Note

Compressed depth sequences

Travis Gagie *

Department of Computer Science, University of Toronto, Canada

ARTICLE INFO

Article history:
Received 26 June 2006
Received in revised form 17 October 2008
Accepted 5 November 2008
Communicated by G. Italiano

Keywords:
Data structures
Data compression

ABSTRACT

We show how, given a probability distribution \( P \) over a set of size \( n \), in \( O(n) \) time we can construct an efficient data structure that stores a code with less than 3 bits redundancy, and takes \( o(n) \) bits of space when \( P \) consists of \( o(n \log n) \) runs of nearly equal probabilities.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

As terabytes of data accumulate, many papers have been written on the importance of compression and, more specifically, the need for fast and compact data structures — either succinct, compressed or both. In this paper we introduce a fast and compact data structure for a natural application: compression itself.

Suppose we are building mobile agents that will exchange and store high-precision sensory data. The data will be drawn from a very large set but according to a distribution with low entropy; to save power, bandwidth and memory, we want the agents to compress the data, which raises the question of how much memory is needed to compress well. Assume we know or can accurately estimate the distribution (otherwise, the agents must build a code adaptively and, so, whenever two agents communicate and want to update the code, they must either both save another copy or broadcast the update to every other agent). Our data structure stores a good code compactly when, as seems likely with sensory data, the distribution consists of relatively few runs of nearly equal probabilities. It may also be useful in other situations in which the available memory is small compared to the alphabet, e.g., when using cache memory for character-based compression of Chinese or word-based compression of English.

Let \( P = p_1, \ldots, p_n \) be a given probability distribution over a set of possible data. Shannon [8] showed that, if each datum is drawn independently according to \( P \), then we cannot expect to send or store it in fewer than \( H(P) \) bits, where \( H(P) = \sum_{i=1}^{n} p_i \log(1/p_i) \) is the entropy of \( P \). A Huffman code [4], for example, has redundancy less than 1 bit; that is, its expected codeword length is in the interval \([H(P), H(P) + 1]\). In the worst case, however, storing a Huffman code takes \( \Omega(n \log n) \) bits; we include a brief proof of this lower bound as an appendix. In fact, even storing a code with constant redundancy takes \( \Omega(n) \) bits in the worst case, which may be prohibitive when there are millions or billions of possible data. To see why, let \( S \) be a string of length \( m \) over an alphabet of size \( n \) and suppose \( P \) is the normalized distribution of the characters in \( S \); notice that, by the definition of entropy, \( H(P) \leq \log m \). By a simple counting argument, storing \( S \) takes \( m \log n \) bits in the worst case. However, given an unambiguous code \( C \) with constant redundancy \( r \) for \( P \), we can store \( S \) in \((H(P) + r)m \leq m \log(2^r m) \) bits plus the space needed to store \( C \). To do this, we simply replace each character in \( S \) by its codeword. Thus, when \( m = n/2^{r+1} \), for example, storing \( C \) takes at least \( m \log n - m \log(2^r m) = n/2^{r+1} \in \Omega(n) \) bits in the worst case.
Symbol grouping (see, e.g., [1]) is often used with large sets: partition the data into bins, store a good code for the bins and let the codeword for each datum be the codeword for its bin followed by its position in the bin. This works well in practice because we can often put adjacent data in the same bin, reducing the complexity of the partition, and still obtain a good code; we formalize this idea to obtain theoretical bounds on symbol grouping that hold regardless of the distribution’s shape. Previous research has shown symbol grouping also works well in theory provided the distribution is monotonic or we have fast access to a permutation that sorts the data into order by probability — but simply storing such a permutation takes \( \log n! \in \Theta(n \log n) \) bits in the worst case. Let \( R \) be the number of runs (maximal consecutive subsequences of equal values) in the sequence \( [\log \min(1/p_1, n)], \ldots, [\log \min(1/p_n, n)] \). In this paper we show how, if we partition the data according to those runs, then the redundancy is less than 3 bits and we can store the code \( C \) in a new data structure, called a compressed depth sequence, that

- takes \( O(R) \) words of memory,
- can be constructed in \( O(n) \) time,
- given \( i \), returns the codeword \( C(i) \) for the \( i \)th datum in \( O(|C(i)|) \) time,
- given \( C(i) \), returns \( i \) in \( O(|C(i)|) \) time,

on a unit-cost word RAM with \( \Theta(\log n) \)-bit words.

We know of no approach with strictly better bounds than ours. For example, suppose \( R = o(n/\log n) \) and we use standard symbol grouping with a Huffman code for the runs; then building the code takes \( \Omega(R \log R) \subset o(n) \) time in the worst case. Now suppose \( R = o(n/\log n) \) and we use an algorithm due to Hu and Tucker [5] to construct an order-preserving code for the data; then we can store the code in \( O(n) \) bits — because we need not store a permutation — but \( O(R) \) words of memory are \( O(R \log n) \subset o(n) \) bits.

In Section 2 we define a compressed depth sequence — essentially a high-degree search tree built on a run-length encoded sequence. In Section 3 we specify our implementation and show how a compressed depth sequence can be constructed in linear time. Finally, in Section 4 we show how to perform the queries needed for compression and decompression; this is the most technical section, but we simplify it by breaking queries down and showing how to handle them at each level of the compressed depth sequence.

2. Definition of a compressed depth sequence

Suppose there exists a binary tree whose leaves, from left to right, have depths \( d_1, \ldots, d_n \). We call \( D = d_1, \ldots, d_n \) a depth sequence. The compressed depth sequence (CDS) for \( D \) is a tree \( T \) whose root has a child, which is a leaf, for every run of the minimum value in \( D \); it also has a child, which is an internal node, for every maximal consecutive subsequence \( d_i, \ldots, d_j \) that does not include the minimum value in \( D \) — this child’s subtree is the CDS for \( d_i, \ldots, d_j \). Notice there is exactly one leaf for each run, and every internal node has at least one child that is a leaf. Thus, the number of nodes in \( T \) is at most twice the number of runs in \( D \). Fig. 1 shows an example.

Each node in \( T \) has a run-length, a run-depth and a run-width, which are stored at that node. A leaf’s run-length and run-width are both equal to the corresponding run’s length, and its run-depth is the value in the run; this means the list of leaves’ run-lengths and run-depths is the run-length encoding of \( D \). An internal node’s run-length is the sum of its children’s run-lengths and its run-depth is the same as that of its leaf siblings. (Notice \( T \)’s root has run-length \( n \); we define its run-depth to be 0.) Suppose \( v \) is an internal node with run-depth \( d \) whose subtree is the CDS for \( d_i, \ldots, d_j \); then \( v \)’s run-width is the maximum number of trees in an ordered forest of binary trees on a total of \( j - i + 1 \) leaves that, from left to right, have depths \( d_i - d, \ldots, d_j - d \). (Notice \( T \)’s root has run-width 1.) Fig. 2 shows the run-lengths, run-widths and run-depths in the CDS from Fig. 1.

We now prove a technical lemma relating an internal node’s run-width to the sum of its children’s run-widths. We use this relationship in Section 4 when analyzing how easily we can encode pointers to nodes and navigate in a CDS.

**Lemma 1.** Suppose \( v \) is an internal node with run-depth \( d \) and run-width \( w \) in a CDS, and \( v \)’s children have run-depth \( d + g \) and run-widths that sum to \( W \); then \( w = \lceil W/2^g \rceil \).

**Proof.** Suppose \( v \)’s subtree is the CDS for \( d_i, \ldots, d_j \). Then there exists an ordered forest \( F_1 \) of \( w \) binary trees on a total of \( j - i + 1 \) leaves with depths \( d_i - d, \ldots, d_j - d \), and an ordered forest \( F_2 \) of \( W \) binary trees on a total of \( j - i + 1 \) leaves with depths \( d_i - d - g, \ldots, d_j - d - g \)
Let $F_3$ be the ordered forest obtained by deleting every node in $F_1$ with depth less than $g$. Notice that, since $F_1$ contains at most $2^g w$ nodes with depth $g$, $F_3$ contains at most $2^g w$ trees, on a total of $i - j + 1$ leaves with depths $d_i - d - g, \ldots, d_j - d - g$. For each occurrence of $d + g$ in $d_i, \ldots, d_j$, there is a tree in $F_3$ containing only a root; for each maximal consecutive subsequence $d_i, \ldots, d_j$ not containing $d + g$, there are binary trees on a total of $\ell - k + 1$ leaves with depths $d_i - d - g, \ldots, d_j - d - g$. Therefore, $w \leq 2^g w$.

Now consider a forest of $\lceil W/2^g \rceil$ binary trees on a total of $W$ leaves, each with depth $g$. Let $F_2$ be the ordered forest obtained from this forest by replacing its leaves with the $W$ roots in $F_2$. Notice $F_2$ contains $\lceil W/2^g \rceil$ binary trees on a total of $j - i + 1$ leaves with depths $d_i - d, \ldots, d_j - d$. Therefore, $w \leq \lceil W/2^g \rceil$. □

### 3. Constructing a compressed depth sequence

Let $P = p_1, \ldots, p_n$ be a probability distribution over a set of possible data and let $R$ be the number of runs in $\lceil \log \min(1/p_1, n) \rceil, \ldots, \lceil \log \min(1/p_n, n) \rceil$. Let $D = d_1, \ldots, d_n$, where $d_i = \lceil \log \min(1/p_i, n) \rceil + 2$; notice $D$ also contains $R$ runs, and $\sum_{i=1}^{n} p_i d_i < H(P) + 3$. The following lemma shows $D$ is a depth sequence.

**Lemma 2.** For any probability distribution $P = p_1, \ldots, p_n$, there exists a binary tree whose leaves have depths at most $\lceil \log \min(1/p_1, n) \rceil + 2, \ldots, \lceil \log \min(1/p_n, n) \rceil + 2$.

**Proof.** Gilbert and Moore [3] showed that, for any probability distribution $P' = p'_1, \ldots, p'_n$, there exists a binary tree whose leaves have depths at most $\lceil \log (1/p'_1) \rceil + 1, \ldots, \lceil \log (1/p'_n) \rceil + 1$; if each $p'_i = (p_i + 1/n)/2$, then the leaves have depths at most $\lceil \log \min(1/p_i, n) \rceil + 2, \ldots, \lceil \log \min(1/p_n, n) \rceil + 2$. □

**Theorem 3.** Given a probability distribution $P = p_1, \ldots, p_n$, it takes $O(n)$ time to construct the CDS for $D = d_1, \ldots, d_n$, where $d_i = \lceil \log \min(1/p_i, n) \rceil + 2$.

**Proof.** We use an algorithm due to Evans and Kirkpatrick [2] to construct a binary tree $T_1$ whose leaves have depths at most $d_1, \ldots, d_n$, in $O(n)$ time. Let $T_2$ be the tree obtained by inserting enough internal nodes above $T_1$'s $i$th leaf that it has depth $d_i$, for $1 \leq i \leq n$. Notice $T_2$ contains at least $n - 1$ internal nodes and $T_2$ contains at most $2^{\lceil \log n \rceil + 2} - 1 < 8n - 1$ internal nodes; thus, constructing $T_2$ from $T_1$ takes $O(n)$ time.

We perform a partial breadth-first traversal of $T_2$ using a queue, until we reach a depth $d$ at which there are leaves. We discard every node in $T_2$ with depth between 1 and $d - 1$. Consider the contents of the queue when we stop — the nodes with depth $d$ in $T_2$, which appear in order from left to right. For each maximal consecutive subsequence of $k$ leaves in the queue, we add a child to the root — this child stores run-length and run-width $k$ and run-depth $d$. For each maximal consecutive subsequence of internal nodes in the queue, we copy that sequence into a new queue and add a child to the root; we then recurse — we start a new breadth-first traversal from the nodes in $q$ but this time, when we stop, we attach children to $v$ instead of to the root.

This process yields the shape of the CDS $T$ for $D$, with the correct information stored at each leaf. Since we visit each node in $T_2$ once during our traversals, if we store each node’s children as a linked list, then we use a total of $O(n)$ time. Notice that, using Lemma 1, it takes $O(1)$ time to compute the information stored at an internal node from the information stored at its children. Therefore, by starting at the leaves and working upward, we can compute and store the correct information at every internal node in $O(n)$ time. □

It is simpler, initially, to construct $T$ with internal nodes storing linked lists of their children, but we will eventually want them in a more efficient data structure. For each internal node $v$ in $T$, we apply the following theorem, due to Mehlhorn [7], to the linked list of $v$’s children; we treat $v$’s children as having probabilities proportional to their run-widths. We then replace the linked list by the resulting binary search tree (BST). Since Mehlhorn’s construction is linear-time, replacing every linked list takes a total of $O(n)$ time.
Theorem 4 (Mehlhorn, 1977). Given a probability distribution \( P = p_1, \ldots, p_n \), it takes \( O(n) \) time to construct a BST whose keys, in order, are stored at depths at most \( \log(1/p_1), \ldots, \log(1/p_n) \).

Finally, we augment each BST so that each node stores the sums of the run-lengths and run-widths in its subtree in the BST. For example, suppose \( v_1 \) is an internal node in \( T \) whose children have run-widths that sum to \( w \); also, \( v_2 \) is a child of \( v_1 \) with run-width \( w \). Since we treat \( v_2 \) as having probability \( w/W \) i.e., proportional to \( w \) when building the BST \( B \) storing \( v_1 \)'s children, \( v_2 \)'s depth in \( B \) is at most \( \log(W/w) \). As well as its own run-length, run-depth and run-width, \( v_2 \) stores the sums of the run-lengths and run-widths in its subtree in \( B \).

Notice that each run-length, run-width and run-depth is an \( O(\log n) \)-bit integer; thus, the sum of any subset of the \( R \leq n \) run-lengths or run-widths is also an \( O(\log n) \)-bit integer. Since BSTs take linear space, the CDS for \( D \) takes a total of \( O(R) \) words of memory on a unit-cost word RAM with \( \Theta(\log n) \)-bit words. Henceforth, whenever we refer to the CDS \( T \), we mean with it constructed this way.

4. Coding with a compressed depth sequence

Suppose Alice and Bob are mobile agents, each with copies of our CDS \( T \), and Alice wants to send Bob the ith possible datum. To do this, she first searches for the leaf \( v \) corresponding to the run in \( D \) containing \( d_i \). She starts at the root and descends to \( v \), using the following lemma and theorem to find each of \( v \)'s ancestors.

Lemma 5. Let \( T \) be the CDS for a depth sequence \( D = d_1, \ldots, d_n \). Let \( v_i \) be an internal node of \( T \) whose subtree is the CDS for \( d_i, \ldots, d_n \), and whose children's run-widths sum to \( w \). Let \( v_j \) be a child of \( v_i \) with run-width \( w \) that corresponds to a consecutive subsequence \( d_j, \ldots, d_k \) in \( d_i, \ldots, d_n \). For any integer \( i \) with \( k \leq t \leq \ell \), given a pointer to \( v_i \) and \( t - i + 1 \), it takes \( O(\log(W/w)) \) time to find \( v_j \).

Proof. Let \( v_j \) be a child of \( v_i \). Consider \( v_j \)'s children that are strictly to the left of \( v_j \); suppose their run-lengths sum to \( L \). If \( v_j \) is to the left of \( v_j \) or \( v_j = v_j \), then \( i \leq k - i - t - t \). If \( v_j \) is strictly to the right of \( v_j \), then \( L \geq \ell - i + 1 \) \( t - i \). Thus, \( v_j \) is the rightmost child of \( v_j \) such that its left-siblings' run-lengths sum to at most \( t \).

Let \( B \) be the BST storing \( v_j \)'s children. Since \( B \) is augmented so that each node stores the sum of the run-lengths in its subtree in \( B \), we can find \( v_j \) in time proportional to its depth, which is at most \( \log(W/w) \). \( \square \)

Theorem 6. Let \( T \) be the CDS for a depth sequence \( D = d_1, \ldots, d_n \). Given \( i \), it takes \( O(d_i) \) time to find both the leaf corresponding to the run in \( D \) containing \( d_i \), and the offset to \( d_i \) from the beginning of that run.

Proof. Let \( v_1, \ldots, v_k \) be the nodes on the path from the root, \( v_1 \), to the desired leaf, \( v_k \). For \( 1 \leq \ell \leq k \), let \( w_\ell \) be the run-width of \( v_\ell \). For \( 1 \leq \ell \leq k - 1 \), let \( W_\ell \) be the sum of \( v_\ell \)'s children's run-widths; let \( g_\ell \) be the increase from \( v_\ell \)'s run-depth to that of its children. Recall \( W_1 = 1 \) and notice \( \sum_{\ell=1}^{k-1} g_\ell = d_i \). Thus, by Lemmas 4 and 5, finding the path \( v_1, \ldots, v_k \) takes

\[
\sum_{\ell=1}^{k-1} O\left(\frac{\log W_\ell}{w_{\ell+1}}\right) \leq O\left(\sum_{\ell=1}^{k-1} \log \frac{2^{g_\ell} w_\ell}{w_{\ell+1}}\right) \leq O\left(\log w_1 + \sum_{\ell=1}^{k-1} g_\ell\right) = O(d_i)
\]
time; here, we collapsed the telescoping sum \( \sum_{\ell=1}^{k-1} \frac{\log(w_\ell/w_{\ell+1})}{w_{\ell+1}} = (\log(w_1/w_k)) \). As a side-effect of finding this path, we also learn the offset to \( d_i \) from the beginning of the run corresponding to \( v_k \). \( \square \)

Once Alice has found the leaf \( v \), she uses the following lemma and theorem to encode a pointer to \( v \) and the offset to \( d_i \), from the beginning of the run corresponding to \( v \). She sends this encoding to Bob; he uses the following lemma and theorem — in the opposite way — to recover \( i \), the datum's index.

Lemma 7. Let \( T \) be the CDS for a depth sequence \( D = d_1, \ldots, d_n \). Let \( v_1 \) be an internal node of \( T \) with run-depth \( d \) and run-width \( w_1 \), whose children have run-depth \( d + g \). For any child \( v_j \) of \( v_1 \) with run-width \( w_j \) and any integer \( i_1 \leq i_2 \leq w_2 \), the following property holds: For some integer \( i_1 \) with \( 1 \leq i_1 \leq w_1 \) and some binary string \( b \in \{0, 1\}^d \), we can compute \( \langle v_1, i_1, b \rangle \) and vice versa, in \( O(\log(w_1/w_2) + g) \) time.

Proof. Let \( L \) be the sum of \( i_2 \) and the run-widths of \( v_1 \)'s children to the left of \( v_2 \). By Lemma 4, \( L \leq 2^k w_1 \). Let \( i_1 = \lfloor (L - 1)/2^k \rfloor + 1 \), so \( 1 \leq i_1 \leq w_1 \); let \( b \) be the \( g \)-bit binary representation of \((L - 1) \mod 2^k \). Let \( B \) be the BST storing \( v_1 \)'s children. Since \( B \) is augmented so that each node stores the sum of the run-widths in its subtree in \( B \), in \( O(\log(w_1/w_2) + g) \) time we can compute \( L - \) and then \( i_1 \) and \( b \) from \( \langle v_2, i_2 \rangle \); and, in the same amount of time, we can compute \( L - \) and then \( v_2 \) and \( i_2 \) from \( \langle v_1, i_1, b_1 \rangle \). \( \square \)

Theorem 8. Let \( T \) be the CDS for a depth sequence \( D = d_1, \ldots, d_n \). Given \( i \), for any integer \( i \) with \( 1 \leq i \leq n \), the following property holds: For some binary string \( C(i) \in \{0, 1\}^d \), we can compute \( C(i) \) from \( i \), and vice versa, in \( O(d_i) \) time.

Proof. Let \( v_1, \ldots, v_k \) be the nodes on the path from the root, \( v_1 \), to the leaf \( v_k \) corresponding to the run containing \( d_i \). By Theorem 6, given \( i \), it takes \( O(d_i) \) time to find \( v_k \) and the offset \( i_k \) to \( d_i \) from the beginning of the run corresponding to \( v_k \). Notice \( i_k \) is between \( 1 \) and \( v_k \)'s run-length; since \( v_k \) is a leaf, its run-length is the same as its run-width.

For \( 1 \leq \ell \leq k \), let \( w_\ell \) be the run-width of \( v_\ell \). For \( 1 \leq \ell \leq k - 1 \), let \( g_\ell \) be the increase from \( v_\ell \)'s run-depth to that of its children's; let \( i_\ell \) and \( b_\ell \) be the integer and binary string such that we can compute \( \langle v_{\ell+1}, i_{\ell+1} \rangle \) from \( \langle v_\ell, i_\ell, b_\ell \rangle \), and vice versa, in \( O(\log(w_\ell/w_{\ell+1}) + g_\ell) \) time with Lemma 7.
By starting with $v_k$ and $i_k$ and applying Lemma 7 a total of $k - 1$ times, once for each proper ancestor of $v_k$, we obtain $b_1 \cdots b_{k-1}$ in
\[
\sum_{\ell=1}^{k-1} O \left( \log \frac{w_\ell}{w_{\ell+1}} + g_\ell \right) \leq O \left( \log w_1 + \sum_{\ell=1}^{k-1} g_\ell \right) = O(d_i)
\]
time. Let $C(i) = b_1 \cdots b_{k-1}$; notice $|C(i)| = \sum_{\ell=1}^{k-1} g_\ell = d_i$.

Similarly, starting with $C(i)$ and applying Lemma 7 a total of $k - 1$ times, we obtain $v_k$ and $i_k$ in $O(d_i)$ time. During this process — that is, as we descend to $v_k$ — we compute the sum of the run-lengths of the leaves to the left of $v_k$, which is $i - i_k$, and so obtain $i$. □

We summarize our results as the following theorem.

**Theorem 9.** Given a probability distribution $P = p_1, \ldots, p_n$ over a set of possible data, we can construct a code $C$ with redundancy less than 3 bits and store $C$ in $O(R)$ words of memory on a unit-cost word RAM with $\Theta(\log n)$-bit words, where $R$ is the number of runs in $\lceil \log \min(1/p_1, n) \rceil, \ldots, \lceil \log \min(1/p_n, n) \rceil$. Finding and storing $C$ both take $O(n)$ time. Given $i$ with $1 \leq i \leq n$, it takes $O(|C(i)|)$ time to find the codeword $C(i)$ for the $i$th datum. Similarly, given $C(i)$, it takes $O(|C(i)|)$ time to find $i$.

**Acknowledgments**

Many thanks to Faith Ellen Fich, Giovanni Manzini and Charlie Rackoff, who supervised this research, the anonymous referee whose comment on an earlier paper inspired this one, and the anonymous referee whose comments on this paper improved the presentation.

**Appendix. A lower bound for storing a Huffman code**

When Huffman’s algorithm is applied to a probability distribution $P = p_1, \ldots, p_n$ over an alphabet, it creates $n$ single-node trees, each labelled with a character from the alphabet and assigned that character’s probability. Then, until there is only one tree $T$ remaining, the algorithm iteratively removes the two trees with smallest probabilities, say $p_i$ and $p_j$, and makes them the subtrees of a new tree with probability $p_i + p_j$. The codeword for each character $a$ in the corresponding code is a binary encoding of the path from the root of $T$ to the leaf labelled $a$, with left edges represented by 0s and right edges by 1s.

Suppose that, for a particular permutation $\pi$, the probabilities $p_{\pi(1)}, \ldots, p_{\pi(n)}$ are proportional to $F_1, \ldots, F_n$, where $F_i$ is the $i$th Fibonacci number. Since $\sum_{i=1}^{n-2} F_i = F_i - 1$ for $i \geq 2$, the tree $T$ is degenerate with the leaves’ depths decreasing as their probabilities increase (the two least probable leaves are both at depth $n - 1$) [6]. Thus, given a Huffman code for $P$ and knowing whether $\pi(1)$ is greater or less than $\pi(2)$, we can recover $\pi$; it follows that storing a Huffman code takes $\Omega(n \log n)$ bits in the worst case.

**References**