b_i = 1]).
\end{align*}
\]

Fig. 2. Decoding of a symbol from a wavelet tree.

\[
\begin{align*}
\text{decode}(t_w, i) &= \left\{ \begin{array}{ll}
\text{the symbol associated to } t_w, & \text{if } t_w \text{ is a leaf}, \\
\text{decode}(t_w0, \text{rank}(L_w, 0, i)), & \text{if } t_w \text{ is not a leaf and the bit in } L_w \text{ at position } i \text{ is } 0, \\
\text{decode}(t_w1, \text{rank}(L_w, 1, i)), & \text{otherwise}.
\end{array} \right.
\end{align*}
\]

at position \( \text{rank}(L_w, 1, i) \) in \( T_1 \). Figure 2 formally describes the decoding process. One obtains \( s_i \) by computing \( \text{decode}(t_c, i) \), by referring to the desired symbol by its position \( i \) in the source string. Like the construction method, the decoding method is guided by the code tree \( t_c \) that has been used on the source string. In Figure 2, we keep implicit the fact that the computation of a rank has to be performed on compressed labels, as proposed by Grossi et al. Decoding a symbol takes time in \( O(\log |\Sigma|) \), due to the top-down traversal of the WT and the constant-time complexity of the ranking operation. 

B. Wavelet Trees Based on Huffman Trees

It is not necessary to use a flat code to build a WT. The construction of a WT can instead be based on a Huffman code. In this case, the shape of the WT simply mimics the shape of the Huffman tree. This brings a few advantages. First, since the code compresses the symbols, there is no more need to compress the labels in the WT. Indeed, the bits in the codewords contain no or little redundancy, up to the performance of the used Huffman code. Second, the decoding of a symbol from the WT can now be performed in average time \( O(H_0) \), where \( H_0 \) is the order-0 entropy of the source symbols, which is sometimes sensibly better than \( O(\log |\Sigma|) \).\(^3\) Third, another advantage is the ability to extract skeleton trees out of the WTs, as we explain below. This idea of skeleton trees is not ours but that of Klein and Shapira [7]. Fourth, storing a WT based on a Huffman code on disk is extremely simple as one merely needs to concatenate the node labels found during a depth-first or breadth-first traversal of the WT. The label of any internal node specifies the number of symbols that are described by the two children of the node. So there is no need for special delimiters between the labels of different nodes. Loading such a WT in memory from disk only requires some simple computations to recover the start and end positions of all the nodes’ labels and then one has an operational direct-access data structure.

Building a Huffman-based WT proceeds almost totally in the same way as one based on a flat code. Given a Huffman code for \( \Sigma \), described by both the coding function \( C \) and the code tree \( t_c \), one simply has to use the \( \text{build}_{\text{main}} \) procedure. The construction is even simpler due to the fact that the various labels \( L_w \) do not need to be compressed. Decoding a symbol from a WT can still be performed using the \( \text{decode} \) function. The symbol at position \( i \) in the source string is recovered by computing \( \text{decode}(t_c, i) \). Note that, while reading a bit from the now uncompressed labels is trivial, we still need extra data structures for the ranking operation to work.

Let us present an example of a WT based on a Huffman code. Let the alphabet \( \Sigma \) be \{A, B, \ldots, H, I\}. Let the source string \( s \) be (the quite artificial):

\[
\text{ABCDEFGHIABCDEFGHABCDEFGABCDEFABCDEABCDABCABA,}
\]

which causes the symbols in \( \Sigma \) to have frequencies 9, 8, \ldots, 2, and 1, respectively. Figure 3(a) depicts a Huffman tree that may be built from this alphabet and these frequencies. The internal nodes have been numbered, for ease of reference. Although the arcs are not labelled, we take the convention that we are interested in recovering the symbol in the exact center of \( s \), i.e. \( s_{22} \). We know that this symbol is an F because we have built the example but the normal user does not know ahead of time which symbol she is going to recover. We obtain the first bit of \( C(s_{22}) \) by getting the bit at position 22 in \( L_c \), which is node 1’s label in our example. It is a 0 and this 0 has rank 8. So we must continue the decoding by looking for the

\(^3\)On the other hand, note that the worst-case decoding time increases proportionally to the depth of the Huffman tree.
...the corresponding parts in the WT [7]. These parts are the code tree may lead to a lighter and faster implementation of C. Skeleton Trees and Canonical Huffman Trees

In particular, all the leaves are trivially balanced. When one of these nodes is reached during decoding, we can determine which instances of a particular leaf we want to recover. For example, at node 3, once we have read the bit 0 at position 5 in L_01, we do not need to know that this 0 is the third of four, since we are to decode an F anyway. Note that, in Figure 3, the balanced nodes are depicted with parentheses, while the unbalanced nodes are depicted with square brackets. In particular, all the leaves are trivially balanced.

Klein and Shapira have also observed that constant-depth subtrees of depth larger than 1 also have the property of removing the requirement of a ranking operation. In other words, any balanced node is relieved from providing the ranking operation. However, in order to benefit from that property, it is necessary to slightly modify the representation of the WTs. The modification is that all the codeword bits that are part of a constant-depth subtree of depth h, h > 1, must be gathered at the root of that subtree. That way, during decoding, when the root n of that subtree is reached, the remaining h bits of the codeword are recovered from n at once, without the need to continue the traversal h steps down to a leaf. The larger the value of h, the more profitable it gets as, not only is n relieved of providing the ranking operation, but all its descendants, too. Modified WTs according to the proposition of Klein and Shapira are called skeleton trees (STs). Using an ST instead of a WT does not change the number of codeword bits the tree contains but it reduces the number of node that have to provide the ranking operation, allowing the omission of the associated data structures. For example, in Figure 3(c), node 2 is the root of a subtree of constant depth 2, which, not only relieves node 2 from providing the ranking operation but it does so for nodes 3 and 4, too. Figure 5 presents the modifications that must be made to the construction and decoding procedures for STs.

Given that the profitability of the constant-depth subtrees increases with their depth, it becomes clear that there is great interest in causing the appearance of constant-depth subtrees, when possible. A large number of constant-depth subtrees can...
be created by following a strategy proposed by Baruch et al. [8]. What they propose consists in changing the underlying Huffman code tree into a canonical version of it. In a canonical code tree, the leaves are ordered by codeword length, from left to right. A canonical code tree can be trivially obtained from a regular Huffman code tree by simply taking note of the codeword lengths prescribed by the Huffman code tree and preserving these lengths when building the canonical code tree. The canonical trees are clearly interesting because of their tendency to include constant-depth subtrees. Indeed, same-length codewords are made consecutive by the ordering on the codeword lengths. Figure 3(b) illustrates the canonical code tree we would obtain from the Huffman code tree of Figure 3(a). We used a stable sort where the primary key is the codeword length and the secondary one is the symbol frequency. Unfortunately in this example, this particular combination of alphabet and probabilities does not benefit from the adoption of a canonical code tree. No larger constant-depth subtrees were formed than those already present in the regular Huffman code tree. Such a lack of improvement is not common, as demonstrated by the experimental results, in Section III. The canonical code trees usually generates numerous larger constant-depth subtrees. In the example we just presented, the alphabet is too small to allow the canonical trees to trigger improvements. Still, using canonical code trees is not the best strategy for constructing code trees with constant-depth subtrees, as we explain next.

II. WAVELET TREES BASED ON POWERS OF TWO

We propose another construction technique for the code trees, which has a greater potential to generate large constant-depth subtrees. It shares many features with the technique of Baruch et al. [8]. There are many similarities: we also use the idea of the skeleton trees; we also start with a regular Huffman code tree and reorganize it; we preserve the lengths of the Huffman codewords without any modification (like the first proposition by Baruch et al. but unlike a second one, which tries to increase the size of the constant-depth subtrees by changing the codeword lengths of certain symbols). What is different in our proposition from the technique by Baruch et al. is the way we gather the leaves associated to symbols with same-length codewords. The technique we propose is based on the following observations. Observation 1.

Observation 2. The best way to generate constant-depth trees out of, say, symbols of the same codeword length consists in partitioning this set of symbols into the fewest subsets whose sizes are powers of 2. Aiming at the minimal number of subsets favours the largest powers of 2. Also, all these powers of 2 have to be distinct from each other; otherwise, the presence of two subsets of size \(2^k\) defeats minimality as they could be replaced by a single subset of size \(2^{k+1}\). Note that the partitioning of a number \(n\) into the minimal number of powers of 2 is equivalent to writing \(n\) in base 2 and choosing powers of 2 that correspond to the bits at 1. Observation 3. Given the partition of \(n\) into the powers \(2^{k_1}, 2^{k_2}, \ldots, 2^{k_j}\), one should insert the \(2^{k_1}\) most frequent symbols into the subset of size \(2^{k_1}\), the \(2^{k_2}\) next most frequent symbols into the subset of size \(2^{k_2}\), and so on down to the last subset. This selection aims at avoiding the additional data structures of the ranking operation for internal nodes that have the longest labels.

Once the set of symbols with codewords of length \(l\) has been partitioned and once this partitioning has been done for all codeword lengths, each partition can be trivially turned into a constant-depth subtree. There merely remains to join together these constant-depth subtrees to form a single code tree. Note that each junction in the last step necessarily creates an unbalanced internal node.

Let us illustrate our technique on our running example. First, let us note that there is a single symbol with a codeword of length 2, which is A, there are 5 symbols with codewords of length 3, which are B, C, D, E, and F, there is a single symbol with a codeword of length 4, which is G, and there are 2 symbols with codewords of length 5. Symbols A and G are alone in their respective categories and, as such, each forms by itself the only partition of size 1 for their category. The two symbols H and I are the only two symbols in their category and also form a partition of size 2 for their category. There remain the 5 symbols with codewords of length 3. This set should be partitioned into a subset of size 4, which should contain B, C, D, and E, and a subset of size 1, which should contain F. Given that the subsets \(\{A\}, \{B, C, D, E\}, \{F\}, \{G\}, \{H, I\}\) ought to occupy the fractions \(\frac{1}{2^1}, \frac{1}{2^2}, \frac{1}{2^4}, \frac{1}{2^7},\) and \(\frac{1}{2^8}\), respectively, of the coding space, then it is clear that the constant-depth subtrees that correspond to the subsets can easily be joined together into a single code tree. Figure 3(c)
III. EXPERIMENTAL RESULTS

Building power code trees instead of canonical code trees cannot hurt performance. This is already good news. However, we are interested to measure the improvements that are achievable. We conducted some experiments to measure how many internal nodes can be made balanced.

In order to generate large code trees, we selected four text files and turned their sets of words into the respective alphabets. Figure 6 describes the benchmark files. Two of them come from the Large Corpus while the other two come from the Canterbury Corpus. For each file, we indicate its size in bytes and in words as well as the size of the alphabet that we extract from these files. For each file, we build a regular code tree, a canonical code tree, and a power code tree. For any given file, using any of the three code trees encodes the benchmark file into a compressed file of the same size, when we measure only the codewords themselves. Figure 7, in its first column, indicates the size of the code tree (which is identical no matter the tree construction method, because the same alphabet is used) and the total size of the emitted codewords when the files get encoded (which is also identical no matter the tree construction method, because the codeword lengths are the same). The next three columns indicate the effect of the code tree on the unbalanced nodes. Each cell in these columns indicates, for a particular file–tree combination, how many internal nodes are unbalanced and the total number of bits that appear in the labels of the unbalanced nodes.

![Fig. 6. Informations about the benchmark files.](image1)

![Fig. 7. Experimental results on benchmarks with large alphabets.](image2)

![Fig. 8. Label sizes of the unbalanced nodes of the canonical and power trees.](image3)

We see that taking some care to create constant-depth subtrees reaps most of the benefits, as both the canonical and power trees do much better that the Huffman trees. However, merely using the canonical trees is not close to optimal as using power trees brings a substantial extra improvement.

A more detailed graph is presented in Figure 8 for benchmark ebib, where each unbalanced node has the size of its label reported on the curves, once for the canonical tree and once for the power tree. From the curves in the graph, we can visually observe that, not only are there fewer unbalanced nodes, but the number of bits they carry in their labels diminishes faster.

IV. FUTURE WORK

Note that we do not claim that our technique is optimal. Although it is clear that the decomposition in a minimal number of subsets of sizes $2^k$ is the best thing one can do for a set of symbols with same-length codewords, it is not clear then how to join subsets with different-length codewords in order to minimize the label sizes of the unbalanced nodes. Devising a technique to build optimal code trees with respect to the label sizes of the unbalanced nodes remains an open problem.

REFERENCES