Huffman Coding

ALISTAIR MOFFAT, The University of Melbourne

Huffman's algorithm for computing minimum-redundancy prefix-free codes has almost legendary status in the computing disciplines. Its elegant blend of simplicity and applicability has made it a favorite example in algorithms courses, and as a result it is perhaps one of the most commonly implemented algorithmic techniques. This article presents a tutorial on Huffman coding and surveys some of the developments that have flowed as a consequence of Huffman's original discovery, including details of code calculation and of encoding and decoding operations. We also survey related mechanisms, covering both arithmetic coding and the recently developed asymmetric numeral systems approach and briefly discuss other Huffman-coding variants, including length-limited codes.

CCS Concepts: • **Theory of computation** → **Design and analysis of algorithms**; **Data compression**; • **Information systems** → *Data compression*; Search index compression; • **Mathematics of computing** → Coding theory;

Additional Key Words and Phrases: Huffman code, minimum-redundancy code, data compression

ACM Reference format:
Alistair Moffat. 2019. Huffman Coding. *ACM Comput. Surv.* 52, 4, Article 85 (August 2019), 35 pages.
https://doi.org/10.1145/3342555

1 INTRODUCTION

No introductory computer science algorithms course would be complete without consideration of certain pervasive problems and discussion and analysis of the algorithms that solve them. That short list of important techniques includes heapsort and quicksort, dictionary structures using balanced search trees, Knuth-Morris-Pratt pattern search, Dijkstra's algorithm for single-source shortest paths, and, of course, David Huffman's iconic 1951 algorithm for determining minimum-cost prefix-free codes [33]—the technique known as *Huffman coding*.

The problem tackled by Huffman was—and still is—an important one. Data representation techniques are at the heart of much of computer science, and Huffman's work marked a critical milestone in the development of efficient ways for representing information. Since its publication in 1952, Huffman's seminal paper has received more the 7,500 citations,[1] and has influenced many of the compression and coding regimes that are in widespread use today in devices such as digital cameras, music players, software distribution tools, and document archiving systems.

Figure 1 shows a screen-shot illustrating a small subset of the many hundreds of images related to Huffman coding that can be found on the world-wide web, and demonstrates the ubiquity of

[1]Google Scholar, accessed August 2, 2018.

Author's address: A. Moffat, The University of Melbourne, Melbourne, Victoria 3010, Australia; email: ammoffat@unimelb.edu.au.

0360-0300/2019/08-ART85 $15.00
https://doi.org/10.1145/3342555

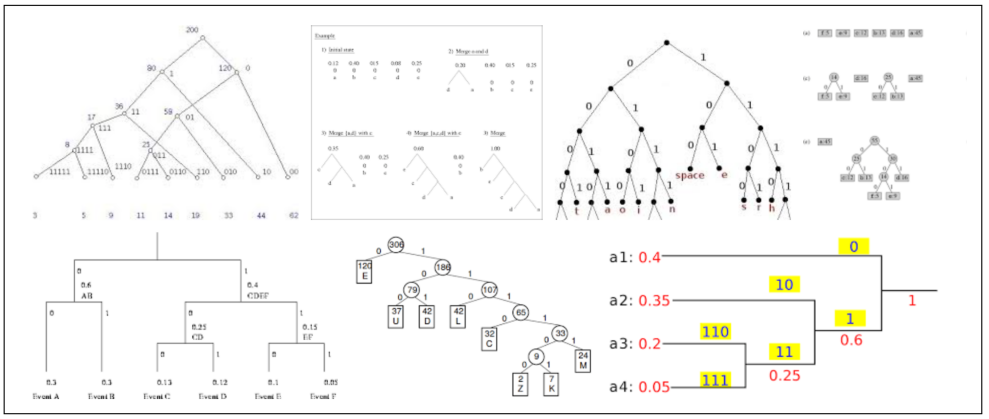

Fig. 1. A small section of the search result page returned for the query "Huffman code" at Google Image Search, captured as a screen-shot on August 5, 2018. The full result page contains approximately one thousand such images.

Huffman's tree-based approach. While an important underlying motivation for Huffman's algorithm, the prevalence of trees as a way of explaining the encoding and decoding processes is, for the most part, a distraction; and much of the focus in this article is on implementations that avoid explicit trees. It is also important to distinguish between a set of *codewords* and a set of *codeword lengths*. Defining the code as a set of codeword lengths allows a choice to be made between many different sets of complying codewords, a flexibility that is—as is explained shortly—very important as far as implementation efficiency is concerned.

The first three sections of this article provide a tutorial describing Huffman coding and its implementation. Section 4 then surveys some of the many refinements and variants that have been proposed through the six decades since Huffman's discovery. Next, Section 5 summarizes two complementary techniques for *entropy-coding*, the general task of representing messages using as few symbols as possible. Finally, Section 6 evaluates the relative strengths and drawbacks of those two newer techniques against the key properties of Huffman's famous method.

1.1 Minimum-redundancy Coding

With those initial remarks in mind, we specify the problem of *minimum-redundancy prefix-free coding* via the following definitions: A *source alphabet* (or simply, *alphabet*) of n distinct symbols denoted by the integers 0 to $n-1$ is assumed to be provided, together with a set of normally strictly positive symbol *weights*, denoted $W = \langle w_i > 0 \mid 0 \le i < n \rangle$. The weights might be integers in some cases, or fractional/real values summing to 1.0 in other cases. It will normally be assumed that the weights are non-increasing; that is, that $w_i \ge w_{i+1}$ for $0 \le i < n-1$. Situations in which this assumption is not appropriate will be highlighted when they arise; note that an ordered alphabet can always be achieved by sorting the weights and then permuting the alphabet labels to match. In a general case, we might also allow $w_i = 0$, and then during the permutation process, make those symbols the last ones in the permuted alphabet and work with a reduced alphabet of size $n' < n$.

An *output alphabet* (sometimes referred to as the *channel alphabet*) is also provided. This is often, but not always, the binary symbols $\{0, 1\}$. A *code* is a list of n positive integers $T = \langle \ell_i > 0 \mid 0 \le i < n \rangle$, with the interpretation that symbol i in the source alphabet is to be assigned a unique fixed *codeword* of length ℓ_i of symbols drawn from the output alphabet. As was already anticipated above, note that we define the code in terms of its set of codeword *lengths* and not

by its individual code *words*. A code $T = \langle \ell_i \rangle$ over the binary output alphabet $\{0, 1\}$ is *feasible*, or *potentially prefix-free,* if it satisfies

$$\mathcal{K}(T) = \sum_{i=0}^{n-1} 2^{-\ell_i} \leq 1, \tag{1}$$

an inequality that was first noted by Kraft [40] and elaborated on by McMillan [48]. If a code $T = \langle \ell_i \rangle$ is feasible, then it is possible to assign to each symbol $0 \leq i < n$ a *codeword* of length ℓ_i in a manner such that no codeword is a prefix of any other codeword, the latter being the standard definition of "prefix free code." A set of codewords *complies* with code T if the i th codeword is of length ℓ_i and none of the codewords is a prefix of any other codeword. For example, suppose that $n = 4$, that $T = \langle 2, 3, 2, 3 \rangle$ and hence that $\mathcal{K}(T) = 3/4$. Then T is feasible, and (among many other options) the codewords `01`, `000`, `10`, and `001` comply with $\langle \ell_i \rangle$. However, the set of codewords `00`, `111`, `10`, and `001` have the right lengths, but nevertheless are not compliant with $T = \langle 2, 3, 2, 3 \rangle$, because `00` is a prefix of `001`. Based on these definitions, it is useful to regard a prefix-free code as having been determined once a feasible code has been identified, without requiring that a complying codeword assignment also be specified.

The *cost* of a feasible code, denoted $C(\cdot, \cdot)$, factors in the weight associated with each symbol,

$$C(W, T) = C(\langle w_i \rangle, \langle \ell_i \rangle) = \sum_{i=0}^{n-1} w_i \cdot \ell_i. \tag{2}$$

If the w_i's are integers, and w_i reflects the frequency of symbol i in a message of total length $m = \sum_i w_i$, then $C(\cdot, \cdot)$ is the total number of channel symbols required to transmit the message using the code. Alternatively, if the w_i's are symbol occurrence probabilities that sum to one, then $C(\cdot, \cdot)$ is the expected per-input-symbol cost of employing the code to represent messages consisting of independent drawings from $0 \ldots n-1$ according to the probabilities $W = \langle w_i \rangle$.

Let $T = \langle \ell_i \mid 0 \leq i < n \rangle$ be a feasible n-symbol code. Then T is a *minimum-redundancy code* for W if, for every other n-symbol code T' that is feasible, $C(W, T) \leq C(W, T')$. Note that for any sequence W there may be multiple different minimum-redundancy codes with the same least cost.

Continuing the previous example, $T = \langle 2, 3, 3, 2 \rangle$ cannot be a minimum-redundancy code for any sequence of weights $\langle w_i \rangle$, since the feasible code $T' = \langle 2, 2, 2, 2 \rangle$ will always have a strictly smaller cost. More generally, when considering binary channel alphabets, a Kraft sum $\mathcal{K}(\cdot)$ that is strictly less than one always indicates a code that cannot be minimum-redundancy for any set of weights W, since at least one codeword can be shortened, thereby reducing $C(\cdot, \cdot)$. Conversely, a Kraft sum that is greater than one indicates a code that is not feasible—there is no possible set of complying codewords. The code $T = \langle 1, 2, 2, 2 \rangle$ is not feasible, because $\mathcal{K}(T) = 5/4$, and hence T cannot be a minimum-redundancy code for any input distribution $W = \langle w_0, w_1, w_2, w_3 \rangle$.

1.2 Fano's Challenge

The origins of Huffman coding are documented by Stix [73], who captures a tale that Huffman told to a number of people. While enrolled as a graduate student at MIT in 1951 in a class taught by coding pioneer Robert Fano, Huffman and his fellow students were told that they would be exempted from the final exam if they solved a coding challenge as part of a term paper. Not realizing that the task was an open problem that Fano had been working on himself, Huffman elected to submit the term paper. After months of unsuccessful struggle, and with the final exam just days away, Huffman threw his attempts in the bin and started to prepare for the exam. But a flash of insight the next morning had him realize that the paper he had thrown in the trash was in fact a path to a solution to the problem. Huffman coding was born at that moment, and following publication

of his paper in *Proceedings of the Institute of Radio Engineers* (the predecessor of *Proceedings of the IEEE*) in 1952, it quickly replaced the previous suboptimal Shannon-Fano coding as the method of choice for data compression applications.

2 HUFFMAN'S ALGORITHM

This section describes the principle underlying Huffman code construction; describes in increasing levels of detail how Huffman coding is implemented; demonstrates that the codes so generated are indeed minimum-redundancy; and, to conclude, considers non-binary output alphabets.

2.1 Huffman's Idea

Huffman's idea is—with the benefit of hindsight—delightfully simple. The n symbols in the input alphabet are used as the initial weights attached to a set of *leaf nodes*, one per alphabet symbol. A greedy process is then applied, with the two least-weight nodes identified and removed from the set, and combined to make a new *internal node* that is given a weight that is the sum of the weights of its two components. That new node-weight pair is then added back to the set, and the process repeated. After $n-1$ iterations of this cycle, the set contains just one node that incorporates all of the original source symbols and has an associated weight that is the sum of the original weights, $m = \sum_{i=0}^{n-1} w_i$; at this point, the process stops. The codeword length ℓ_i to be associated with symbol i can be determined as the number of times that the original leaf node for i participated in combining steps. For example, consider the set of $n = 6$ symbol weights $W = \langle 10, 6, 2, 1, 1, 1\rangle$. Using bold to represent weights, the initial leaves formed are:

$$(0, \mathbf{10}),\ (1, \mathbf{6}),\ (2, \mathbf{2}),\ (3, \mathbf{1}),\ (4, \mathbf{1}),\ (5, \mathbf{1}).$$

Assume (for definiteness and without any loss of generality) that when ties on leaves arise, higher-numbered symbols are preferred; that when ties between internal nodes and leaf nodes arise, leaves are preferred; and that when ties between internal nodes arise, the one formed earlier is preferred. With that proviso, the last two symbols are the first ones combined. Using square brackets and the original identifiers of the component symbols to indicate new nodes, the set of nodes and weights is transformed to an arrangement that contains one internal node:

$$(0, \mathbf{10}),\ (1, \mathbf{6}),\ ([4, 5], \mathbf{2}),\ (2, \mathbf{2}),\ (3, \mathbf{1}).$$

Applying the tie-breaking rule again, the second combining step joins two more of the original symbols and results in:

$$(0, \mathbf{10}),\ (1, \mathbf{6}),\ ([2, 3], \mathbf{3}),\ ([4, 5], \mathbf{2}).$$

At the next step, those two newly created internal nodes are the ones of least weight:

$$(0, \mathbf{10}),\ (1, \mathbf{6}),\ ([[2, 3], [4, 5]], \mathbf{5});$$

the fourth step combines the nodes of weight 6 and 5 to generate:

$$([1, [[2, 3], [4, 5]]], \mathbf{11}),\ (0, \mathbf{10});$$

and then a final step generates a single node that represents all six original symbols:

$$([0, [1, [[2, 3], [4, 5]]]], \mathbf{21}).$$

The depth of each source symbol in the final nesting of square brackets is the number of times it was combined, and yields the corresponding codeword length: symbol 0 is one deep, and so $\ell_0 = 1$; symbol 1 is two deep, making $\ell_1 = 2$; and the remaining symbols are four deep in the nesting. Hence, a Huffman code for the weights $W = \langle 10, 6, 2, 1, 1, 1\rangle$ is given by $T = \langle 1, 2, 4, 4, 4, 4\rangle$. As is the case with all binary Huffman codes, $\mathcal{K}(T) = 1$; the cost of this particular code is

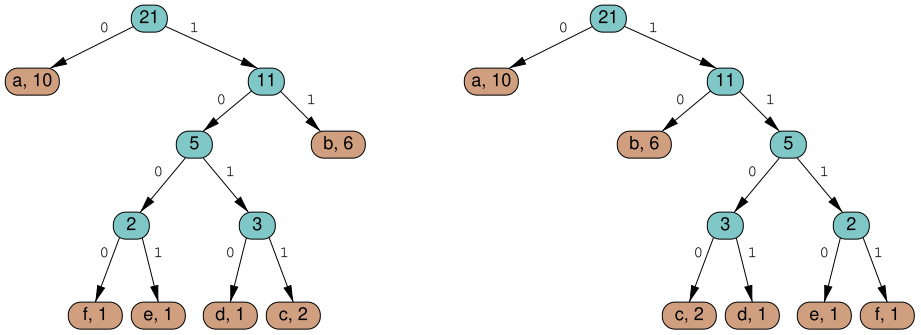

Fig. 2. Two alternative code trees for $W = \langle 10, 6, 2, 1, 1, 1\rangle$ and code $T = \langle 1, 2, 4, 4, 4, 4\rangle$. Leaves are labeled with letters corresponding to their symbol numbers and with their weights; internal nodes with their weight only. The tree on the left is generated by a strict application of Algorithm 1; the one on the right exchanges left and right children at some of the internal nodes so the leaves are sorted. The two trees represent different sets of complying codewords.

$10 \times 1 + 6 \times 2 + (2 + 1 + 1 + 1) \times 4 = 42$ bits. If the tie at the second step had been broken in a different way, the final configuration would have been:

$$([0, [1, [2, [3, [4, 5]]]]], \mathbf{21}),$$

and the code $T' = \langle 1, 2, 3, 4, 5, 5\rangle$ would have emerged. This code has different maximum codeword length to the first one, but the same cost, since $10 \times 1 + 6 \times 2 + 2 \times 3 + 1 \times 4 + (1 + 1) \times 5$ is also equal to 42. No symbol-by-symbol code can represent $W = \langle 10, 6, 2, 1, 1, 1\rangle$ in fewer than 42 bits.

2.2 Textbook Implementation

As was noted above, Huffman coding is used as an example algorithm in many algorithms textbooks. Rather than compute codeword lengths, which is the description of the problem preferred here, textbooks tend to compute binary codeword assignments by building an explicit code tree. Algorithm 1 describes this process in terms of trees and tree operations; and Figure 2 shows two such Huffman trees. The tree on the left is formed by an exact interpretation of Algorithm 1, so as each pair of elements is combined, the first node extracted from the queue is assigned to the left subtree (step 13), and the second one is assigned to the right subtree (step 14). In this example, the priority queue is assumed to comply with the tie-breaking rule that was introduced in the previous section. Note that in both trees the source alphabet $\{0, 1, 2, 3, 4, 5\}$ has been mapped to the labels $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f}\}$ to allow differentiation between symbol labels and numeric symbol weights.

The tree on the right in Figure 2 is a rearranged form of the left-hand tree, achieved by reordering the leaves according to their depths and symbol numbers and then systematically re-assigning tree edges to match. In this example (but not always, as is noted in Section 2.5), the rearrangement is achieved by swapping left and right edges at some of the internal nodes. In either of the two trees shown in Figure 2, a set of complying codewords—satisfying the prefix-free property and possessing the required distribution of codeword lengths—can easily be generated. Using the usual convention that a left edge corresponds to a `0` bit and right edge to a `1` bit, the codewords in the right-hand tree are given by $\langle$`0`, `10`, `1100`, `1101`, `1110`, `1111`$\rangle$, and—because of the rearrangement—are in lexicographic order. This important "ordered leaves" property will be exploited in the implementations described in Section 3.

Other trees emerge if ties are broken in different ways, but all resulting codes have the same cost. Any particular tie-breaking rule simply selects one among those equal-cost choices. The rule

ALGORITHM 1: Compute Huffman codeword lengths, textbook version.

```
0:  function CalcHuffLens(W, n)
1:      // initialize a priority queue, create and add all leaf nodes
2:      set Q ← [ ]
3:      for each symbol s ∈ ⟨0 . . . n − 1⟩ do
4:          set node ← new(leaf)
5:          set node.symb ← s
6:          set node.wght ← W[s]
7:          Insert(Q, node)
8:      // iteratively perform greedy node-merging step
9:      while |Q| > 1 do
10:         set node_0 ← ExtractMin(Q)
11:         set node_1 ← ExtractMin(Q)
12:         set node ← new(internal)
13:         set node.left ← node_0
14:         set node.rght ← node_1
15:         set node.wght ← node_0.wght + node_1.wght
16:         Insert(Q, node)
17:     // extract final internal node, encapsulating the complete hierarchy of mergings
18:     set node ← ExtractMin(Q)
19:     return node, as the root of the constructed Huffman tree
```

given above—preferring symbols with high indexes to symbols with lower indexes and preferring leaves to internal nodes—has the useful side effect of generating a code with the smallest maximum codeword length, $L = \max_{0 \le i < n} \ell_i$.

The edge rearrangements employed in Figure 2 mean that multiple complying codeword sets can be generated for every sequence of weights. Indeed, since there are $n - 1$ internal nodes in every binary tree with n leaves and each internal node has two alternative orientations, at least 2^{n-1} different arrangements of a Huffman tree can be achieved via edge swaps alone. Even more codes are possible if non-sibling leaves at the same depth are swapped with each other, which can be done once the code has been derived, even if those nodes have different weights.

Algorithm 1 makes use of a priority queue data structure, denoted Q in the pseudo-code, and standard priority queue operations to insert a new object and to identify and delete the item of smallest weight. A total of $2n - 1$ queue *Insert* operations, and $2n - 1$ queue *ExtractMin* operations are required to construct the final tree of n leaves and $n - 1$ internal nodes. This formulation suggests that the binary heap is a suitable queue structure, supporting *Insert* and *ExtractMin* operations in $O(\log n)$ time each, and hence allowing Huffman codes (via a traversal of the Huffman tree) to be computed in $O(n \log n)$ time.

At this point, most algorithms textbooks move on to their next topic, leaving the impression that Huffman codes are constructed using an $O(n \log n)$-time heap-based algorithm; that encoding is carried out by tracing the path in a tree from the root through to a specified leaf; and that decoding is also performed by following edges in the Huffman tree, this time as bits are fetched one-by-one from the compressed data stream. The same is true for the Wikipedia article on Huffman coding.[2] The remainder of this section addresses the first of these misconceptions; and then Section 3 examines the mechanics of encoding and decoding.

[2] https://en.wikipedia.org/wiki/Huffman_coding, accessed August 17, 2018.

2.3 van Leeuwen's Approach

Nearly 25 years after Huffman, van Leeuwen [79] recognized that a heap-based priority queue was not required if the input weights were presented in sorted order and that $O(n)$ time was sufficient. The critical observation that makes this linear-time approach possible is that the internal nodes constructed at step 12 in Algorithm 1 are generated in non-decreasing weight order and are also consumed in non-decreasing order. That is, all that is required to handle the internal nodes is a simple queue. Hence, if the input weights are provided in sorted order, two first-in first-out queues can be used for the priority queue structure: one storing the leaves and built from the initial weights, inserted in increasing weight order; and one containing the internal nodes that are generated at step 12, also inserted in increasing weight order. Each *ExtractMin* operation then only needs to compare the two front-of-queue items and take the smaller of the two. Since *Append* and *ExtractHead* operations on queues can be readily achieved in $O(1)$ time each, the $4n - 2$ such operations required by Algorithm 1 in total consume $O(n)$ time.

As already noted, van Leeuwen's approach is applicable whenever the input weights are sorted. This means that Algorithm 1 need never be implemented via a heap data structure, since an $O(n \log n)$-time pre-sort of the symbol weights followed by the $O(n)$-time sequential construction mechanism has the same asymptotic cost and is simpler in practice. So, while it is perfectly legitimate to seek a minimum redundancy code for (say) the set of weights $W = \langle 99, 1, 99, 1, 99, 1 \rangle$, the best way to compute the answer is to develop a code T' for the permuted weights $W' = \langle 99, 99, 99, 1, 1, 1 \rangle$ and then de-permute to obtain the required code T. In the remainder of this article, we assume that the weights are presented in non-increasing order. It is worth noting that the example given by Huffman in his paper in 1952 similarly makes use of a sorted input alphabet and in effect merges two sorted lists [33, Table 1]. (As an interesting sidelight, Klein and Shapira [38] consider the compression loss that is incurred if a "sorted input required" construction algorithm is applied to an unsorted sequence of weights.)

2.4 In-place Implementation

In 1995, the approach of van Leeuwen was taken a step further, and an $O(n)$-time *in-place* Huffman code computation was described [56]. Algorithm 2 provides a detailed explanation of this process and shows how it uses just a small number of auxiliary variables to compute a code. Starting with an input array $W[0 \ldots n-1]$ containing the n symbol weights w_i, three sequential passes are made, each transforming the array into a new form. Elements in the array are used to store, at different times, input weights, weights of internal nodes, parent locations of internal nodes, internal node depths, and, finally, leaf depths. All of this processing is carried out within the same n-element array without a separate data structure being constructed.

The three phases are marked by comments in the pseudo-code. In the first phase, from steps 2 to 12, items are combined into pairs, drawing from two queues: the original weights, stored in decreasing-weight order in $W[0 \ldots leaf]$; and the internal node weights, also in decreasing order, stored in $W[next + 1 \ldots root]$. At first there are no internal weights to be stored, and the entire array is leaves. At each iteration of the loop, steps 5 to 10 compare the next smallest internal node (if one exists) with the next smallest leaf (if one exists) and choose the smaller of the two. This value is assigned to $W[next]$; and then the next smallest value is added to it at step 12. If either of these two is already an internal node, then it is replaced in W by the address of its parent, *next*, as a parent-pointered tree skeleton is built, still using the same array. At the end of this first phase, $W[0]$ is unused; $W[1]$ is the weight of the root node of the Huffman tree and is the sum of the original w_i values; and the other $n - 2$ elements correspond to the internal nodes of the Huffman tree below the root, with $W[i]$ storing the offset in W of its parent.

ALGORITHM 2: Compute Huffman codeword lengths, in-place linear-time version [56].

```
 0: function CalcHuffLens(W, n)
 1:     // Phase 1
 2:     set leaf ← n − 1, and root ← n − 1
 3:     for next ← n − 1 downto 1 do
 4:         // find first child
 5:         if leaf < 0 or (root > next and W[root] < W[leaf]) then
 6:             // use internal node
 7:             set W[next] ← W[root], and W[root] ← next, and root ← root − 1
 8:         else
 9:             // use leaf node
10:             set W[next] ← W[leaf], and leaf ← leaf − 1
11:         // find second child
12:         repeat steps 5–10, but adding to W[next] rather than assigning to it
13:     // Phase 2
14:     set W[1] ← 0
15:     for next ← 2 to n − 1 do
16:         set W[next] ← W[W[next]] + 1
17:     // Phase 3
18:     set avail ← 1, and used ← 0, and depth ← 0, and root ← 1, and next ← 0
19:     while avail > 0 do
20:         // count internal nodes used at depth depth
21:         while root < n and W[root] = depth do
22:             set used ← used + 1, and root ← root + 1
23:         // assign as leaves any nodes that are not internal
24:         while avail > used do
25:             set W[next] ← d, and next ← next + 1, and avail ← avail − 1
26:         // move to next depth
27:         set avail ← 2 · used, and depth ← depth + 1, and used ← 0
28:     return W, where W[i] now contains the length ℓ_i of the i th codeword
```

Figure 3 gives an example of code construction, and shows several snapshots of the array W as Algorithm 2 is applied to the 10-sequence $W = \langle 20, 17, 6, 3, 2, 2, 2, 1, 1, 1\rangle$. By the end of phase 1, for example, $W[6]$ represents an internal node whose parent is represented in $W[4]$. The internal node at $W[6]$ has one internal node child represented at $W[9]$, and hence also has one leaf node as a child, which is implicit and not actually recorded anywhere. In total, $W[6]$ is the root node of a subtree that spans three of the original symbols.

The second phase—at steps 14 to 16—traverses that tree from the root down, converting the array of parent pointers into an array of internal node depths. Element $W[0]$ is again unused; by the end of this phase, the other $n - 1$ elements reflect the depths of the $n - 1$ internal nodes, with the root, represented by $W[1]$, having a depth of zero. Node depths are propagated downward through the tree via the somewhat impenetrable statement at step 16: "$W[next] \leftarrow W[W[next]] + 1$."

Steps 18 to 27 then process the array a third time, converting internal node depths into leaf depths. Quantity *avail* records the number of unused slots at level *depth* of the tree, starting with values of one and zero, respectively. As each level of the tree is processed, some of the *avail* slots are required to accommodate the required number of internal nodes; the rest must be leaves, and so can be assigned to the next slots in W as leaf depths. The number of available slots at the next depth is then twice the number of internal nodes at the current depth.

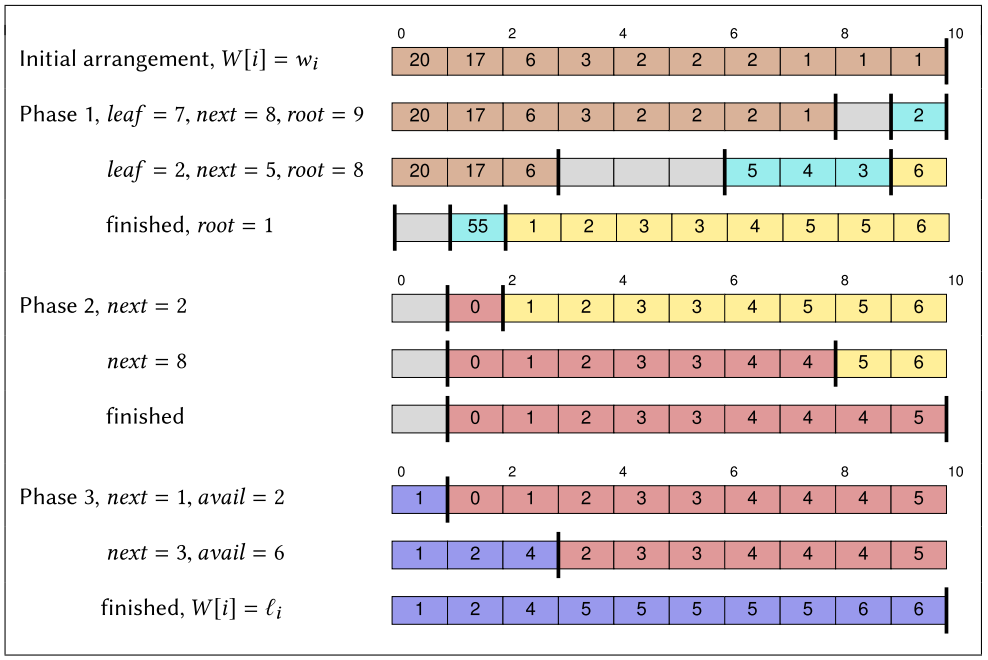

Fig. 3. Tracing Algorithm 2 for the input $W = \langle 20, 17, 6, 3, 2, 2, 2, 1, 1, 1\rangle$. The first row shows the initial state of the array, with brown elements indicating $W[i] = w_i$. During phase 1, the light blue values indicate internal node weights before being merged; and yellow values indicate parent pointers of internal nodes after they have been merged. Pink values generated during phase 2 indicate depths of internal nodes; and the purple values generated during phase 3 indicate depths of leaves. Grey is used to indicate elements that are unused. The final set of codeword lengths is $T = \langle 1, 2, 4, 5, 5, 5, 5, 5, 6, 6\rangle$.

At the conclusion of the third phase, each original symbol weight w_i in array element $W[i]$ has been over-written by the corresponding codeword length ℓ_i of a Huffman code. What is particularly notable is that a complete working implementation of Algorithm 2 is only a little longer than the pseudo-code that is shown here. There is no need for trees, pointers, heaps, or dynamic memory, and it computes quickly in $O(n)$ time.

The presentation in this subsection is derived from the description of Moffat and Katajainen [56]; Section 4.3 briefly summarizes some other techniques for computing minimum-redundancy codes.

2.5 Assigning Codewords

The definition of a code as being a set of n codeword lengths is a deliberate choice and means that a lexicographic ordering of codewords can always be used—a benefit that is not available if the codeword assignment must remain faithful to the tree generated by a textbook (Algorithm 1) implementation of Huffman's algorithm. Table 1 illustrates this idea, using the sequence of codeword lengths developed in the example shown in Figure 3.

Generation of a set of lexicographically ordered codewords from a non-decreasing feasible code $T = \langle \ell_i \rangle$ is straightforward [15, 69]. Define $L = \max_{i=0}^{n-1} \ell_i$ to be the length of the longest codewords required; in the case when the weights are non-increasing, that means $L = \ell_{n-2} = \ell_{n-1}$. In the example, $L = 6$. A codeword of length ℓ_i can then be thought of as being either a right-justified ℓ_i-bit integer or a left-justified L-bit integer. The rightmost column in Table 1 shows the latter, and

Table 1. Canonical Assignment of Codewords for the Example Code $T = \langle 1, 2, 4, 5, 5, 5, 5, 5, 6, 6\rangle$, with a Maximum Codeword Length of $L = \max_{i=0}^{n-1} \ell_i = 6$

i	w_i	ℓ_i	codeword	ℓ_i-bit integer	L-bit integer
0	20	1	0	0	0
1	17	2	10	2	32
2	6	4	1100	12	48
3	3	5	11010	26	52
4	2	5	11011	27	54
5	2	5	11100	28	56
6	2	5	11101	29	58
7	1	5	11110	30	60
8	1	6	111110	62	62
9	1	6	111111	63	63
10		– *sentinel* –		64	64

The sentinel value in the last row is discussed in Section 3.2.

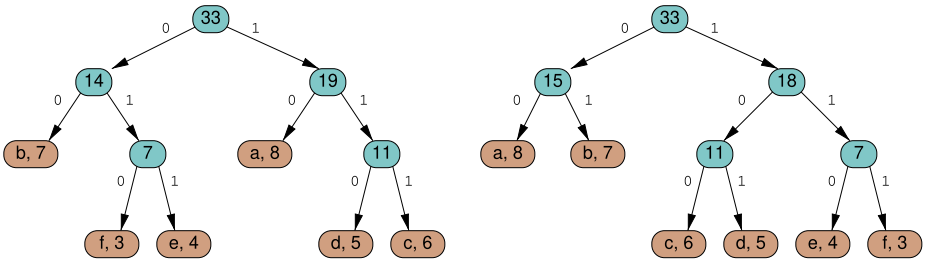

Fig. 4. Two alternative code trees for $W = \langle 8, 7, 6, 5, 4, 3\rangle$ and minimum-redundancy code $T = \langle 2, 2, 3, 3, 3, 3\rangle$. Leaves are again labeled with an alphabetic symbol identifier and numeric weight; internal nodes with their weight only. The right-hand tree cannot be generated by Algorithm 1.

the column to the left of it shows the former. The first of the L-bit integers, corresponding to the most frequent symbol, is always zero; thereafter, the $i + 1$ st L-bit integer is computed by adding $2^{L-\ell_i}$ to the L-bit integer associated with the i th symbol. For example, in the table, the 32 in the last column in the second row is the result of adding 2^{6-1} to the zero at the end of the first row.

Once the set of L-bit integers has been computed, the corresponding ℓ_i-bit values are found by taking the first ℓ_i bits of the L-bit integer. Those ℓ_i-bit integers are exactly the bitstrings assigned in the column headed "codeword". To encode an instance of i th symbol, the ℓ_i low-order bits of the i th value from the "ℓ_i-bit integer" column of the table are appended to an output buffer. Section 3.1 describes the encoding process in more detail.

The codewords implied by the right-hand tree in Figure 2 were assigned in this structured manner, meaning that the leaf depths, symbol identifiers, and codewords themselves (as L-bit integers) are all in the same order. The result is referred to as a *canonical code*, the "ordered leaves" tree arrangement that was mentioned in Section 2.2. In the example shown in Figure 2, the canonical code could be obtained from the Huffman tree via left-right child-swaps at internal node. But such rearrangement is not always possible. For example, consider the weights $W = \langle 8, 7, 6, 5, 4, 3\rangle$. Figure 4 shows two codeword assignments for those weights: on the left as a result of the application of Algorithm 1, and on the right as a result of the application of Algorithm 2 to obtain codeword lengths, followed by sequential assignment of canonical codewords. The internal nodes

in the two trees have different weights, and there is no sequence of left-right child swaps that transforms one to the other, even though the two codes have the same cost.

The desire to work with the regular codewords patterns provided by canonical codes is why Section 1.1 defines a code as sequences of codeword lengths rather than as particular sets of complying codewords. To a purist, minimum-redundancy codes and Huffman codes are different, because the codeword assignment $\langle \texttt{00}, \texttt{01}, \texttt{100}, \texttt{101}, \texttt{110}, \texttt{111} \rangle$ that is a canonical minimum-redundancy code for $W = \langle 8, 7, 6, 5, 4, 3 \rangle$ cannot be generated by Algorithm 1. That is, an application of Huffman's algorithm will always create a minimum-redundancy code, but not every possible minimum-redundancy code can emerge from an application of Huffman's algorithm. But for fast decoding, discussed in Section 3.2, the tightly structured arrangement of codewords shown in Table 1 is desirable, and if the definition of a code is as a set of codeword lengths, then the somewhat arbitrary distinction between "minimum-redundancy" and "Huffman" codes becomes irrelevant. That reasoning is why we deliberately blend the two concepts here.

2.6 Optimality

To demonstrate that Huffman's algorithm does indeed result in a minimum-redundancy code, two steps are required [33]. The first step is to confirm the *sibling property* [25], which asserts that a minimum-redundancy code exists in which the two least-weight symbols are siblings and share a common parent in the corresponding binary code tree. The second step is to verify that joining those two symbols into a combined node with weight given by their sum, then constructing a minimum-redundancy code for the reduced-by-one symbol set, then expanding that symbol again into its two components, yields a minimum-redundancy code for the original set of symbols. The inductive principle takes care of the remainder of the proof, because a minimum-redundancy code for the case $n = 2$ cannot be anything other than $\langle \ell_i \rangle = \langle 1, 1 \rangle$; that is, the two codewords `0` and `1`.

Consider the sibling property. Suppose that $T = \langle \ell_i \rangle$ is known to be a minimum-redundancy code for the n-sequence $W = \langle w_i \rangle$. Since every internal node in the code tree must have two children (because if it did not, a whole subtree could be promoted to make a cheaper code), there must be at least two nodes at the greatest depth, $L = \max_i \ell_i$. Now suppose, without loss of generality, that w_{n-1} and w_{n-2} are the two lowest weights, possibly equal. If the leaves for both of these symbols are at depth L (that is, $\ell_{n-2} = \ell_{n-1} = L$), then they can be moved in the code tree via a leaf relabeling process to make them both children of the same internal node in a way that does not alter the cost $C(\cdot, \cdot)$ of the code. This is sufficient to satisfy the sibling property.

However, if (say) $\ell_{n-1} = L$ and $\ell_{n-2} < L$, then there must be a different symbol a such that $\ell_a = L$; that is, there must be a symbol a at the deepest level of the code tree that is neither symbol $n-2$ nor symbol $n-1$. Now consider the code T' formed by exchanging the lengths of the codewords assigned to symbols a and $n-2$. The new code may have an altered cost compared to the old code; if so, the difference is given by

$$\begin{aligned} C(W, T') - C(W, T) \\ &= (w_a \cdot \ell_{n-2} + w_{n-2} \cdot \ell_a) - (w_a \cdot \ell_a + w_{n-2} \cdot \ell_{n-2}) \\ &= (w_a - w_{n-2}) \cdot (\ell_{n-2} - \ell_a) \\ &\geq 0, \end{aligned}$$

with the final inequality holding, because the original code T is minimum-redundancy for W, meaning that no other code can have a smaller cost. But $w_a \geq w_{n-2}$ and $\ell_{n-2} < \ell_a$ in both cases as a result of the assumptions made, and hence it can be concluded that $w_a = w_{n-2}$. That is, symbol a and symbol $n-2$ must have the same weight and can have their codeword lengths exchanged without altering the cost of the code. With that established, the sibling property can be confirmed.

Consider the second inductive part of the argument, and suppose that

$$T_1 = \langle \ell_0, \ldots, \ell_{n-2}, \ell_{n-1} \rangle$$

is an n-element minimum-redundancy code of cost $C(W_1, T_1)$ for the n weights

$$W_1 = \langle w_0, \ldots, w_{n-3}, w_{n-2}, w_{n-1} + x \rangle,$$

for some set of weights such that $w_0 \geq w_1 \geq \cdots \geq w_{n-1} \geq x > 0$. Starting with the n-symbol feasible code T_1, now form the $(n+1)$-symbol feasible code

$$T_2 = \langle \ell_0, \ldots, \ell_{n-3}, \ell_{n-2}, \ell_{n-1} + 1, \ell_{n-1} + 1 \rangle.$$

By construction, the extended code T_2 has cost $C(W_2, T_2) = C(W_1, T_1) + w_{n-1} + x$ for the $(n+1)$-sequence

$$W_2 = \langle w_0, \ldots, w_{n-3}, w_{n-2}, w_{n-1}, x \rangle.$$

Suppose next that T_3 is a minimum-redundancy code for W_2. If T_2 is *not* also a minimum-redundancy code, then

$$C(W_2, T_3) < C(W_2, T_2) = C(W_1, T_1) + w_{n-1} + x.$$

But this leads to a contradiction, because the sibling property requires that there be an internal node of weight $w_{n-1} + x$ in the code tree defined by T_3, since w_{n-1} and x are the two smallest weights in W_2. And once identified, that internal node could be replaced by a leaf of weight $w_{n-1} + x$ without altering any other part of the tree, and hence would give rise to an n-element code T_4 of cost

$$C(W_1, T_4) = C(W_2, T_2) - w_{n-1} - x < C(W_1, T_1),$$

and that would mean in turn that T_1 could *not* be a minimum-redundancy code for W_1.

In combination, these two arguments demonstrate that the codes developed by Huffman are indeed minimum-redundancy—and that he fully deserved his subject pass in 1951.

2.7 Compression Effectiveness

The previous subsection demonstrated that Huffman's algorithm computes minimum-redundancy codes. The next question to ask is, how good are they?

In a foundational definition provided by information theory pioneer Claude Shannon [70], the *entropy* of a set of n weights $W = \langle w_i \rangle$ is given by

$$\mathcal{H}(W) = -\sum_{i=0}^{n-1} w_i \cdot \log_2 \frac{w_i}{m}, \tag{3}$$

where $m = \sum_{i=0}^{n-1} w_i$ is the sum of the weights, and where $-\log_2(w_i/m) = \log_2(m/w_i)$ is the *entropic cost* (in bits) of one instance of a symbol that is expected to occur with probability given by w_i/m.

If $m = 1$, and the w_i's are interpreted as symbol probabilities, then the quantity $\mathcal{H}(W)$ has units of bits per symbol and represents the expected number of output symbols generated per source symbol. If the w_i's are integral occurrence counts and m is the length of the message that is to be coded, then $\mathcal{H}(W)$ has units of bits and represents the minimum possible length of the compressed message when coded relative to its own statistics.

Given such a sequence W of n weights, the *relative effectiveness loss* $\mathcal{E}(W, T)$ of a feasible code $T = \langle \ell_i \rangle$ is the fractional difference between the cost $C(W, T)$ of that code and the entropy of W:

$$\mathcal{E}(W, T) = \frac{C(W, T) - \mathcal{H}(W)}{\mathcal{H}(W)}. \tag{4}$$

A relative effectiveness loss of zero indicates that symbols are being coded in their entropic costs; values larger than zero indicate that the coder is admitting some degree of compression leakage.

Table 2. Entropy, Cost, and Relative Effectiveness Loss of Example Huffman Codes

Weights $W = \langle w_i \rangle$	Code $T = \langle \ell_i \rangle$	$\mathcal{H}(W)/m$	$C(W,T)/m$	$\mathcal{E}(W,T)$
$\langle 10, 6, 2, 1, 1, 1 \rangle$	$\langle 1, 2, 4, 4, 4, 4 \rangle$	1.977	2.000	1.2%
$\langle 20, 17, 6, 3, 2, 2, 2, 1, 1, 1 \rangle$	$\langle 1, 2, 4, 5, 5, 5, 5, 5, 6, 6 \rangle$	2.469	2.545	3.1%
$\langle 99, 99, 99, 1, 1, 1 \rangle$	$\langle 2, 2, 2, 3, 4, 4 \rangle$	1.666	2.017	21.1%
$\langle 8, 7, 6, 5, 4, 3 \rangle$	$\langle 2, 2, 3, 3, 3, 3 \rangle$	2.513	2.545	1.3%

To allow comparison, both entropy and cost are normalized to "average bits-per-symbol" values; and relative effectiveness loss is expressed as a percentage, with big values worse than small values.

Table 2 shows the result of these calculations for some of the sets of weights that have been used as examples elsewhere in this article. In many situations, including three of the four illustrated examples, Huffman codes provide compression effectiveness that is within a few percentage points of the entropy-based lower bound. The most egregious exceptions occur when the weight of the most frequent symbol, w_0, is large relative to the sum of the remaining weights; that is, when w_0/m becomes large. Indeed, it is possible to make the relative effectiveness loss arbitrarily high by having $w_0/m \to 1$. For example, the five-sequence $W = \langle 96, 1, 1, 1, 1 \rangle$ has an entropy of $\mathcal{H}(W) = 32.2$ bits, a minimum redundancy code $T = \langle 1, 3, 3, 3, 3 \rangle$ with cost $C(W,T) = 108$ bits, and hence a relative effectiveness loss of $\mathcal{E}(W,T) = 235\%$. While T is certainly "minimum-redundancy," it is a long way from being good. Even the $W = \langle 99, 99, 99, 1, 1, 1 \rangle$ example suffers from non-trivial loss of effectiveness. Other coding approaches that have smaller relative effectiveness loss in this kind of highly skewed situation are discussed in Section 5.

A number of bounds on the effectiveness of Huffman codes have been developed. For example, in an important followup to Huffman's work, Gallager [25] shows that for a set of n weights $W = \langle w_i \rangle$ summing to $m = \sum_{i=0}^{n-1} w_i$, and with corresponding minimum-redundancy code $T = \langle \ell_i \rangle$:

$$C(W,T) - \mathcal{H}(W) \le \begin{cases} w_0 + 0.086 \cdot m, & \text{when } w_0 < m/2, \\ w_0, & \text{when } w_0 \ge m/2\,. \end{cases}$$

This relationship can then be used to compute an upper limit on the relative effectiveness loss. Capocelli and De Santis [9] and Manstetten [47] have also studied code redundancy.

2.8 Non-binary Output Alphabets

The examples thus far have assumed that the channel alphabet is binary and consists of "`0`" and "`1`." Huffman [33] also considered the more general case of an r-ary output alphabet, where $r \ge 2$ is specified as part of the problem instance, and the channel alphabet is the symbols $\{0, 1, \ldots, r-1\}$.

Huffman noted that if each internal node has r children, then the final tree must have $k(r-1)+1$ leaves for some integral k. Hence, if an input of n weights $W = \langle w_i \rangle$ is provided, an augmented input W' of length $n' = (r-1)\lceil (n-1)/(r-1) \rceil + 1$ is created, extended by the insertion of $n' - n$ dummy symbols of weight $w'_n = w'_{n+1} = \cdots = w'_{n'-1} = 0$, with symbol weights of zero permitted in this scenario. Huffman's algorithm (Algorithm 1 or Algorithm 2) is then applied, but joining least-cost groups of r nodes at a time, rather than groups of two. That is, between 0 and $r-2$ dummy symbols are appended, each of weight zero, before starting the code construction process.

For example, consider the sequence $W = \langle 20, 17, 6, 3, 2, 2, 2, 1, 1, 1 \rangle$ already used as an example in Figure 3. It has $n = 10$ weights, so if an $r = 5$ code is to be generated, then the augmented sequence $W' = \langle 20, 17, 6, 3, 2, 2, 2, 1, 1, 1, 0, 0, 0 \rangle$ is formed with three additional symbols to make a total size of $n' = 13$. Three combining steps are then sufficient to create the set of codeword lengths $T' = \langle 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3 \rangle$. From these, a canonical code can be constructed over the channel alphabet $\{0, 1, 2, 3, 4\}$, yielding the 10 codewords $\langle$`0`, `1`, `2`, `3`, `40`, `41`, `42`, `43`, `440`, `441`$\rangle$,

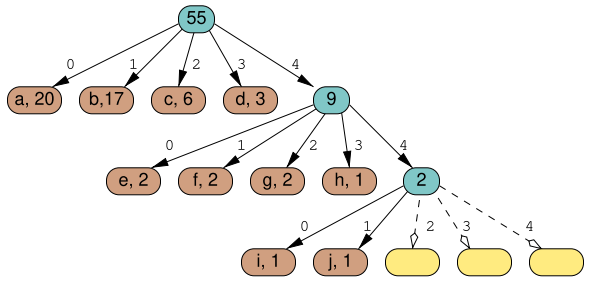

Fig. 5. Radix-5 minimum-redundancy canonical code tree for the weights $W = \langle 20, 17, 6, 3, 2, 2, 2, 1, 1, 1\rangle$. Three dummy nodes are required as children of the rightmost leaf to create an extended sequence W' containing $n' = 13$ leaves. Leaves are labeled with an alphabetic letter corresponding to their integer source symbol identifier.

with three further codewords (442, 443, and 444) nominally assigned to the three added "dummy" symbols, and hence unused. Figure 5 shows the resultant code tree.

The equivalent of the Kraft inequality (Equation (1)) is now given by

$$\mathcal{K}(T) = \sum_{i=0}^{n-1} r^{-\ell_i} \le 1\,, \tag{5}$$

with equality only possible if $n = n'$, and n is already one greater than a multiple of $r - 1$. Similarly, in the definition of entropy in Equation (3), the base of the logarithm changes from 2 to r when calculating the minimum possible cost in terms of expected r-ary output symbols per input symbol.

One interesting option is to take $r = 2^8 = 256$, in which case what is generated is a Huffman code in which each output symbol is a byte. For large alphabet applications in which even the most frequent symbol is relatively rare—for example, when the input tokens are indices into a dictionary of natural language words—the relative effectiveness loss of such a code might be small, and the ability to focus on whole bytes at decode-time can lead to a distinct throughput advantage [21].

2.9 Other Resources

Two previous surveys provide summaries of the origins of Huffman coding, and data compression in general—those of Lelewer and Hirschberg [42] and Bell et al. [6]. A range of textbooks cover the general area of compression and coding, including work by Bell et al. [5], by Storer [74], by Sayood [67], by Witten et al. [84], and by Moffat and Turpin [63].

3 ENCODING AND DECODING MINIMUM-REDUNDANCY CODES

We now consider the practical use of binary minimum-redundancy codes in data compression systems. The first subsection considers the encoding task; the second the decoding task; and then the third considers the question of how the code $T = \langle \ell_i \rangle$ can be economically communicated from encoder to decoder. Throughout this section, it is assumed that the code being applied is a canonical one in which the set of codewords is lexicographically sorted.

3.1 Encoding Canonical Codes

The "textbook" description of Huffman coding as being a process of tracing edges in a binary tree is costly in a number of ways: explicit manipulation of a tree requires non-trivial memory space; traversing a pointer in a large data structure for each generated bit may involve cache misses; and writing bits one at a time to an output file is slow. Table 1 in Section 2.5 suggests how these difficulties can be resolved. In particular, suppose that the column in that table headed "ℓ_i" is

Table 3. Tables *first_symbol*[], *first_code_r*[], and *first_code_l*[] for the Canonical Code Shown in Table 1

ℓ	*first_symbol*[ℓ]	*first_code_r*[ℓ]	*first_code_l*[ℓ]
0	0	0	0
1	0	0	0
2	1	2	32
3	2	6	48
4	2	12	48
5	3	26	52
6	8	62	62
7	—	64	64

The last row is a sentinel to aid with loop control.

available in an array *code_len*[], indexed by symbol identifier, and that the column headed "ℓ_i-bit integer" is stored in a parallel array *code_word*[]. To encode a particular symbol $0 \le s < n$, all that is then required is to extract the *code_len*[s] low-order bits of the integer in *code_word*[s]:

set $\ell \leftarrow$ *code_len*[s];
putbits(*code_word*[s], ℓ);

where *putbits*(*val*, *count*) writes the *count* low-order bits from integer *val* to the output stream and is typically implemented using low-level mask and shift operators. This simple process both eliminates the need for an explicit tree and means that each output cycle generates a whole codeword, rather than just a single bit.

Storage of the array *code_len*[] is relatively cheap—one byte per value allows codewords of up to $L = 255$ bits and is almost certainly sufficient. Other options are also possible [24]. But the array *code_word*[] still has the potential to be expensive, because even 32-bits per value might not be adequate for the codewords associated with a large or highly skewed alphabet. Fortunately, *code_word*[] can also be eliminated and replaced by two compact arrays of just $L + 1$ entries each, where (as before) L is the length of a longest codeword.

Table 3 provides an example of these two arrays: *first_symbol*[], indexed by a codeword length ℓ, storing the source alphabet identifier of the first codeword of length ℓ; and the right-aligned first codeword of that length, *first_code_r*[], taken from the column in Table 1 headed "ℓ_i-bit integer." For example, in Table 1, the first codeword of length $\ell = 5$ is for symbol 3, and its 5-bit codeword is given by the five low-order bits of the integer 26, or "`11010`."

With these two arrays, encoding symbol s is achieved using:

set $\ell \leftarrow$ *code_len*[s];
set *offset* $\leftarrow s -$ *first_symbol*[ℓ];
putbits(*first_code_r*[ℓ] + *offset*, ℓ).

Looking at Tables 1 and 3 together, to encode (say) symbol $s = 6$, the value *code_len*[6] is accessed, yielding $\ell = 5$; the subtraction $6 -$ *first_symbol*[5] $= 3$ indicates that symbol 6 is the third of the 5-bit codewords; and then the 5 low-order bits of *first_code_r*[5] + 3 = 26 + 3 = 29 are output. Those five bits ("`11101`") are the correct codeword for symbol 6.

If encode-time memory space is at a premium, the n bytes consumed by the *code_len*[] array can also be eliminated, but at the expense of encoding speed. One of the many beneficial consequences of focusing on canonical codes is that the *first_symbol*[] array is a sorted list of symbol numbers. Hence, given a symbol number s, the corresponding codeword length can be determined by linear or binary search in *first_symbol*[], identifying the value ℓ such that

first_symbol[ℓ] $\le s <$ *first_symbol*[$\ell + 1$]. The search need not be over the full range and can be constrained to valid codeword lengths. The shortest codeword length $\ell_{\min}$ is given by $\min_i \ell_i = \ell_0$, and the searched range can thus be restricted to $\ell_{\min} \ldots L$. When that range is small, linear search may be just as efficient as binary search; moreover, the values being searched may be biased (because the weights are sorted, and small values of ℓ correspond to frequently occurring symbols) in favor of small values of ℓ, a further reason why linear search might be appropriate.

Note that the encoding techniques described in this section only apply if the source alphabet is sorted, the symbol weights are non-increasing, and the canonical minimum-redundancy codewords are thus also lexicographically sorted (as shown in Table 1). In some applications, those relatively strong assumptions may not be valid, in which case a further n words of space must be allocated for a permutation vector that maps source identifiers to sorted symbol numbers, with the latter then used for the purposes of the canonical code. If a permutation vector is required, it dominates the cost of storing *code_len*[i] by a factor of perhaps four, and the savings achieved by removing *code_len*[i] may not be warranted.

A range of authors have contributed to the techniques described in this section, with the early foundations laid by Schwartz and Kallick [69] and Connell [15]. Hirschberg and Lelewer [30] and Zobel and Moffat [86] also discuss the practical aspects of implementing Huffman coding. The approach described here is as presented by Moffat and Turpin [61], who also describe the decoding process that is explained in the next subsection and, in a separate paper, the prelude representations that are discussed in Section 3.3.

3.2 Decoding Canonical Codes

The constrained structures of canonical codes mean that it is also possible to avoid the inefficiencies associated with tree-based bit-by-bit processing during decoding. Now it is the "left-justified in L bits" form of the codewords that are manipulated as integers, starting with the column in Table 1 headed "L-bit integer" and extracted into the array *first_code_l*[] that is shown in Table 3. The *first_symbol*[] array is also used during decoding.

Making use of the knowledge that no codeword is longer than L bits, a variable called *buffer* is employed that always contains the next L undecoded bits from the compressed bit-stream. The decoder uses *buffer* to determine a value for ℓ, the number of bits in the next codeword; then it identifies the symbol that corresponds to that codeword; and finally replenishes *buffer* by shifting/masking out the ℓ bits that have been used and fetching ℓ more bits from the input. A suitable sentinel value is provided in *first_code_l*[$L + 1$] to ensure that the search process required by the first step is well defined:

identify ℓ such that *first_code_l*[ℓ] $\le$ *buffer* $<$ *first_code_l*[$\ell + 1$];
set *offset* $\leftarrow$ (*buffer* $-$ *first_code_l*[ℓ]) >> ($L - \ell$);
set $s \leftarrow$ *first_symbol*[ℓ] + *offset*;
set *buffer* $\leftarrow$ ((*buffer* << ℓ) & $mask_L$) + *getbits*(ℓ);
output s;

where >> is a right-shift operator; << is a left-shift operator; & is a bitwise logical "and" operator; $mask_L$ is the bitstring $(1 << L) - 1$ containing L bits, all "1"; and where *getbits*(ℓ) extracts the next ℓ bits from the input stream and returns them as an ℓ-bit integer. For example, if the six bits in *buffer* are `110010`, or integer 50, then the code fragment first identifies $\ell = 4$, since $48 =$ *first_code_l*[4] $\le 50 <$ *first_code_l*[5] $= 52$; then computes *offset* as $(50-48) >> 2 = 0$; sets s to be *first_symbol*[4] $+ 0 = 2$; and finally shifts *buffer* left by 4 bits, zeros all but the final 6 bits, and adds in four new bits from the compressed bit-stream.

Table 4. Partial Decode Tables $search\_start_t[]$ for a t-bit Prefix of the Buffer *buffer*, for Two Different Values of t

v	$search\_start_2[v]$	$search\_start_3[v]$
0	1	1
1	1	1
2	2	1
3	4*	1
4	—	2
5	—	2
6	—	4*
7	—	5*

Asterisks indicate entries that may require loop iterations following the initial assignment.

The first step of the process—"identify ℓ such that"—is the most costly one. As with encoding, a linear or binary search over the range $\ell_{\min} \ldots L$ can be used, both of which take $O(\ell)$ time, where ℓ is the number of bits being processed. That is, even if linear search is used, the cost of decoding is proportional to the size of the compressed file being decoded.

If more memory can be allocated, other "identify ℓ" options are available, including direct table lookup. In particular, if 2^L bytes of memory can be allowed, an array indexed by *buffer* can be used to store the length of the first codeword contained in *buffer*. Looking again at the code shown in Tables 1 and 3, such an array would require a 64-element table, of which the first 32 entries would be 1, the next 16 would be 2, the next 4 would be 4, and so on. One byte per entry is sufficient in this table, because the codewords can be assumed to be limited to 255 bits. Even so, 2^L could be much larger than n, and the table might be expensive. Moreover, in a big table the cost of cache misses alone could mean that a linear or binary search in $first\_code\_l[]$ might be preferable in practice.

Moffat and Turpin [61] noted that the search and table lookup techniques can be blended, and demonstrated that a partial table that accelerated the linear search process was an effective hybrid. In this proposal, a $search\_start[]$ table of 2^t entries is formed for some $\ell_{\min} \le t \le L$. The first t bits of *buffer* are used to index this table, with the stored values indicating either the correct length ℓ, if t bits are sufficient to unambiguously determine it; or the smallest valid value of ℓ, if not. Either way, a linear search commencing from the indicated value is used to confirm the correct value of ℓ relative to the full contents of *buffer*.

Table 4 continues the example shown in Tables 1 and 3 and shows partial decoding tables $search\_start_t[]$ for $t = 2$ and $t = 3$. The first few entries in each of the two tables indicate definite codeword lengths; the later entries are lower bounds and are the starting point for the linear search, indicated by the "*" annotations. With a t-bit partial decode table $search\_start[]$ available, and again presuming a suitable sentinel value in $first\_code\_l[L+1]$, the first "identify ℓ" step then becomes:

set $\ell \leftarrow search\_start[buffer >> (L - t)]$;
while $buffer \ge first\_code\_l[\ell + 1]$ **do**
    set $\ell \leftarrow \ell + 1$.

Because the short codewords are by definition the frequently used ones, a surprisingly high fraction of the coded symbols can be handled "exactly" using a truncated lookup table. In the example, even when a $t = 2$ table of just four values is used, $(20 + 17 + 6)/55 = 78\%$ of the symbol occurrences get their correct length assigned via the table, and the average number of times the

loop guard is tested when $t = 2$ is just 1.25. That is, even a quite small truncated decoding table can allow canonical Huffman codes to be decoded using near-constant time per output symbol. Note also that neither *putbits*() nor *getbits*() should be implemented using loops over bits. Both operations can be achieved through the use of constant-time masks and shifts guarded by if-statements, with the underlying writing and reading (respectively) steps based on 32- or 64-bit integers.

If relatively large tracts of memory space can be employed, or if L and n can both be restricted to relatively small values, decoding can be completely table-based, an observation made by a number of authors [2, 10, 11, 12, 13, 28, 29, 34, 37, 50, 64, 71, 75, 77, 83]. The idea common to all of these approaches is that each of the $n - 1$ internal nodes of an explicit code tree can be regarded as being a *state* in a decoding *automaton*, with each such state corresponding to a recent history of unresolved bits. For example, the right-most internal node in the right-hand tree in Figure 4 (labeled with a weight of "7") represents the condition in which "11" has been observed, but not yet "consumed." If the next k bits from the input stream are then processed as a single entity—where $k = 8$ might be a convenient choice, for example—they drive the automaton to a new state and might also give rise to the output of source symbols. Starting at that internal node labeled "7" in the right-hand tree in Figure 4, the $k = 8$ input block "`01010101`" would lead to the output of four symbols, "e, d, b, b" (that is, original symbols "4,3,1,1") and would leave the automaton at the internal node labeled with a weight of "33," the root of the tree. Each other k-bit input block would give rise to a different set of transitions and/or outputs.

Across the $n - 1$ internal nodes, the complete set of $(n - 1)2^k$ transitions can be computed in advance and stored in a two-dimensional array with (at most) k symbols to be emitted as each k-bit input block is processed. Decoding is then simply a matter of starting in the state corresponding to the code tree root and then repeatedly taking k-bit units from the input, accessing the table, writing the corresponding list of source symbols (possibly none) associated with the transition, and then shifting to the next state indicated in the table.

Each element in the decoding table consists of a destination state, a count of output symbols (an integer between 0 and k), and a list of up to k output symbols; and the table requires $2^k(n - 1)$ such elements. Hence, even quite moderate values of n and k require non-trivial amounts of memory. For example, $n = 2^{12}$ and $k = 8$ give rise to a table containing $(8 + 2) \times 256 \times (2^{12} - 1)$ values, and if each is stored as (say) a 16-bit integer, will consume in total around 20 MiB. While not a huge amount of memory, it is certainly enough that cache misses might have a marked impact on actual execution throughput. To reduce the memory cost, yet still capture some of the benefits of working with k-bit units, hybrid methods that blend the "search for ℓ" technique described earlier in this section with the automaton-based approach are also possible [44].

3.3 Housekeeping

Unless the minimum-redundancy code that will be employed in some application is developed in advance and then compiled into the encoder and decoder programs (as was the case, for example, for some facsimile coding standards in the early 1980s), a *prelude* component must also be associated with each message transmitted, describing the details of the code that will be used for the body of the message. Elements required in the prelude include the length m of the message, so the decoder knows when to stop decoding; a description of the size n and composition of the source alphabet, if it cannot be assumed by default or is only a subset of some larger universe; and the length ℓ_i of the codeword associated with each of the n source symbols that appears in the message.

The two scalars, m and n, have little cost. But if the source alphabet is a subset of a larger universe, then a *subalphabet selection vector* is required and might be a rather larger overhead. For example, if the source universe is regarded as being the set of all 32-bit integers, then any particular

message will contain a much smaller number of distinct symbols with, typically, $n < m \ll 2^{32}$. The simplest selection approach is to provide a list of n four-byte integers, listing the symbols that appear in the current message. But it is also possible to regard the subalphabet as being defined by a bitvector of length 2^{32}, where a "1" bit at position u indicates that symbol u is part of the subalphabet and has a code assigned to it. Standard representations for sparse bitvectors (including for cases where it is dense in localized zones, which is also typical) can then be used, reducing the storage cost. Moffat and Turpin [63, Chapter 3] describe several suitable methods.

The last component of the prelude is a set of n codeword lengths, $T = \langle \ell_i \rangle$. These are more economical to transmit than the weights $W = \langle w_i \rangle$ from which they are derived, and also more economical than the actual codewords that will be assigned, since the codeword lengths are integers over a relatively compact range, $\ell_{\min} \ldots L$. The codeword lengths should be provided in subalphabet order as a sequence of integer values, perhaps using $\lceil \log_2(L - \ell_{\min} + 1) \rceil$ bits each, or perhaps using a secondary minimum-redundancy code in which $n' = L - \ell_{\min} + 1$.

Once the decoder knows the subalphabet and the length of each of the codewords, it generates the canonical symbol ordering, sorting by non-decreasing codeword length, and breaking ties by symbol identifier, so its canonical code assignment matches the one that gets formed by the encoder. That is, the symbol weights must be ignored during the canonical reordering, since they are never known to the decoder. Encoding (and later on, when required, decoding) using the techniques described earlier in this section can then be commenced.

If the message is very long or is of unknown length, it can be broken into large fixed-length blocks and a prelude constructed for each block. The cost of multiple preludes must then be accepted, but the use of locally fitted codes and possibly different subalphabets within each block means that overall compression might actually be *better* than if a single global whole-of-message code was developed and a single prelude constructed.

Turpin and Moffat [78] provide a detailed description of preludes and the processes employed in encoder and decoder that allow them to remain synchronized.

4 SPECIALIZED MINIMUM-REDUNDANCY CODES

We now consider variants of the core minimum-redundancy coding problem, in which additional constraints and operating modes are introduced.

4.1 Length-limited Codes

Suppose that an upper limit L is provided and a code must be constructed for which $\ell_{n-1} \le L < L_{\text{Huff}}$, where L_{Huff} is the length of the longest codeword in a Huffman code. Moreover, the code should have the least possible cost, subject to that added constraint. This is the *length-limited coding* problem. Note that, as in the previous section, L is the length of a longest codeword in the code that is being constructed and deployed, and that L_{Huff} is a potentially larger value that is not necessarily known nor computed.

Hu and Tan [32] and Van Voorhis [80] provided early algorithms that solved this problem; the one we focus on here is the more efficient *package-merge* paradigm proposed in 1990 by Larmore and Hirschberg [41]. The key idea is that of building *packages* of symbols in much the same way as Huffman's algorithm does, but taking care that no item can take part in more than L combining steps. That restriction is enforced by creating all least-cost subtrees of different maximum depths and retaining information for each of the possible node depths in a separate structure. Algorithm 3 provides pseudo-code for this approach.

The first list of packages—referred to as *packages*[1] in Algorithm 3—is taken directly from the set of original symbols and their corresponding weights, $W[]$. These are the only subtrees that are

ALGORITHM 3: Package-Merge process for length-limited codes [41].

```
0: function CalcCodeLens(W, n, L)
1:     // Create a least-cost code for the weights W[0 ... n − 1] in which no codeword
2:     //       is longer than L
3:     set packages[1] ← W
4:     for level ← 2 to L do
5:         set packages[level] ← the empty set
6:         form all pairs of elements from packages[level − 1],
7:               taking them in increasing weight order
8:         join each such pair to make a new subtree in packages[level],
9:               with weight given by the sum of the two component weights
10:        merge another copy of W into the set packages[level]
11:    set solution[L] ← the smallest 2n − 2 items in packages[L]
12:    for level ← L − 1 downto 1 do
13:        set count ← the number of multi-item packages among the items in solution[level + 1]
14:        set solution[level] ← the smallest 2 · count items in packages[level]
15:    set ⟨ℓ_i⟩ ← ⟨0, 0, ..., 0⟩
16:    for level ← L downto 1 do
17:        for each leaf node s in solution[level] do
18:            set ℓ_s ← ℓ_s + 1
19:    return ⟨ℓ_i⟩
```

possible if their depth is limited to one. Another way of interpreting this list is that it represents all of the different ways in which a reduction in the Kraft sum $\mathcal{K}(\cdot)$ of 2^{-L} can be achieved, by "demoting" a subtree (node) from having its root at depth $L-1$ to having its root at depth L.

A list of subtrees of depth up to two is then generated, representing ways in which the Kraft sum might be decreased by 2^{-L+1}, corresponding to moving a subtree or element from depth $L-2$ to depth $L-1$. To do this, the items in *packages*[1] are formed into pairs, starting with the two smallest ones and working up to the largest ones. That process generates $\lfloor n/2 \rfloor$ packages, each with a weight computed as the sum of the two item weights representing a possible internal node in a code tree. That list of packages is extended by merging it with another copy of $W[]$, generating the full list *packages*[2] of $n + \lfloor n/2 \rfloor$ subtrees of depths up to two, again ordered by cost.

Figure 6 illustrates the computation involved. Items in brown are drawn from $W[]$, and all $n = 10$ of them appear in every one of the *packages*[] rows. Interspersed are the composite nodes, marked in blue, each constructed by pairing two elements from the row above. In the diagram, the pairs are indicated by underbars, and some (not all) are also traced by arrows to show their locations in the next row. Because an $L = 5$ code is being sought, the fifth row marks the end of the packaging process—none of the subtrees represented in that row can have a depth that is greater than five. That fifth row is a collection of elements and subtrees that when demoted from a nominal level of zero in the tree—that is, occurring at the level of the root of the desired code tree—to a nominal level of one, each decrease the Kraft sum by $2^{-L+(L-1)} = 0.5$.

If every element (leaf or subtree) is assumed to be initially at level zero, then the Kraft sum $\mathcal{K}(\cdot)$ has the value $\sum_{i=0}^{n-1} 2^{-0} = n$ and exceeds the limit of 1 that indicates a feasible code. Indeed, to arrive at a feasible code, the Kraft sum must be reduced by $n-1$ compared to this nominal starting situation. Therefore, selecting the least-weight $2n-2$ of the items in *packages*[L] forms a set of items—called *solution*[L] in Algorithm 3—that when each is demoted by one level, leads to the required code. But to achieve those demotions, each of the composite items in *solution*[L] need to have their demotions propagated to the next level down to form the set *solution*[$L-1$]. In turn,

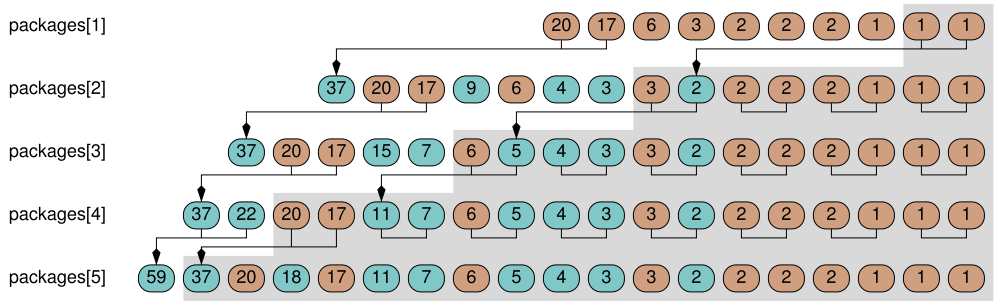

Fig. 6. Package-merge algorithm applied to the weights $W = \langle 20, 17, 6, 3, 2, 2, 2, 1, 1, 1\rangle$ with $L = 5$. Packages are shown in blue and leaf nodes in brown. The shaded region shows the five *solution*[] sets, starting with the least-cost $2n - 2$ items in the fifth row and then enclosing the necessary packets to construct it in the rows above, each as a subset of the corresponding *packages*[] set. The final code is $T = \langle 2, 2, 3, 4, 4, 4, 4, 4, 5, 5\rangle$ with cost 142, two greater than the minimum-redundancy code $\langle 1, 2, 4, 5, 5, 5, 5, 6, 6\rangle$. If the final $2n - 2$ elements in the fourth row are expanded, an $L = 4$ length-limited minimum-redundancy code $T = \langle 2, 2, 4, 4, 4, 4, 4, 4, 4, 4\rangle$ is identified, with cost 146. It is not possible to form an $L = 3$ code for $n = 10$ symbols.

the composite items in *solution*$[L - 1]$ drive further demotions in *solution*$[L - 2]$, continuing until *solution*[1] is determined, which, by construction, contains no packages. The process of identifying the L *solution*[] sets is described by the loop at steps 12 to 14 in Algorithm 3.

The shaded zone in Figure 6 shows the complete collection of *solution*[] sets, one per level, that collectively include all of the required leaf demotions. For example, symbol 0 with weight $w_0 = 20$ appears twice across the five *solution*[] sets, and hence needs to be demoted twice from its nominal $\ell_0 = 0$ starting point, meaning that it is assigned $\ell_0 = 2$. At the other end of the alphabet, symbol 9 with weight $w_9 = 1$ appears in all five *solution*[] sets and is assigned a codeword length of $\ell_9 = 5$. Steps 16 to 18 in Algorithm 3 check the L solution sets, counting the number of times each original symbol needs to be demoted, and in doing so, computing the required code $T = \langle \ell_i \rangle$. The complete code generated by this process is $T = \langle 2, 2, 3, 4, 4, 4, 4, 5, 5\rangle$ and has a cost of 142 bits, two more than the Huffman code's 140 bits.

The package-merge implementation described in Algorithm 3 and illustrated in Figure 6 corresponds to the description of Huffman's algorithm that is provided in Algorithm 1—it captures the overall paradigm that solves the problem, but leaves plenty of room for implementation details to be addressed. When implemented as described, it is clear that $O(nL)$ time and space might be required. In their original presentation, Larmore and Hirschberg [41] also provided a more complex version that reduced the space requirement to $O(n)$ but added (by a constant factor) to the execution time; and a range of other researchers have also described package-merge implementations. For example, Turpin and Moffat [76] noted that it is more efficient to compute the complement of the *solution*[], and present a *reverse package merge* that requires $O(n(L - \log_2 n + 1))$ time; Katajainen et al. [36] describe an implementation that executes in $O(nL)$ time and requires only $O(L^2)$ space; and Liddell and Moffat [45] describe an implementation that starts with a Huffman code and then optimally rearranges it without computing all the packages, executing in time $O(n(L_{\text{Huff}} - L + 1))$ time, where L_{Huff} is the length of the Huffman code for the same set of input weights.

Other authors have considered approximate solutions that limit the codeword lengths and in practice appear to give codes of cost close to or equal to the minimum cost [23, 27, 43, 52]. Milidiú and Laber [49] analyze the relative effectiveness of length-limited codes and show that for all but extremely constrained situations the compression loss relative to a Huffman code is small.

4.2 Dynamic Huffman Coding

Suppose that the alphabet that will be used for some message is known, but not the symbol frequencies. One clear option is to scan the message in question, determine the symbol frequencies, and then proceed to construct a code. In this *semi-static* approach, assumed through all of the preceding discussion, the prelude is what informs the decoder of the codewords in use, and decoding is carried out using the constant code that is so defined.

But there is also another option, and that is to make use of *adaptive* probability estimates. Instead of scanning the whole message and constructing a prelude, some default starting configuration is assumed—perhaps that every symbol in the alphabet has been seen "once" prior to the start of encoding, and hence is equally likely—and the first message symbol is coded relative to that distribution, with no prelude required. Then, once that symbol has been processed, both encoder and decoder adjust their symbol estimates to take into account that first symbol and, always staying in step, proceed to the second symbol, then the third, and so on. In this approach, just *after* the last symbol is coded and transmitted, both processes will know the final set of message occurrence counts and could potentially be used to construct the code that would have been used had an initial scan been used, and a prelude sent.

Several authors have contributed techniques that allow *dynamic Huffman coding* and demonstrated that it is possible to maintain an evolving code tree in time that is linear in the number of bits required to adaptively communicate the message, commencing in 1973 with work by Faller [20] and with enhancements and further developments added subsequently by Gallager [25], Cormack and Horspool [16], Knuth [39], Vitter [81, 82], Lu and Gough [46], Milidiú et al. [51], and Novoselsky and Kagan [65]. The common theme across these mechanisms is that a list of all leaf nodes and internal tree nodes and their weights is maintained in sorted non-increasing order, and each time the weight of a leaf is increased because of a further occurrence of it in the message, the list is updated (if required) by shifting that leaf to a new position and then considering what effect that change has on the pairing sequence that led to the Huffman tree. In particular, if a leaf swaps in the ordering with another node, they should exchange positions in the Huffman tree, altering the weights associated with their parents and potentially triggering further updates. When appropriate auxiliary linking information is maintained, incrementing the weight associated with a leaf currently at depth b in the tree (and hence, presumably just coded using a b-bit codeword) can be accomplished in $O(b)$ time; that is, in $O(1)$ time per tree level affected. Overall, these arrangements can thus be regarded as taking time linear in the number of inputs and outputs.

Dynamic Huffman coding approaches make use of linked data structures and cannot benefit from the canonical code arrangements described in Section 3. That means they tend to be slow in operation. In addition, they require several values to be stored for each of the n alphabet symbols, so space can also become an issue. In combination, these drawbacks mean that dynamic Huffman coding is of only limited practical use. There are no general-purpose compression systems that make use of dynamic Huffman coding, hence our relatively brief treatment of them here.

4.3 Adaptive Algorithms

An *adaptive* algorithm is one that is sensitive in some way to the particular level of difficulty represented by the given problem instance. For example, in the field of sorting, a wide range of adaptive algorithms are known that embed and seek to exploit some notion of "pre-existing order," such as the number of inversions; and in doing so provide faster execution when the input sequence has low complexity according to that secondary measure [66].

Adaptive algorithms for computing minimum-redundancy codes have also been described. These should not be confused with *adaptive coding*, another name for the dynamic coding problem

described in Section 4.2. In these adaptive algorithms, it is again usual for a second "quantity" to be used to specify the complexity of each problem instance, in addition to n, the instance size. As a first variant of this type of algorithm, Moffat and Turpin [62] introduce a concept they refer to as *runs*, noting that in many typical large-alphabet coding problems the observed symbol frequency distribution has a long tail, and that many symbols have the same low frequencies. The distribution $W = \langle 20, 17, 6, 3, 2, 2, 2, 1, 1, 1\rangle$ used as an example previously has a few repetitions; and in the run-length framework of Moffat and Turpin could equally well have been represented as $W = \langle 1(20), 1(17), 1(6), 1(3), 3(2), 3(1)\rangle$, where the notation "$r_j(w_j)$" means "$r_j$ repetitions of weight w_j." If the coding instance is presented as the usual sorted list of frequencies, then it requires $O(n)$ time to convert it into this run-based format, and there is little to be gained compared to the use of Algorithm 2. But if the coding instance is provided as input in this (typically) more compact format, then Huffman's algorithm can be modified so it operates on a data structure that similarly maintains runs of symbols, and the output $T = \langle 1(1), 1(2), 1(4), 5(5), 2(6)\rangle$ can be generated. If the input consists of n symbols with r different symbol weights, that code calculation process requires $O(r(1 + \log(n/r)))$ time, which is never worse than $O(n)$. A runlength-based implementation of the package-merge process that was described in Algorithm 3 is also possible [36].

Milidiú et al. [53] have also considered techniques for efficiently implementing Huffman's algorithm. They consider the maximum codeword length L generated by an application of Huffman's algorithm to be the secondary indicator of the complexity of a problem instance and describe methods that compute Huffman codes for sorted inputs in $O(n)$ time and $O(L)$ additional space. Note that the computational model employed by Milidiú et al. requires that the input weights W not be overwritten; in this more restrictive framework, Algorithm 2 requires n additional words of memory and is no longer "in place." Kärkkäinen and Tischler [35] have also considered the question of memory space required during code construction.

Belal and Elmasry [3, 4] have considered adaptivity using a similar framework and show that Huffman codes for unsorted sequences of weights can be computed in $O((16^{L_{\text{diff}}})n)$ time, where L_{diff} is the number of distinct codeword lengths, $L_{\text{diff}} \le L$. But note that $L_{\text{diff}} \ge 2$ for all interesting cases, and hence that $n > 2^{256}$ is required before $(16^{L_{\text{diff}}})n < n \log_2 n$. Belal and Elmasry also demonstrate that if the weights are presented in sorted order, then $O((9^{L_{\text{diff}}}) \log^{2k} n)$ time is sufficient, which is $o(n)$ under the same conditions—if L_{diff} is very small and n is astronomically large. Implementations of these complex algorithms have not yet been measured, and it seems likely that if they do get implemented they will be impractical for all plausible combinations of L_{diff} and n.

Barbay [1] has also developed an adaptive algorithm for computing Huffman codes. In his work, the secondary measure of instance complexity is the number of *alternations* induced by the sequence of input weights, where each alternation is a switch (in terms of Algorithms 1 and 2) from consumption of original leaf nodes to the consumption of internal nodes during the tree formation process. When the number of alternations in W, denoted $\alpha(W)$, is high, the resultant tree has many distinct levels and the instance cost is high; when $\alpha(W)$ is low, the tree is likely to be relatively shallow and easier to construct. Based on this approach, Barbay describes a mechanism for computing a Huffman code from a non-sorted input sequence W of n symbol weights in $O(n(1 + \log \alpha(W)))$ time; Barbay also demonstrates that when there are $\alpha(W)$ alternations, it is not possible to compute a minimum-redundancy code in less than this much time, and hence that relative to the alternations measure, his approach is optimally adaptive. Note also that $L_{\text{diff}} \le \alpha(W)$ for all sequences W, and that it may thus be possible to further refine Barbay's approach and develop an adaptive algorithm that requires only $O(n(1 + \log L_{\text{diff}}))$ time.

4.4 Coding with Infinite Alphabets

Finally, it is worth mentioning the work of Solomon Golomb [26], who developed minimum-redundancy codes for certain types of infinite probability distributions, notably the geometric distribution. Discussion of these approaches is outside the scope of this article.

5 OTHER ENTROPY-CODING TECHNIQUES

In this section, we provide an introduction to two other entropy coding techniques. Section 6 then compares them to the Huffman coding mechanism described in Sections 2 and 3. Both of these two techniques are able to obtain compression closer to the entropic bound (Equation (3)) than is Huffman coding, because of their use of *state* variables that carry information—fractional bits, so to speak—forward from one coded symbol to the next. Where these two methods differ is in terms of what their respective state variables represent and how channel symbols are extracted from the state variables and committed to the output stream in an incremental manner. In particular, in an arithmetic coder, a fractional state value becomes increasingly precise as symbols are encoded, with more and more leading bits being bound to their final values; whereas, in an ANS coder, an integer state value is allowed to grow larger and larger as the symbols are coded. The next two sections provide examples that illustrate these two methods and the differences between them.

5.1 Arithmetic Coding

The basic principles of arithmetic coding are described by a number of authors [5, 63, 67, 84], with the presentation here largely built on the description of multi-symbol alphabet adaptive coding first given in 1987 by Witten et al. [85] and later extended by Moffat et al. [57]. Prior to 1987, activity had largely focused on binary alphabets and binary arithmetic coding for messages over biased two-symbol arrangements (when $W = \langle w_0, w_1 \rangle$ with $w_0 \gg w_1$) using less than one bit per symbol.

The key idea that underpins all arithmetic coding implementations is that of a current coding *state*, described by a pair of values $\langle lower, range \rangle$, both of which (in simple terms) should be thought of as being arbitrary-precision values between 0 and 1. Together they constrain a *value* that represents the input message and which must also be in the range $[0, 1)$:

$$lower \le value < lower + range.$$

At the commencement of encoding, before any symbols have been processed, $lower = 0$ and $range = 1$, with *value* free to take on any number between zero and one.

As each message symbol s is processed, the interval corresponding to the current state is replaced by a narrower interval $[next\_lower, next\_lower + next\_range)$ that further constrains *value*,

$$lower \le next\_lower \le value < next\_lower + next\_range \le lower + range.$$

The narrowing is carried out in strict numeric proportion to the probability range within the interval $[0, 1)$ that is associated with symbol s, based on the relative symbol weights expressed in W. The first part of Algorithm 4 provides details of this process, making use of a pre-computed array $base[s]$ that stores the cumulative weight of all symbols prior to s in the source alphabet. The half-open interval $[base[s]/m, base[s + 1]/m)$ is the fraction assigned to s in the unit interval $[0, 1)$ (recall that m is the sum of the symbol frequency counts; that is, the length of the sequence being compressed), and that means that each interval narrowing step results in

$$next\_range = range \cdot (w_s/m).$$

Once all of the message symbols have been processed, a shortest-possible unambiguous binary *value* that lies fully within the final interval is identified and transmitted to the decoder. As already

ALGORITHM 4: Computing the state transitions required by arithmetic coding [57, 85].

```
1: function arith_encode(lower, range, s)
2:     // Compute the arithmetic coding state change, assuming that
3:     //     base[s] = Σ_{i=0}^{s-1} W[i] for 0 ≤ s ≤ n has been precomputed, with base[n] = m
4:     set scale ← range/m
5:     set next_lower ← lower + scale · base[s]
6:     set next_range ← scale · W[s]
7:     return ⟨next_lower, next_range⟩

8: function arith_decode(lower, range, value)
9:     // Compute symbol identifier by partitioning the range so that it straddles value
10:    set scale ← range/m
11:    set target ← ⌊(value − lower)/scale⌋
12:    find s such that base[s] ≤ target < base[s + 1]
13:    set next_lower ← lower + scale · base[s]
14:    set next_range ← scale · W[s]
15:    return ⟨next_lower, next_range, s⟩
```

noted, prior to any symbols being coded, $range = 1$. Hence, once the sequence of m symbols making up the message have all been processed, and assuming that exact arbitrary-precision arithmetic is available, the final interval $[lower, lower + range)$ completely captures the message and can be represented by any single number *value* that satisfies $lower < value < lower + range$. Such a value cannot require more than $(-\log_2 range) + 2$ bits to describe, meaning that the m symbols in the source message can be coded in at most $2 + \sum_s \log_2(m/w_s)$ bits; that is, in at most two more bits than the summed entropic cost of the symbols making up the message.

To decode the message embodied in *value*, the decoder uses *base*[] to determine a symbol identifier s whose probability assignment in the range $[0, 1)$ straddles (in proportion) the relationship that *value* has to the current interval $[lower, lower + range)$. The second part of Algorithm 4 describes this computation, with *lower* and *range* being initialized to 0 and 1 again, and then faithfully tracing the same sequence of values in the decoder as they did in the encoder, with the interval narrowing around the message *value*, which remains constant.

Table 5 gives an example of this exact form of arithmetic coding, rendering an 11-symbol message "0,0,1,0,2,0,1,5,0,3,1" into a 24-bit compressed message. Relative to the probabilities established by W, the entropic cost of this message would be 22.95 bits. The Huffman code depicted in Figure 2 also requires 23 bits. In this short example, the ability of arithmetic coding to faithfully match the entropic costs does not show through particularly well. But in more extreme cases, the advantage becomes clear. For the weights $W = \langle 99, 1\rangle$, the 24-symbol message "0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" is coded as "`11110010`" and takes just 8 bits, a substantial reduction on the 24 that would be required by a minimum-redundancy code.

Practical implementations of arithmetic coding avoid arbitrary-precision arithmetic and instead include a *renormalization* process that emits a bit (or byte—see Schindler [68]) and doubles *range* (or multiplies it by 256) whenever a leading bit (or byte) of *lower* can be unambiguously determined. That ability means that *lower* and *range* can be manipulated as 32-bit integers [57, 85], and hence that coding one symbol involves a small number of integer multiplication and integer division operations to compute *next_lower* and *next_range* from *lower*, *range*, *base*[s], and w_s; possibly followed by one or more renormalization cycles. The fact that the calculations are limited to at most 32 bits of precision and are not exact means that some slight compression effectiveness loss emerges, but in practical systems it is very small.

Table 5. Encoding the Sequence 0,0,1,0,2,0,1,5,0,3,1 Relative to the Weights $W = \langle 10, 6, 2, 1, 1, 1\rangle$, Assuming an Alphabet of $n = 6$ Symbols

Symbol		Current code interval		
		lower	*range*	*lower + range*
initial		0.00000000000000000000000000	1.00000000000000000000000000	1.00000000000000000000000000
0	[0/21, 10/21)	0.00000000000000000000000000	0.47619047619047616404230890	0.47619047619047616404230890
0	[0/21, 10/21)	0.00000000000000000000000000	0.22675736961451245510268390	0.22675736961451245510268390
1	[10/21, 16/21)	0.10797969981643451564679030	0.06478781988986069551028630	0.17276751970629522503486440
0	[0/21, 10/21)	0.10797969981643451564679030	0.03085134280469556747328320	0.13883104262113007965062650
2	[16/21, 18/21)	0.13148548481048827341055580	0.00293822312425672058783240	0.13442370793474500700881440
0	[0/21, 10/21)	0.13148548481048827341055580	0.00139915386869367643941920	0.13288463867918196004147550
1	[10/21, 16/21)	0.13215174855748526572263300	0.00039975824819819322968400	0.13255150680568345000764910
5	[20/21, 21/21)	0.13253247069862639229143040	0.00001903610705705681836370	0.13255150680568345000764910
0	[0/21, 10/21)	0.13253247069862639229143040	0.00000906481288431277089140	0.13254153551151071077640610
3	[18/21, 19/21)	0.13254024053824151052616290	0.00000043165775639584618500	0.13254067219599791060957730
1	[10/21, 16/21)	0.13254044608955409256800810	0.00000012333078754167031970	0.13254056942034164734423030
final values (binary)				
lower		0.00100001 11101110 00101011 10110001 01011010 · · ·		
lower + range		0.00100001 11101110 00101101 11000011 00001101 · · ·		

The encoded message can be represented in decimal by the number *value* = 0.13254050, or in binary by the 24-bit string (with the leading zero and binary point suppressed) "`00100001 11101110 00101100`."

Compared to Huffman coding, arithmetic coding offers two significant advantages: it handles skewed probability distributions accurately; and it readily supports adaptive models, in which the probability estimates are adjusted as the message is processed and in which there might be multiple contexts, with the choice as to which one gets used for any particular symbol determined by the sequence of previous symbols. These are both enormously useful attributes and have been used to good advantage in multi-state multi-context compression schemes such as prediction by partial matching (PPM) compression systems [14, 22, 31, 54, 72]. However, the multiplications and divisions required mean that arithmetic coding decodes more slowly than does canonical minimum-redundancy coding, even when the symbol frequencies are fixed and the compression system is static or semi-static. A wide range of methods have also been introduced that make use of approximate arithmetic in some way, where decreased compression effectiveness is accepted in return for faster execution, but these still do not offer the decoding speed that is possible with canonical Huffman decoding.

When carrying out arithmetic encoding based on fixed symbol probabilities, the required calculations are all based on values that can be pre-computed and stored in n-element arrays. But when decoding, even with static probabilities, the set of cumulative weights *base*[] must be searched to identify the symbol s for which $base[s] \le target < base[s+1]$. One option is to spend $O(\log n)$ time per symbol to binary-search the array *base*[]; another is to allocate a further array of m elements, *symbol*[], to map directly from the set of possible *target* numbers to the corresponding symbol identifiers. It is also possible to carry out the search step using n-element arrays in $O(1 + \log b)$ time, where b is the number of bits associated with the code for the symbol in question [55]. Moffat and Turpin [63, Chapter 5] describe these structures and a range of other details required in a full arithmetic coding implementation.

Semi-static arithmetic coding requires that the set of n symbol weights $W = \langle w_i\rangle$ be communicated to the decoder via a prelude, a more onerous requirement than the codeword lengths

Table 6. Nominal ANS Transition Table $A[\cdot,\cdot]$ for Weights $W = \langle 10, 6, 2, 1, 1, 1\rangle$

Symbol		Mapping from *state* → *next_state*															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	···
0	10/21	1	2	3	4	5	6	7	8	9	10	**22**	**23**	**24**	**25**	**26**	**27**	···
1	6/21	11	12	13	14	15	16	**32**	**33**	**34**	**35**	**36**	**37**	53	54	55	56	···
2	2/21	17	18	**38**	**39**	59	60	80	81	···								
3	1/21	19	**40**	61	82	···												
4	1/21	20	**41**	62	83	···												
5	1/21	21	**42**	63	84	···												

The second cycle in the state transition sequence is picked out in blue/bold to illustrate the numbering regime that is followed, covering the transitions into states 22 to 42 inclusive. The vertical lines in each row denote the boundaries between the cycles.

associated with Huffman coding, because of the increased precision in the numbers involved. That increased precision is what allows more precise probability estimates to be employed and is what takes arithmetic coding closer to the entropic information bound $\sum \log_2(m/w_s)$, usually, but not always, recouping the additional prelude cost.

5.2 Asymmetric Numeral Systems

The past 10 years have seen the emergence of a third mechanism for entropy coding, the *asymmetric numeral systems* (ANS) technique originally developed by Jarek Duda [17, 18] and explored in practical terms by Yann Collet.[3] Like arithmetic coding, the message is treated as a whole, and a single "number" emerges at the end of the input and is regarded as representing the entire input message; and, also like arithmetic coding, a *state* is also maintained, in the case of ANS coding, as a single integer. The initial value of *state* is zero, representing the empty string ϵ. From that starting point, each input symbol s is used to shift from the current state to a new state via an update $next\_state \leftarrow A[state, s]$, where $A[\cdot,\cdot]$ is a pre-computed transition table.

Table 6 shows part of the table $A[\cdot,\cdot]$ for the weights used in several of the previous examples, $W = \langle 10, 6, 2, 1, 1, 1\rangle$, with $n = 6$. To create the table, the integers from 1 to m are allocated across the rows, distributed so that the i th row contains w_i of them. The integers from $m + 1$ to $2m$ are then likewise allocated in a second cycle; then the integers from $2m + 1$ to $3m$; and so on. The second cycle of values (incomplete for symbol 0) is picked out in blue in Table 6, with the vertical bars indicating the end of each cycle.

Now consider the sample message "0,0,1,0,2,0,1,5,0,3,1." The final value of *state* will be

$$A[A[A[A[A[A[A[A[A[A[0, 0], 0], 1], 0], 2], 0], 1], 5], 0], 3], 1],$$

of which the first four steps can be traced in Table 6 and with the remainder easily calculated (because of the regular nature of the table—see Algorithm 5). The total sequence of states associated with this example is

$$0 \rightarrow 1 \rightarrow 2 \rightarrow 13 \rightarrow 25 \rightarrow 270 \rightarrow 568 \rightarrow 1989 \rightarrow 41{,}790 \rightarrow 87{,}760 \rightarrow 1{,}842{,}979 \rightarrow 6{,}450{,}435,$$

and the final value—which is a 23-bit binary integer—completely encapsulates the 11 symbols in the input message. As already noted, the minimum-redundancy codes shown in Figure 2 would also require 23 bits for this sample message.

Decoding works in reverse: the final value for *state* is located in the table $A[\cdot,\cdot]$, with the row number indicating the symbol to be stacked (symbols are determined in reverse order

[3]For example, https://github.com/Cyan4973/FiniteStateEntropy.

ALGORITHM 5: Computing the mappings $A[\cdot,\cdot]$ and $A^{-1}[\cdot]$ for range ANS coding [18].

```
1: function ans_encode(state, s)
2:     // Compute the ANS forward mapping, assuming that
3:     //     base[s] = 1 + Σ_{i=0}^{s-1} W[i] for 0 ≤ s ≤ n has been precomputed, with base[n] = m + 1
4:     set f ← state div W[s]
5:     set r ← state mod W[s]
6:     set next_state ← f · m + base[s] + r
7:     return next_state

8: function ans_decode(state)
9:     // Compute the ANS inverse mapping, again assuming base[s]
10:    set r ← 1 + (state − 1) mod m
11:    set f ← (state − r) div m
12:    set s ← symbol[r]
13:    set prev_state ← f · W[s] − base[s] + r
14:    return ⟨prev_state, s⟩
```

during decoding) and the column number indicating the *prev_state* to revert to. For example, the integer 6,450,434 (one less than the example given above) gives rise to the output sequence "1,0,2,0,2,1,0,0,0,2,1"; and the integer 6,450,436 (one larger) leads to the decoded sequence "0,0,1,0,2,0,1,5,0,4,1."

What is quite remarkable in this technique is that—despite the seeming unpredictability of the values involved—it provides a deterministic one-to-one mapping from strings to integers in which the final *state* value associated with each string is very close to being the reciprocal of the products of the probabilities of the symbols comprising the string. This happens because the sequence of *state* values at each step increases by the required ratio. For example, in the sequence of states shown above as being associated with the string "0,0,1,0,2,0,1,5,0,3,1," the final three ratios are 87760/41790 = 2.100 = 21/10 (symbol 0); 1842979/87760 = 21.000 = 21/1 (symbol 3); and 6450435/1842979 = 3.500 = 21/6 (symbol 1). Hence, it is unsurprising that $\lceil \log_2 state \rceil$ is close to the entropy cost.

Like arithmetic coding, ANS can obtain compression effectiveness of less than one bit per symbol. For $W = \langle 99, 1\rangle$ and the message "0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0" that was used as an earlier example, the range ANS approach described in Algorithm 5 generates a final *state* of 730, and hence can represent the 24 symbols in 10 bits.

Use of an infinite table is, of course, impossible. But the highly regular nature of the cycles means that simple computations suffice to calculate the encoding and decoding transformation, given only an n-element array of integral symbol weights $W[]$ and a pre-computed cumulative sum array *base*[] of the same size. Algorithm 5 provides details, with the encoder mapping returning a *next_state*; and the inverse decoder mapping returning a tuple containing a *prev_state* and the corresponding decoded symbol s.

In the decoder, one additional array is assumed, the m-element inverse of *base*[]. In this array, $symbol[r]$ is the source symbol associated with the r th offset in each of the ANS cycles. In the example, the first 10 elements of *symbol*[], from subscript 1 to subscript 10, would indicate "0"; then the next 6 elements would indicate "1," and so on; with *symbol*[21], at the end of the cycle, storing "5." If space is at a premium, the *symbol*[] array can be replaced by a linear or binary search in *base*[], as was already discussed in connection with arithmetic coding.

A very important factor of ANS that allows fast decoding is the ability to adjust m by scaling the counts $W = \langle w_i \rangle$ to new values $W' = \langle w'_i \rangle$ so their sum m' is a power of two. If that is done,

then the inverse mapping function's key computations at steps 10 and 11 can be implemented as shift/mask operations. Slight compression effectiveness loss may be introduced as a result of the adjusted weights, but the overhead is likely to be very small, provided m', the new value, is several multiples larger than n, the size of the source alphabet. For example, a character-based coder for the $n = 256$ byte values might be designed so the observed character frequencies are scaled to yield $m' = 4{,}096$ or $m' = 65{,}536$. The latter will allow more precise probability estimates but also involve a larger *symbol*[] array, potentially slowing decoding throughput because of cache misses. In the special case in which m' is an integer power of two and all of the $W[i]$ values that sum to m' are also rounded to integer powers of two, then each coding operation adds an integral number of bits of precision to the value of *state*. That is, a Huffman code can be thought of as being a special case of an ANS code.

Omitted from the description in Algorithm 5 is the mechanism used to periodically renormalize *state*, necessary in an implementation so integer overflow does not arise. Renormalization occurs in the encoder by removing some number k of bits from the low-order end of *state* and writing them to the output stream and shifting the remaining bits to the right, where $k = 8$ or $k = 16$ might be attractive units. In the decoder, renormalization shifts *state* to the left by k bits and brings in the next k bits from the coded message. Careful attention to detail is required to ensure that encoder and decoder remain in step and that the sequence of states traversed backwards by the decoder is exactly faithful to the sequence traversed forwards by the encoder. As part of this mechanism, *state* is maintained within a range determined by m, the sum of the occurrence counts, and by $r = 2^k$, the radix of the code. One way of doing this is to require that (after an initial ramp-up stage during which the first of the two inequalities is not enforced) $C \cdot m \le state < C \cdot m \cdot r$ after every encoding step, and prior to every decoding step, for some integer multiplier $C \ge 1$.

Bounding the range of *state* in the encoder is achieved by anticipating what will happen at each upcoming encoding step and if the putative *next_state* is greater than the upper bound on the range, reducing *state* ahead of the encoding operation. That is, prior to step 6 in Algorithm 5, intervention may be necessary to ensure that the upper and lower bounds on *state* are complied with when that step does get performed. In the decoder, the corresponding update takes place following a decoding operation at step 13, and is indicated by the computed *prev_state* falling below the lower bound on the range (except during the wind-down phase at the end of the message as the first few symbols to have been encoded are regenerated).

The constant C provides a tradeoff between the fidelity of the arithmetic to the entropic cost and the number of bits used in the computation, and hence the size of the arrays used in a table-based implementation. When $C = 1$, compression effectiveness is compromised by round-off issues as *state* gets scaled, but is likely to still be better than a Huffman code; compression close to the entropic cost occurs when C is larger.

There is a clear difference between ANS renormalization and arithmetic coding renormalization—in the case of arithmetic coding, the encoder renormalization process takes bits (or bytes) from the most-significant end of the state variables *lower* and *range*, whereas in ANS they are taken from the least-significant end of *state*. Nevertheless, there are also many aspects of the renormalization process that are shared between the two methods.

Because the decoder of necessity must consume code digits in reverse order to their generation by the encoder and regenerates the output string from right to left, it is usual in an ANS coder for the input sequence to be reversed at the encoder, then the encoding to be performed with the output digits stored into a buffer, and finally the buffer to be written as the coded message, also in reverse order. Applying both of the "reversing" steps within the encoder allows the decoder to consume the code digits (starting with the encoder's final state) sequentially, at the same time writing

alphabet symbols directly to the output to regenerate the original input, thereby minimizing the cost of the decoding process.

Like arithmetic coding, ANS achieves its best compression if it has precise symbol counts available, in which case the prelude cost for ANS is the same as the prelude cost for an arithmetic coder, and greater than the prelude cost for a Huffman coder. If "approximated" power-of-two frequency counts as represented by a code $T = \langle \ell_i \rangle$ are used with an ANS coder, then compression equal to that of a Huffman coder will result.

6 CONCLUSION: IS HUFFMAN CODING DEAD?

In an article written not long after multi-symbol arithmetic coding was popularized, Bookstein and Klein [8] asked the question, 'Is Huffman coding dead?" Their investigation was prompted by the claims being made for arithmetic coding: that it provided compression close to the entropic cost; that it was easy to implement it adaptively; and that it was possible to not just use it adaptively with changing symbol probabilities, but also possible to use it with multiple probability distributions, so each symbol was coded in a context established by the symbols that preceded it in the input. Bookstein and Klein considered a number of factors, including compression effectiveness on typical zero-order character-based alphabets, applicability to short messages, operational performance in the face of inaccurate symbol probability estimates, robustness in the face of channel errors that corrupted message bits, and encoding and decoding speed. Their conclusion was that:

> *… for a substantial portion of compression applications, Huffman coding, because of its speed, simplicity, and effectiveness, is likely to be the preferred choice … for adaptive coding, or when dealing with highly skewed alphabets that cannot be redefined, arithmetic coding may well be the better of the two.*

Moffat et al. [60] also compare Huffman and arithmetic coding using a range of criteria and comment positively on the decoding speed of static Huffman coding and negatively on the speed of dynamic Huffman techniques.

Table 7 considers the same question, 25 years on from the evaluation of Bookstein and Klein, and now with ANS coding added. The "speed" values provided are deliberately qualitative, but in broad terms for a pure coding application (and with no modeling stage required; for example, taking an input file of 32-bit binary integers and representing it as an output file of coded bytes), "very fast" can be interpreted to mean something in the vicinity of 1–5 nanoseconds per decoded integer when executed on commodity hardware; "moderately fast" as meaning 10–50 nanoseconds per decoded integer; and "slow" as meaning 100 or more nanoseconds per integer.

Considering the table's rows in turn, the big weakness of Huffman coding is its inability to approach the entropic cost when the probability distribution is dominated by the most common symbol. In terms of speed, arithmetic coding requires more computation than either Huffman or ANS coding, and if implemented adaptively (one of its key application areas), speed further drops, because of the updates that take place to the statistics data structure. The question of ANS decoding speed relative to Huffman decoding speed is one that has received detailed exploration by practitioners; see, for example, the results collected by Yann Collet,[4] who suggests that Huffman decoding is still faster than ANS decoding. Indeed, since Huffman coding can be regarded as being a special case of ANS coding, any implementation technique or performance gain made in regard to ANS will likely be transferable to Huffman coding. Moreover, the techniques described in Section 3.2 mean that minimum-redundancy coding can be very fast indeed, particularly if a

[4] https://github.com/Cyan4973/FiniteStateEntropy.

Table 7. Comparison of Huffman, Arithmetic, and ANS Entropy Coding Techniques.

Attribute	Minimum-redundancy	Arithmetic	ANS
Relative effectiveness	Close to entropy limit when $w_0/m \to 0$; arbitrarily bad as $w_0/m \to 1$.	Close to entropy limit for all inputs.	Close to entropy limit for all inputs.
Static encoding speed	Very fast.	Moderately fast.	Very fast.
Static decoding speed	Very fast.	Moderately fast.	Very fast.
Prelude cost for semi-static coding	Small.	Slightly higher.	Same as for arithmetic.
Adaptive encoding speed	Slow, and with significant space overheads.	Only marginally changed from static encoding.	n/a
Adaptive decoding speed	Slow, and with significant space overheads.	Only marginally changed from static decoding.	n/a
Multi-context operation	Likely to have inferior relative effectiveness because of skewed and/or small-alphabet coding situations.	Straightforward, little overhead.	n/a

length limit is applied to control the size of the decoding tables. Huffman coding is used in the well-known ZLib library, for example[5]; and in the bzip2 general purpose compressor.[6]

However, arithmetic decoding is not as fast. In particular, in the decoder (Algorithm 4), two state variables are maintained, and only one of the two divisions (the one by m) is "controllable," in the sense of it being possible in a semi-static coder to adjust the symbol frequencies so m is a fixed power of two. The second division cannot be dealt with in this way. In the ANS decoder (Algorithm 5), only a single state variable is maintained, and the division that is required at each decoding step is controllable and can always be implemented using a single shift/mask.

The ANS approach cannot be used where adaptive probability estimates and multi-context models are in operation, because the symbols are regenerated by the decoder in the reverse order that they are processed by the encoder. The high cost of dynamic Huffman coding thus means that complex multi-context models must still be coupled with arithmetic coding.

Table-based implementations of ANS that are similar to the table-based Huffman approaches result in further speed gains. Because of its near-entropy compression effectiveness and its fast decoding speed, ANS coding has been incorporated in a range of general-purpose compression tools in which block-based operation means that static codes are appropriate and that the two reversing steps can be accommodated. These include the ZStd library[7] and software by Apple and Google. Interest in ANS has been in part sparked by several blogs including those of Charles Bloom,[8] Yann Collet,[9] and Franz Giesen.[10] Applications that have incorporated ANS include image

[5] http://zlib.net.
[6] http://www.bzip.org/.
[7] https://github.com/facebook/zstd.
[8] For example, http://cbloomrants.blogspot.com/2014/02/02-01-14-understanding-ans-3.html.
[9] For example, http://fastcompression.blogspot.com/2014/01/fse-decoding-how-it-works.html.
[10] For example, https://fgiesen.wordpress.com/2014/02/02/rans-notes/; and software at https://github.com/Cyan4973/FiniteStateEntropy.

compression [19]; compressed indexes for document retrieval systems [58, 59]; and compression of time-series data [7].

To obtain their compression advantage, arithmetic and ANS coders require more detailed prelude statistics, which adds a small overhead and might discourage their use for short messages. Note, however, that the implicit approximation of symbol occurrence counts that is an inevitable part of a minimum-redundancy coding prelude (Section 3.3) could also be explicitly employed with the arithmetic and ANS coding mechanisms, and so the "short messages" differential noted by Bookstein and Klein [8] is not a fundamental one in any way.

Taking all of the facets listed in Table 7 into account, it is clear that arithmetic coding remains the preferred choice for adaptive and multi-context modeling situations, but also that ANS is an important new technique that should be used in preference to minimum-redundancy (Huffman) coding in many of the latter's traditional application areas. However, it is also clear that when comparing ANS and minimum-redundancy coding, a trade-off between decoding speed and compression effectiveness still exists, and if the symbol probability distribution is not skewed, then canonical minimum-redundancy decoding continues to be the appropriate choice.

More than 60 years since it was first invented, Huffman's famous algorithm is no longer the irresistible force that it once was. But even so, Huffman coding remains alive and well, and plays an important role in practical data compression systems.

Software

An implementation of minimum-redundancy coding that includes many of the techniques described in this article and operates on general integer sequences is available at https://github.com/turpinandrew/shuff. Yann Collet's highly tuned implementation of table-based ANS (called Finite State Entropy coding, or FSE) is available at https://github.com/Cyan4973/FiniteStateEntropy and is accompanied by a corresponding carefully engineered implementation of Huffman coding, both operating over character-based input rather than general integer input. An implementation of arithmetic coding that is relatively untuned is available at http://people.eng.unimelb.edu.au/ammoffat/arith_coder; it also has a mode that processes general integer sequences.

ACKNOWLEDGMENTS

Multiple co-authors have contributed to the body of work that is summarized here, including (and not limited to) Andrew Turpin, Matthias Petri, Mike Liddell, Justin Zobel, and Jyrki Katajainen. Jérémy Barbay provided helpful input in connection with Section 4.3; and the referees provided detailed comments that have led to a range of useful improvements in the presentation.

REFERENCES

[1] J. Barbay. 2016. Optimal prefix free codes with partial sorting. In *Proceedings of the Symposium on Combinatorial Pattern Matching*. LIPIcs Series, Leibniz-Zentrum fuer Informatik, 29:1–29:13.

[2] M. A. Bassiouni and A. Mukherjee. 1995. Efficient decoding of compressed data. *J. Amer. Soc. Inform. Sci.* 46, 1 (1995), 1–8.

[3] A. A. Belal and A. Elmasry. 2006. Distribution-sensitive construction of minimum-redundancy prefix codes. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Vol. 3884. Lecture Notes in Computer Science, 92–103.

[4] A. A. Belal and A. Elmasry. 2016. Optimal prefix codes with fewer distinct codeword lengths are faster to construct. Retrieved from: *CoRR* abs/cs/0509015 (2016), 23.

[5] T. C. Bell, J. G. Cleary, and I. H. Witten. 1990. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ.

[6] T. C. Bell, I. H. Witten, and J. G. Cleary. 1989. Modeling for text compression. *Comput. Surv.* 21, 4 (1989), 557–591.

[7] D. Blalock, S. Madden, and J. Guttag. 2018. Sprintz: Time series compression for the internet of things. *ACM Interact., Mob., Wear. Ubiq. Technol.* 2, 3 (2018), 93.1–93.23.

[8] A. Bookstein and S. T. Klein. 1993. Is Huffman coding dead? *Computing* 50, 4 (1993), 279–296.

[9] R. M. Capocelli and A. De Santis. 1990. Minimum codeword length and redundancy of Huffman codes. In *Proceedings of the International Symposium on Coding Theory and Applications*, Vol. 514. Lecture Notes in Computer Science, 309–317.
[10] H.-C. Chen, Y.-L. Wang, and Y.-F. Lan. 1999. A memory-efficient and fast Huffman decoding algorithm. *Inform. Process. Lett.* 69, 3 (1999), 119–122.
[11] Y. Choueka, S. T. Klein, and Y. Perl. 1985. Efficient variants of Huffman codes in high level languages. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM Press, 122–130.
[12] K.-L. Chung and J.-G. Wu. 2001. Faster implementation of canonical minimum redundancy prefix codes. *J. Inform. Sci. Eng.* 17, 2 (2001), 341–345.
[13] K.-L. Chung. 1997. Efficient Huffman decoding. *Inform. Process. Lett.* 61, 2 (1997), 97–99.
[14] J. G. Cleary and I. H. Witten. 1984. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.* 32, 4 (1984), 396–402.
[15] J. B. Connell. 1973. A Huffman-Shannon-Fano code. *Proc. IEEE* 61, 7 (1973), 1046–1047.
[16] G. V. Cormack and R. N. Horspool. 1984. Algorithms for adaptive Huffman codes. *Inform. Process. Lett.* 18, 3 (1984), 159–165.
[17] J. Duda. 2009. Asymmetric numeral systems. *CoRR* abs/0902.0271 (2009), 1–47. Retrieved from: http://arxiv.org/abs/0902.0271.
[18] J. Duda. 2013. Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *CoRR* abs/1311.2540 (2013), 1–24. Retrieved from: http://arxiv.org/abs/1311.2540.
[19] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. 2015. The use of asymmetric numeral systems as an accurate replacement for Huffman coding. In *Proceedings of the Picture Coding Symposium*. IEEE, 65–69.
[20] N. Faller. 1973. An adaptive system for data compression. In *Proceedings of the 7th Asilomar Conference on Circuits, Systems, and Computers*. IEEE, 593–597.
[21] A. Fariña, N. R. Brisaboa, G. Navarro, F. Claude, Á. S. Places, and E. Rodríguez. 2012. Word-based self-indexes for natural language text. *ACM Trans. Inform. Syst.* 30, 1 (2012), 1:1–1:34.
[22] P. M. Fenwick. 2012. PPM compression without escapes. *Softw.—Practice Exper.* 42, 2 (2012), 255–260.
[23] A. S. Fraenkel and S. T. Klein. 1993. Bounding the depth of search trees. *Comput. J.* 36, 7 (1993), 668–678.
[24] T. Gagie, G. Navarro, Y. Nekrich, and A. Ordóñez Pereira. 2015. Efficient and compact representations of prefix codes. *IEEE Trans. Inform. Theor.* 61, 9 (2015), 4999–5011.
[25] R. G. Gallager. 1978. Variations on a theme by Huffman. *IEEE Trans. Inform. Theor.* IT-24, 6 (1978), 668–674.
[26] S. W. Golomb. 1966. Run-length encodings. *IEEE Trans. Inform. Theor.* IT–12, 3 (1966), 399–401.
[27] M. Hankamer. 1979. A modified Huffman procedure with reduced memory requirements. *IEEE Trans. Commun.* 27, 6 (1979), 930–932.
[28] R. Hashemian. 1995. High speed search and memory efficient Huffman coding. *IEEE Trans. Commun.* 43, 10 (1995), 2576–2581.
[29] R. Hashemian. 2004. Condensed table of Huffman coding, a new approach to efficient decoding. *IEEE Trans. Commun.* 52, 1 (2004), 6–8.
[30] D. S. Hirschberg and D. A. Lelewer. 1990. Efficient decoding of prefix codes. *Commun. ACM* 33, 4 (1990), 449–459.
[31] P. G. Howard and J. S. Vitter. 1994. Design and analysis of fast text compression based on quasi-arithmetic coding. *Inform. Process. Manag.* 30, 6 (1994), 777–790.
[32] T. C. Hu and K. C. Tan. 1972. Path length of binary search trees. *SIAM J. Appl. Math.* 22, 2 (1972), 225–234.
[33] D. A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proc. Inst. Radio Engineers* 40, 9 (1952), 1098–1101.
[34] V. Iyengar and K. Chakrabarty. 1997. An efficient finite-state machine implementation of Huffman decoders. *Inform. Process. Lett.* 64, 6 (1997), 271–275.
[35] J. Kärkkäinen and G. Tischler. 2013. Near in place linear time minimum redundancy coding. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press, 411–420.
[36] J. Katajainen, A. Moffat, and A. Turpin. 1995. A fast and space-economical algorithm for length-limited coding. In *Proceedings of the International Symposium on Algorithms and Computation*. LNCS 1004, Springer-Verlag, 12–21.
[37] S. T. Klein. 2000. Skeleton trees for the efficient decoding of Huffman encoded texts. *Inform. Retriev.* 3, 1 (2000), 7–23.
[38] S. T. Klein and D. Shapira. 2011. Huffman coding with non-sorted frequencies. *Math. Comput. Sci.* 5, 2 (2011), 171–178.
[39] D. E. Knuth. 1985. Dynamic Huffman coding. *J. Algor.* 6, 2 (1985), 163–180.
[40] L. G. Kraft. 1949. *A Device for Quantizing, Grouping, and Coding Amplitude Modulated Pulses*. Master's thesis. MIT, Cambridge, MA.
[41] L. L. Larmore and D. S. Hirschberg. 1990. A fast algorithm for optimal length-limited Huffman codes. *J. ACM* 37, 3 (1990), 464–473.
[42] D. A. Lelewer and D. S. Hirschberg. 1987. Data compression. *Comput. Surv.* 19, 3 (1987), 261–296.

[43] M. Liddell and A. Moffat. 2001. Length-restricted coding in static and dynamic frameworks. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society, 133–142.
[44] M. Liddell and A. Moffat. 2006. Decoding prefix codes. *Softw.—Practice Exper.* 36, 15 (2006), 1687–1710.
[45] M. Liddell and A. Moffat. 2007. Incremental calculation of minimum-redundancy length-restricted codes. *IEEE Trans. Commun.* 55, 3 (2007), 427–435.
[46] W.-W. Lu and M. P. Gough. 1993. A fast-adaptive Huffman coding algorithm. *IEEE Trans. Commun.* 41, 4 (1993), 535–538.
[47] D. Manstetten. 1992. Tight upper bounds on the redundancy of Huffman codes. *IEEE Trans. Inform. Theor.* 38, 1 (1992), 144–151.
[48] B. McMillan. 1956. Two inequalities implied by unique decipherability. *Inst. Radio Eng. Trans. Inform. Theor.* IT-2 (1956), 115–116.
[49] R. L. Milidiú and E. S. Laber. 2001. Bounding the inefficiency of length-restricted prefix codes. *Algorithmica* 31, 4 (2001), 513–529.
[50] R. L. Milidiú, E. S. Laber, L. O. Moreno, and J. C. Duarte. 2003. A fast decoding method for prefix codes. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society, 438.
[51] R. L. Milidiú, E. S. Laber, and A. A. Pessoa. 1999. Bounding the compression loss of the FGK algorithm. *J. Algor.* 32, 2 (1999), 195–211.
[52] R. L. Milidiú, A. A. Pessoa, and E. S. Laber. 1998. In-place length-restricted prefix coding. In *Proceedings of the Symposium on String Processing and Information Retrieval*. IEEE Computer Society, 50–59.
[53] R. L. Milidiú, A. A. Pessoa, and E. S. Laber. 2001. Three space-economical algorithms for calculating minimum-redundancy prefix codes. *IEEE Trans. Inform. Theor.* 47, 6 (2001), 2185–2198.
[54] A. Moffat. 1990. Implementing the PPM data compression scheme. *IEEE Trans. Commun.* 38, 11 (1990), 1917–1921.
[55] A. Moffat. 1999. An improved data structure for cumulative probability tables. *Softw.—Practice Exper.* 29, 7 (1999), 647–659.
[56] A. Moffat and J. Katajainen. 1995. In-place calculation of minimum-redundancy codes. In *Proceedings of the Workshop on Algorithms and Data Structures*. Springer-Verlag, LNCS 955, 393–402. Source code available from http://people.eng.unimelb.edu.au/ammoffat/inplace.c.
[57] A. Moffat, R. Neal, and I. H. Witten. 1998. Arithmetic coding revisited. *ACM Trans. Inform. Syst.* 16, 3 (1998), 256–294. Source code available from http://people.eng.unimelb.edu.au/ammoffat/arith_coder.
[58] A. Moffat and M. Petri. 2017. ANS-based index compression. In *Proceedings of the International Conference on Knowledge and Information Management*. ACM Press, 677–686.
[59] A. Moffat and M. Petri. 2018. Index compression using byte-aligned ANS coding and two-dimensional contexts. In *Proceedings of the International Conference on Web Search and Data Mining*. ACM Press, 405–413.
[60] A. Moffat, N. Sharman, I. H. Witten, and T. C. Bell. 1994. An empirical evaluation of coding methods for multi-symbol alphabets. *Inform. Process. Manag.* 30, 6 (1994), 791–804.
[61] A. Moffat and A. Turpin. 1997. On the implementation of minimum-redundancy prefix codes. *IEEE Trans. Commun.* 45, 10 (1997), 1200–1207.
[62] A. Moffat and A. Turpin. 1998. Efficient construction of minimum-redundancy codes for large alphabets. *IEEE Trans. Inform. Theor.* 44, 4 (1998), 1650–1657.
[63] A. Moffat and A. Turpin. 2002. *Compression and Coding Algorithms*. Kluwer Academic Publishers, Boston, MA.
[64] Y. Nekritch. 2000. Byte-oriented decoding of canonical Huffman codes. In *Proceedings of the IEEE International Symposium on Information Theory*. IEEE, 371.
[65] A. Novoselsky and E. Kagan. 2016. Remark on "Algorithm 673: Dynamic Huffman Coding." *ACM Trans. Math. Softw.* 42, 1 (2016), 10.
[66] O. Petersson and A. Moffat. 1995. A framework for adaptive sorting. *Discrete Appl. Math.* 59, 2 (1995), 153–179.
[67] K. Sayood. 2006. *Introduction to Data Compression* (3rd ed.). Morgan Kaufmann, San Francisco, CA.
[68] M. Schindler. 1998. A fast renormalisation for arithmetic coding. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press, 572.
[69] E. S. Schwartz and B. Kallick. 1964. Generating a canonical prefix encoding. *Commun. ACM* 7, 3 (1964), 166–169.
[70] C. E. Shannon. 1948. A mathematical theory of communication. *Bell Syst. Tech. J.* 27 (1948), 379–423, 623–656.
[71] A. Siemiński. 1988. Fast decoding of the Huffman codes. *Inform. Process. Lett.* 26, 5 (1988), 237–241.
[72] P. Skibinski and S. Grabowski. 2004. Variable-length contexts for PPM. In *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press, 409–418.
[73] G. Stix. 1991. Profile: Information theorist David A. Huffman. *Sci. Amer.* 265, 3 (1991), 54–58. Reproduced at http://www.huffmancoding.com/my-uncle/scientific-american.
[74] J. A. Storer. 1988. *Data Compression: Methods and Theory*. Computer Science Press, Rockville, MD.

[75] H. Tanaka. 1987. Data structure of the Huffman codes and its application to efficient encoding and decoding. *IEEE Trans. Inform. Theor.* IT-33, 1 (1987), 154–156.
[76] A. Turpin and A. Moffat. 1995. Practical length-limited coding for large alphabets. *Comput. J.* 38, 5 (1995), 339–347.
[77] A. Turpin and A. Moffat. 1998. Comment on "Efficient Huffman decoding" and "An Efficient Finite-State Machine Implementation of Huffman Decoders."*Inform. Process. Lett.* 68, 1 (1998), 1–2.
[78] A. Turpin and A. Moffat. 2000. Housekeeping for prefix coding. *IEEE Trans. Commun.* 48, 4 (2000), 622–628. Source code available from http://people.eng.unimelb.edu.au/ammoffat/mr_coder/.
[79] J. van Leeuwen. 1976. On the construction of Huffman trees. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*. Edinburgh University Press, Edinburgh University, Scotland, 382–410.
[80] D. C. Van Voorhis. 1974. Constructing codes with bounded codeword lengths. *IEEE Trans. Inform. Theor.* IT-20, 2 (1974), 288–290.
[81] J. S. Vitter. 1987. Design and analysis of dynamic Huffman codes. *J. ACM* 34, 4 (1987), 825–845.
[82] J. S. Vitter. 1989. Algorithm 673: Dynamic Huffman coding. *ACM Trans. Math. Software* 15, 2 (1989), 158–167.
[83] P.-C. Wang, Y.-R. Yang, C.-L. Lee, and H.-Y. Chang. 2005. A memory-efficient Huffman decoding algorithm. In *Proceedings of the International Conference on Advanced Information Networking and Applications*. IEEE Computer Society Press, 475–479.
[84] I. H. Witten, A. Moffat, and T. C. Bell. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images* (2nd ed.). Morgan Kaufmann, San Francisco, CA.
[85] I. H. Witten, R. M. Neal, and J. G. Cleary. 1987. Arithmetic coding for data compression. *Commun. ACM* 30, 6 (1987), 520–541.
[86] J. Zobel and A. Moffat. 1995. Adding compression to a full-text retrieval system. *Softw.—Practice Exper.* 25, 8 (1995), 891–903.

Received December 2018; revised May 2019; accepted May 2019