Programming Techniques and Data Structures

Ian Munro Editor

A Locally Adaptive Data Compression Scheme

JON LOUIS BENTLEY, DANIEL D. SLEATOR, ROBERT E. TARJAN, and VICTOR K. WEI

ABSTRACT: A data compression scheme that exploits locality of reference, such as occurs when words are used frequently over short intervals and then fall into long periods of disuse, is described. The scheme is based on a simple heuristic for self-organizing sequential search and on variable-length encodings of integers. We prove that it never performs much worse than Huffman coding and can perform substantially better; experiments on real files show that its performance is usually quite close to that of Huffman coding. Our scheme has many implementation advantages: it is simple, allows fast encoding and decoding, and requires only one pass over the data to be compressed (static Huffman coding takes two passes).

1. INTRODUCTION

Data compression schemes can be categorized by the unit of data they transmit. Huffman [14] codes are typical of "defined-word" schemes: the context defines sequences of input symbols (which we shall call words) that are transmitted by a variable-length code. At the other extreme, Ziv-Lempel [26] codes transmit variable-length sequences of input symbols, often using a fixed-length code.

In this article we describe a defined-word scheme that uses a technique from another domain that deals with defined words: *self-organizing sequential search*, in which we wish to maintain a sequential list of words so that frequently accessed words are near the front. Our data compression scheme uses a self-organizing list as an auxiliary data structure, and employs short encodings to transmit words near the front of the list. The scheme never performs much worse than Huffman coding. If the message to be transmitted exhibits locality of reference (i.e., if the local frequency of words changes dramatically within the message), the scheme performs better than Huffman coding because a word will have a short encoding when it is used frequently and a long encoding when it is used rarely.

Section 2 describes the basic scheme and several dimensions along which it may vary. Mathematical analyses of the performance of the scheme are given in Section 3 and in the Appendix. Experimental evidence is presented in Section 4. Section 5 discusses implementation considerations, and Section 6 contains concluding remarks. A preliminary version of our results appeared as a conference paper [2].

2. THE THEME AND SOME VARIATIONS

We shall illustrate our scheme by compressing simple "telegraph" messages of words consisting of upper case letters separated by single spaces and terminated by a final "●". For concreteness, we will transmit the message

THE CAR ON THE LEFT HIT THE CAR I LEFT ●

Sender and receiver maintain identical word lists using the "move-to-front" heuristic: after a word is used it is deleted from its current position and moved to the front of the list. This attempts to ensure that frequently used words appear near the front of the list.

The list is initially empty. To transmit the word W, the sender looks it up in the list. If it is present in position I, the sender transmits I, which the receiver decodes by writing the Ith element in the list; both then move W to the front of their respective lists, shifting the words in positions $1..I-1$ to positions $2..I$. If W is not in the list of N words, the sender reacts as though it were in the $N+1$st position and sends the integer $N+1$ followed by the word W (which the receiver expects because $N+1$ is greater than the size of the current list); both sender and receiver then move W to the front of their lists. For example, after transmitting the first three words of the above message, both parties have identical lists

ON CAR THE

The next word, THE, is encoded by the integer 3. The entire message is encoded as

1 THE 2 CAR 3 ON 3 4 LEFT 5 HIT 3 5 6 I 5 ●

Each word is transmitted as a string of letters just once; subsequent occurrences are encoded by integers. The integer encoding a word is one greater than the total number of different words that have occurred since its last previous appearance [20, 21].

This trivial example illustrates the most important property of our scheme: if a word has been recently used then it will be near the front of the list and therefore have a short decimal encoding. Because the integer I requires roughly $\log_{10} I$ characters to encode, frequent words are transmitted with few characters. There are, however, many variations on the basic idea.

Lexical Analysis. English text may be divided into "words" in many ways. A simple scheme might classify each character as a word, while a more complex scheme could find true English words, together with capitalization information. Transmitting program text, executable object code, or digitally encoded pictures demands a more subtle definition of words.

List Organization Discipline. Bentley and McGeoch [1] and Sleator and Tarjan [20, 21] refer to many self-organization heuristics other than move-to-front. The transpose rule, for instance, moves the accessed element one closer to the front; it is an instance of the move-ahead-k heuristic with $k = 1$.

List Length. The above example assumed an infinite list; the scheme may also be implemented with fixed-size lists. The move-to-front scheme with a finite list induces a least-recently-used discipline of discarding words from the list (which, in this context, may be viewed as a word cache).

Encoding List Position. Position in a finite list can be encoded with a fixed-length binary code, but the scheme is usually more effective if used with a variable-length code. If the data are to be transmitted as they are read, the variable-length prefix encodings of integers described by Elias [7] and by Bentley and Yao [3] provide suitable encodings; these will be discussed in detail later. If, on the other hand, the system can make two passes over the data, then the first pass can count the number of times each list position is accessed and the second pass can encode the positions using a Huffman code.

Transmission of New Words. This is a classical problem in information theory.

We shall see several combinations of these choices in the next sections.

3. THEORETICAL ANALYSIS OF PERFORMANCE

In this section we show that the move-to-front scheme is sometimes much better than Huffman codes but can never be much worse. Here we summarize our theoretical results; the Appendix contains a more complete analysis and proofs.

A simple example shows that the move-to-front scheme can be much better than any static encoding scheme. Consider the sequence formed by repeating each of n words n times: $1^n 2^n \ldots n^n$. A static Huffman code uses roughly $\log n$ bits per word sent,[1] whereas the move-to-front scheme uses only a small constant number of bits per word.

To analyze the move-to-front scheme we need to specify a particular encoding of the integers. One method is to prefix the binary representation of the integer $i \geq 1$ with $\lfloor \log i \rfloor$ 0's. This yields a prefix code since the total length of a codeword is exactly one plus twice the number of 0's in the prefix. Once the length is known the boundary between codewords can be found. This method encodes i with $1 + 2\lfloor \log i \rfloor$ bits.

We will analyze the scheme in which the size of the move-to-front list is equal to the number of different words to be sent, and the list is initialized with all the words in their order of occurrence in the sequence of words to be sent. This is the scheme

[1] All logarithms without an explicit base are binary in this article.

that was outlined in Section 2, except that we ignore the cost of sending the raw words (we ignore this cost in both of the schemes being compared). Let $\rho_{MF}(X)$ be the average number of bits per word used to compress a sequence X with the move-to-front scheme as described above. Let $\rho_H(X)$ be the analogous quantity for a static Huffman code. With the above encoding of the integers we have

$$\rho_{MF}(X) \le 2\rho_H(X) + 1. \tag{1}$$

We will now sketch a proof of this theorem; a proof of a stronger result appears in the Appendix. Suppose in the sequence X of length N, the symbol a occurs N_a times. The distance a can move from the front of the list between successive occurrences of a is bounded by the number of other accesses between these occurrences. The average size of these gaps between successive accesses to a is about N/N_a. If a were at its average position whenever it occurred in the sequence, then the number of bits needed to transmit each occurrence would be $2 \log(N/N_a) + 1$ bits. Although a is not always at this position in the list, the concavity of the log function implies that $2 \log(N/N_a) + 1$ is an upper bound on the average cost of transmitting an a. Since the cost of transmitting a with an optimum prefix code is at least $\log(N/N_a)$, the result follows.

By using more sophisticated encodings of integers, stronger results can be proved. For example,

$$\rho_{MF}(X) \le 1 + \rho_H(X) + 2 \log(1 + \rho_H(X)). \tag{2}$$

These results compare two schemes that differ in two important ways. The move-to-front scheme is dynamic (the encoding of a word may change with time), whereas Huffman codes are static (they are fixed in advance). This should, of course, give move-to-front an advantage. On the other hand, the move-to-front scheme works on-line (words are transmitted as they appear), whereas Huffman coding is off-line (it requires a pass over all the data before anything is sent).

Gallager [10] and Knuth [16] have studied a dynamic version of Huffman coding in which an optimum code is maintained based on the frequencies of words so-far transmitted. Recently, Vitter [25] has shown that with dynamic Huffman coding the average number of bits per word is at most twice the number for static Huffman coding. He has also shown that a modified scheme uses at most one more bit per word than static Huffman coding. These dynamic schemes run on-line and thus avoid the two passes necessary for static Huffman coding, but they still have the drawback that they do not exploit locality of reference. Vitter's lower bounds on dynamic Huffman coding imply that inequalities (1) and (2) remain true if H is a dynamic Huffman scheme, provided that the additive constant 1 is replaced by 2.

A theorem in the Appendix compares the performance of the move-to-front method when compressing a discrete memoryless source with the entropy of the source. This result shows that the move-to-front scheme is asymptotically optimum (see Gallager [9]) because

$$\frac{\langle \rho_{MF}(X) \rangle}{H(s)} \to 1 \quad \text{as} \quad H(s) \to \infty, \tag{3}$$

where $H(s)$ is the entropy of the discrete memoryless source s which generates the sequence X and $\langle \rho_{MF}(X) \rangle$ is the expected number of bits per word sent by the move-to-front scheme averaged over all sequences.

If the cost of rearranging the items in the list is high, then a modified version of the move-to-front scheme called intermittent-move-to-front can be used. This is actually a family of schemes parameterized by an integer $\tau \ge 1$. Rather than moving an item to the front each time it is accessed, the intermittent scheme moves an item to the front every τth time it is accessed. To do this, the method maintains a count of the number of accesses to each item since its last move. (Bitner [4] called this scheme wait-c-and-move, and analyzed it as a list updating heuristic.)

We can prove nearly as strong a theorem about the performance of intermittent-move-to-front as we can about ordinary move-to-front:

$$\rho_{IMF}(X) \le 1 + \rho_H(X) + 2\log(1 + \rho_H(X)) + \epsilon \tag{4}$$

for an arbitrary sequence X, where $\epsilon \to 0$ as the length of the sequence goes to infinity. Furthermore, when the sequence X is generated by a discrete memoryless source s, we have

$$\frac{\langle \rho_{IMF}(X) \rangle}{H(s)} \to 1 \quad \text{as} \quad H(s) \to \infty. \tag{5}$$

Therefore, the intermittent move-to-front scheme is also asymptotically optimum.

4. EXPERIMENTAL ANALYSIS OF PERFORMANCE

To gain further insight into the performance of our scheme we have implemented prototypes of the following three compression algorithms.

Byte-Level Huffman Code. A Huffman code is used on individual bytes. We did not charge for transmitting the Huffman tree, which requires less than 200 bytes.

Word-Level Huffman Code. Words were defined as longest sequences of alphanumeric and nonalphanu-

TABLE I. Compression Produced by Huffman Coding and the Move-to-Front Scheme

FILE TYPE	SIZE IN BYTES	HUFFMAN CODE		MOVE-TO-FRONT CACHE					
		BYTE	WORD	8	16	32	64	128	256
PROGRAM TEXT (C)	20593	5.285	2.958	4.253	3.785	3.385	3.049	2.917	2.822
	19118	4.576	2.321	3.071	2.356	2.245	2.196	2.183	2.233
	17160	5.248	3.026	4.672	4.147	3.630	3.217	3.033	3.019
	17470	5.210	3.520	4.510	4.028	3.814	3.637	3.413	3.379
	18441	5.288	3.585	4.505	4.088	3.645	3.388	3.263	3.270
	16282	5.150	2.316	4.044	3.517	2.784	2.368	2.253	2.271
	23225	5.321	2.219	4.025	3.269	2.711	2.408	2.269	2.113
(PASCAL)	31930	5.102	2.467	4.468	3.892	3.312	2.856	2.661	2.501
	19144	5.267	2.678	4.424	3.902	3.392	3.035	2.801	2.642
	14673	5.124	2.521	4.323	3.750	3.327	2.906	2.580	2.454
	8535	5.082	2.662	4.452	3.968	3.446	3.022	2.778	2.675
	12833	5.349	2.822	4.343	3.886	3.478	3.177	2.945	2.627
TERMINAL SESSION	142762	5.168	3.412	5.101	4.783	4.504	4.212	3.929	3.685
BOOK SECTION	20616	4.832	3.460	4.826	4.551	4.274	3.958	3.735	3.559
	18225	4.748	3.374	4.807	4.471	4.164	3.879	3.632	3.471
	22360	4.978	3.698	4.985	4.695	4.424	4.140	3.992	3.852
	17471	4.878	3.747	4.880	4.629	4.382	4.125	3.967	3.847
	15709	4.883	3.617	4.971	4.638	4.363	4.108	3.921	3.739
	15104	4.856	3.453	4.837	4.486	4.136	3.840	3.637	3.509
	19113	5.073	3.530	4.923	4.579	4.239	3.934	3.729	3.594
	22346	4.902	3.054	4.723	4.270	3.819	3.547	3.313	3.134

meric characters. This divides the input stream into two disjoint classes, which we therefore compressed separately; the decoder knows to alternate between the two classes when decoding. No case information was recorded; "the" and "The" were thus treated as distinct words. The words were transmitted using a byte-level Huffman code (as above, without charge for transmitting the tree).

MTF Cache. Many attributes of this scheme are the same as for the word-level Huffman code, including word definition, two-word sets, lack of case information, and transmission of words. Because there are two-word sets, the 8-element MTF cache stores 16 words (8 alphanumeric and 8 nonalphanumeric). For ease of implementing the prototype, we encoded the position in the list by a Huffman code, which implies that an implementation would have to make two passes over the data.

None of the implementations actually compresses and restores data; rather, they measure the efficiency of the various approaches. The performance of the MTF cache scheme tested can be no worse than the method using a fixed encoding of the positions in the list, but it could be better. We view this work only as a preliminary experiment to demonstrate the plausibility of the scheme and to gain insight into its behavior.

The results of the experiments are presented in Table I. The numbers show the bits per character used by each scheme (the original encoding uses 8

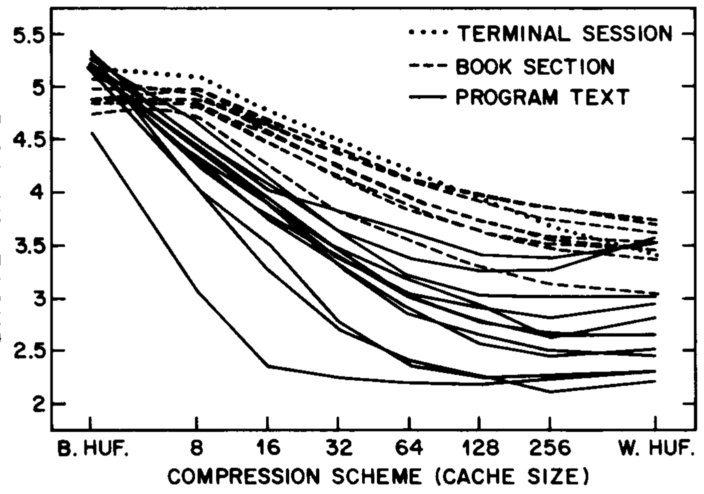

FIGURE 1. Graph of the Data in Table I

bits per character, but that can be easily reduced to 7). The C and Pascal programs were written by several different programmers; the book sections (written by two sets of multiple authors) include TROFF formatting commands. The terminal session is the transcript of a several-hour session (we include it to underscore the point that the performance of the various schemes is quite dependent on the context).

Most of the data in the table are represented in the graph of Figure 1, in which each file is represented

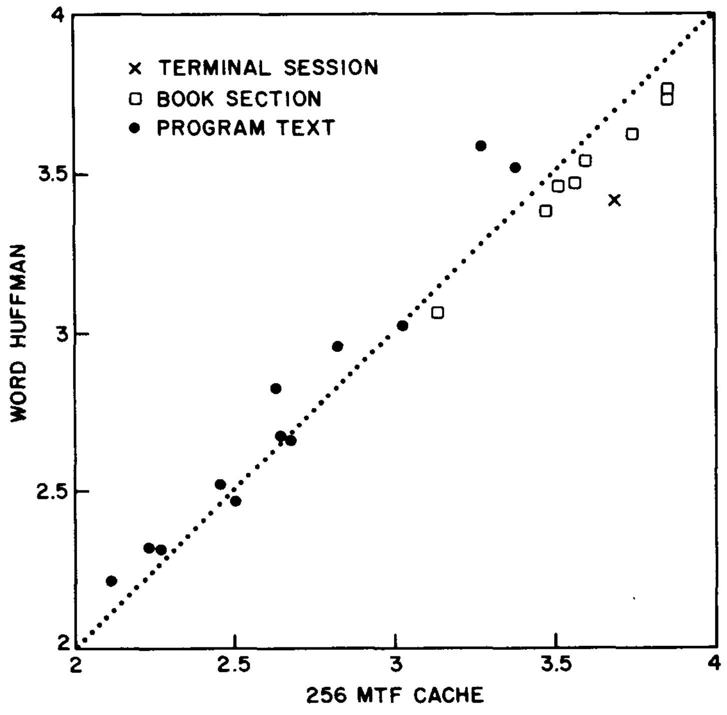

FIGURE 2. Comparison Between Word-Based Huffman Coding and the Move-to-Front Scheme With a Word-List Size of 256

by a line. Each line represents (from left to right) the cost of the byte-level Huffman encoding, MTF cache encodings of increasing size, and the word-level Huffman encoding.

This graph tells several stories. It is obvious that the three types of input have different characteristics; there are enough data on program text and book sections to draw plausible conclusions. Byte-level Huffman codes for the book sections are roughly as effective as the 8-element MTF cache; as the cache increases the move-to-front scheme shows steady improvement, but even at the 256-element cache it is still somewhat inferior to word-level Huffman codes. The relative ordering of the book sections is quite stable through all the encoding schemes.

The programs differ dramatically from the book sections. Byte-level Huffman codes are less effective for programs (presumably due to a larger character set), but the 8-element cache is already down to about 4 bits per character (presumably due to strong locality of reference). The improvement with increasing cache size is erratic; some documents even exhibit nonmonotonicity (retransmitting a few words costs less than sending longer codes for the words that are kept in the cache). The 256-element MTF cache is a little more effective than word-level Huffman codes. The outlying program contains a large table that represents legal hyphenations of English words, so one might expect it to be quite different.

The word-level Huffman encoding is compared to the 256-element MTF cache in the graph of Figure 2. For program text, the 256-element MTF cache is usually superior to Huffman codes; it requires between 2.1 and 3.4 bits per character (which represents a space reduction factor of between 2 and 4). For book sections, the word Huffman code is slightly better; it uses from 3.05 to 3.75 bits per character. All in all, though, the two schemes are quite comparable.

5. IMPLEMENTATION ISSUES

It is easy to implement the move-to-front scheme if efficiency is not important. In this section we describe an efficient implementation of the scheme, designed to minimize the worst-case running time to within a constant factor. The time to compress (encode) or expand (decode) a word is proportional to the total number of bits in the expanded and compressed forms of the word. We shall assume that the move-to-front word list is of fixed finite size n. Our computation model is a sequential random access machine with unit cost measure.

Let us define the compression and expansion algorithms precisely. There are two parts to each: we must convert a word into the corresponding integer list position (or vice-versa) and we must convert an integer into the corresponding prefix code (or vice-versa). We shall use the following operations on the word list:

position(w): Compute and return the position of word w in the word list, or $n + 1$ if w is not in the list. (The positions in the list are indexed from 1 to n.)

word(p): Compute and return the word in position p in the list.

mtf(p): Move the word in position p to the front of the list.

insert(w): Insert word w at the front of the list.

delete(p): Delete the word in position p of the list.

To manipulate the prefix codes for the integers we need two primitives:

encode(p): Compute and return the prefix code of the integer p.

decode: Read bits from the input until an entire prefix codeword has been read; then return the corresponding integer.

The compression algorithm can be implemented as the following program (written in a variant (Tarjan [23, pp. 12–14]) of the guarded command language of Dijkstra [6]) applied to a word w:

```
compress: p := position(w);
          if p < n + 1 →
            mtf(p); c := encode(1);
            output c;
          | p = n + 1 →
            delete(n); insert(w); c := encode(n + 1);
            output c; output w in raw form
          fi;
```

The expansion algorithm can be implemented as the following program:

```
expand: p := decode;
        if p < n + 1 →
          mtf(p); w := word(1);
          output w
        | p = n + 1 →
          read a word w from the input in raw
            form;
          delete(n); insert(w);
          output w
        fi;
```

To implement the primitive operations we need four data structures: one each for encoding and decoding integers and one each for converting words into list positions and vice-versa. The various operations will have the following running times: $O(|c|)$ for *encode*(p) and *decode*, where c is the relevant codeword and $|c|$ is its length; $O(\log p + |w|)$ for *position*(w) and *word*(p); $O(\log p)$ for *mtf*(p); $O(|w|)$ for *insert*(w); $O(1)$ for *delete*(n). We assume that the length of a prefix codeword grows with p; that is, $|c(p)|$ is a nondecreasing function of p, where $c(p)$ is the codeword of integer p. This assumption implies by counting that $p \leq 2^{|c(p)|}$, that is, $\log p \leq |c(p)|$. By substituting $|c|$ for $\log p$ in the running times of the operations and examining the programs above we see that the time to compress or expand is bounded by $O(|w| + |c|)$, where w and c are the word invovled and its compressed form.

It remains for us to describe the data structures and the implementation of the primitive operations. Let us begin with the data structures for prefix coding of integers. We shall describe general-purpose methods that apply to any prefix code; for specific codes such as those discussed in Section 3 and the Appendix; special-purpose algorithms can be used instead.

To implement *encode*, we use an array with positions 1 to $n + 1$, with position p holding the codeword for p. Then *encode* takes a single array access and $O(1)$ time, or $O(|c|)$ time if we charge one per bit for reading out c.

To implement *decode*, we use a *binary trie* (Knuth [15, pp. 481–499]). This is a binary tree such that each left edge (edge to a left child) is labeled 0 and each right edge (edge to a right child) is labeled 1. Each path through the tree corresponds to the word obtained by concatenating the labels of the edges along the path (top-down). To represent a prefix code, we construct a trie in which the paths from the root to the leaves (nodes with no children) represent the codewords (constructing such a trie is possible because of the prefix property). In each leaf we store the corresponding integer. To perform *decode*, we start at the trie root, read bits from the input, and follow corresponding edges of the trie until reaching a leaf; then we return the integer in the leaf. Performing *decode* takes $O(|c|)$ time.

Maintaining the word list efficiently is somewhat more complicated. We use two interlinked data structures, a binary trie to convert words into integer positions, and a binary tree to represent the order of words in the word list. (See Figure 3, p. 326).

The trie contains *marked* and *unmarked* nodes; node x is marked if the path from the root to x corresponds to (the binary representation of) a word in the word list. Given a word w in the list, the corresponding node can be found in $O(|w|)$ time by traversing the appropriate path down from the root in the trie.

The binary tree contains one node per word in the word list; the node contains the corresponding word. Symmetric order in the tree corresponds to front-to-back order in the word list. (Symmetric order is defined recursively as follows: for any node x, all nodes in its left subtree are less than x, and all nodes in its right subtree are greater than x.) The *size* of a node is the number of nodes in the subtree rooted there. Every node in the tree contains pointers to its parent and to its left and right children. Every node except those on the leftmost and rightmost paths (the paths from the root to the leftmost (smallest) and rightmost (largest) nodes) contains its size. Every node on the leftmost path contains a *mark* indicating that it is on the leftmost path.

We access the tree via a pointer to its leftmost node. If the tree is balanced, we can find the pth node in symmetric order in $O(\log p)$ time by starting at the leftmost node, walking up the leftmost path, accumulating size information in right children, and walking down into the subtree containing the pth node. Conversely, given a pointer to any node in the tree, we can compute its symmetric-order position p in $O(\log p)$ time by walking up from the node until reaching the leftmost path and then walking down to the leftmost node, accumulating size information along the way.

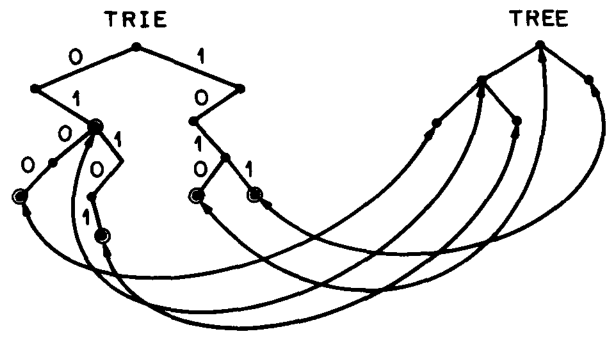

FIGURE 3. Data Structures Representing the List of Words (in Binary) [01, 1010, 0100, 01101, 1011]. Marked nodes in the trie are circled.

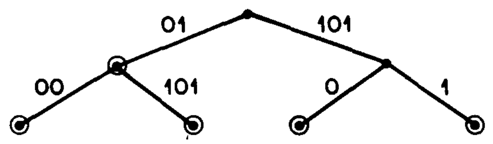

FIGURE 4. The Compressed Form of the Trie in Figure 3

The trie and the tree are linked together as follows: each marked node in the trie contains a pointer to the corresponding node in the tree, and vice-versa (see Figure 3). To compute *position*(w), we walk down the path in the trie corresponding to w, jump to the corresponding node in the tree, and compute the symmetric-order position p of this node as described above. This takes $O(\log p + |w|)$ time. To compute *word*(p), we find the pth node in the tree and return the word contained there. This takes $O(\log p)$ time, or $O(\log p + |w|)$ time if we charge one per bit for reading out w.

The update operations (*mtf*(p), *insert*(w), and *delete*(p)) are more difficult to implement because they must modify the data structures. To make these operations efficient, we make the binary search tree into a *finger search tree* with fingers at the leftmost and rightmost nodes. A finger search tree [5] is a data structure such that an insertion or deletion at a position d away from a finger takes $O(\log d)$ time. There are ways to implement a finger search tree to obtain the $O(\log d)$ time bound either in the worst case (Huddleston [12], Kosaraju [17], Tsakalidis [24]) or in the amortized (time-averaged) case (Huddleston and Mehlhorn [13], Maier and Salveter [18]). Any of these methods will suit our purposes. The only modification that must be made to these structures is that the size information must be updated after an insertion or deletion, but this does not affect the running time, given that we do not need to maintain the size information along the leftmost and rightmost paths.

We perform the update operations as follows. To carry out *mtf*(p), we find the node in position p in the tree, delete it, and insert it in the leftmost position. This takes $O(\log p)$ time. (The trie does not change.) To carry out *insert*(w), we put the word w into the trie, insert a node containing w into the tree at the leftmost position, and link the corresponding trie and tree nodes together. This takes $O(|w|)$ time. To carry out *delete*(n), we delete the nth node in the tree (this takes $O(1)$ time starting from the right finger) and delete nodes in the trie, starting with the node representing the deleted word w_n and walking up until finding a node with a marked descendant other than w_n. (The node at which we stop is w_n if w_n has at least one child, or the nearest ancestor of w_n with at least two children otherwise.)

Deleting w_n from the trie using this method takes $O(|w_n|)$ time, whereas our goal is an $O(1)$ time bound. There are at least four ways to make deletion more efficient, depending on which ground rules we are willing to accept. First, if the word list is large enough to hold all words that are ever compressed; then deletion never takes place. Second, if we are prepared to accept an amortized time bound, we can charge the time for a trie deletion to the corresponding insertion. Third, we can delete a word in the trie by merely unmarking the corresponding node, and run a background process that deletes unneeded trie nodes. Fourth, we can compress the trie, so that each edge represents a bit string rather than just a single bit. (See Figure 4.) Then a deletion requires removing at most a single edge, and even if the bit string associated with an edge is stored in a linked list, returning the entire linked list to the free list takes $O(1)$ time.

This implementation, though theoretically efficient, is more complicated than one would like. A simpler, practical alternative is to use a hash table in place of the trie for representing words and a self-adjusting search tree (Sleator and Tarjan [19, 22]) in place of the finger search tree for representing the word list. The hash table will have an $O(1)$ average time bound for inserting and deleting words, and it is conjectured that self-adjusting search trees support accesses, insertions, and deletions in the vicinity of a finger in $O(\log d)$ amortized time (Sleator and Tarjan [22]). Thus this alternative implementation is probably efficient in theory as well as in practice.

6. REMARKS

We have described a simple data compression scheme and analyzed its performance both theoretically and experimentally. Both analyses suggest that the method may be useful in practice. An intriguing area for future research is to devise other locally adaptive data compression schemes and compare them with the move-to-front scheme. Dynamic Huffman coding can be made locally adaptive by keeping a "window" as suggested by Knuth [16], maintaining a Huffman tree for word frequencies within the window. Another possibility is to maintain a dynamic Huffman tree based on a weight for each word that is incremented by one each time the word is compressed; periodically all word weights are multiplied by a constant factor less than one. Recently Elias [8] independently discovered the move-to-front scheme and derived inequalities (1) and (2). He also proposed a related scheme called *interval coding*, in which a word is encoded as a prefix code of the number of words occuring since its last appearance. Elias showed that inequalities (1) and (2) hold for interval coding (which also follows from our analysis). Interval coding always needs at least as many bits as the move-to-front scheme but is easier to implement. It would be useful to derive further results comparing these locally adaptive schemes.

It is important to note that with our scheme loss of synchronization between sender and receiver can be catastrophic, whereas this is not true with static Huffman coding. This suggests the study of adaptive schemes that might overcome this problem.

APPENDIX: ANALYSIS

To analyze our scheme we need to have specific prefix codes for the integers. Elias [7] and Bentley and Yao [3] describe a series of encoding schemes in which the integer i is encoded with roughly $\log i$ bits. The various schemes differ in their choice of trade-off between performance on small numbers and performance on large numbers.

The simplest of the schemes encodes the integer $i \geq 1$ with $1 + 2\lfloor \log i \rfloor$ bits. The encoding of i consists of $\lfloor \log i \rfloor$ 0's followed by the binary representation of i (which takes $1 + \lfloor \log i \rfloor$ bits, the first of which is a 1). This results in a prefix code since the total length of a codeword is exactly one plus twice the number of 0's in the prefix. Once the length is known the boundary between codewords can be found.

Another scheme results if we replace the $\lfloor \log i \rfloor$ 0's followed by a 1, by a two part prefix (an encoding of $1 + \lfloor \log i \rfloor$ by the above scheme) which takes $1 + 2\lfloor \log(1 + \lfloor \log i \rfloor) \rfloor$ bits. Thus we have a scheme that encodes i with $1 + \lfloor \log i \rfloor + 2\lfloor \log(1 + \log i) \rfloor$ bits. (Note that $\lfloor \log(1 + \lfloor \log i \rfloor) \rfloor = \lfloor \log(1 + \log i) \rfloor$.)

These ideas can be applied again to give an encoding for i with $1 + \lfloor \log i \rfloor + \lfloor \log(1 + \log i) \rfloor + 2\lfloor \log(1 + \log(1 + \log i)) \rfloor$ bits. This process can be continued; however, the codes that result are better only for astronomically large numbers.

Knowing the range of numbers to be encoded in advance can be used to advantage. For example, if the numbers are bounded above by n, then in the first scheme the $\lfloor \log i \rfloor$ 0's followed by a 1 can be replaced by $\lfloor \log(1 + \log n) \rfloor$ bits, giving an encoding for i with $\lfloor \log i \rfloor + \lfloor \log(1 + \log n) \rfloor$ bits. The same idea applied to the second scheme gives an encoding of i in $\lfloor \log i \rfloor + \lfloor \log(1 + \log i) \rfloor + \lfloor \log(1 + \log(1 + \log n)) \rfloor$ bits.

For the following discussion we assume that an encoding of the integers has been chosen, and that the number of bits needed to encode the integer i is at most $f(i)$, where $f(i)$ is a concave monotonically increasing function defined on real values of $i \geq 1$. For example, if we choose the second scheme then we can let $f(i) = 1 + \log i + 2\log(1 + \log i)$. We assume that the input stream has been partitioned into a sequence of dictionary words, which we shall call symbols. Let the sequence of symbols be $X = x_1, x_2, \ldots, x_N$. The symbols are taken from a dictionary S of size n. Let $\rho_{\mathrm{MF}}(X, f)$ be the average number of bits per symbol needed to transmit X by the move-to-front scheme using a code with codeword length function f. That is, $\rho_{\mathrm{MF}}(X)$ is the total number of bits needed to transmit the sequence X divided by N. (From now on we omit the reference to f.) Let N_a be the number of occurrences of a symbol a in X. Then we have

THEOREM 1.

$$\rho_{\mathrm{MF}}(X) \leq \sum_{a \in S} \frac{N_a}{N} f\left(\frac{N}{N_a}\right). \tag{6}$$

PROOF.
Let $t_1, t_2, \ldots, t_{N_a}$ be the times when the N_a occurrences of the symbol a are sent. That is, $x_{t_i} = a$ and $t_i < t_{i+1}$. When a occurs at time t_1 its position in the list is at most t_1. Furthermore, when a occurs at time t_i for $i > 1$ its position is at most $t_i - t_{i-1}$. Therefore the cost of transmitting the first a is at most $f(t_1)$, and the cost of transmitting the ith a is at most $f(t_i - t_{i-1})$. If we let $R_a(X)$ be the total number of bits used to transmit the N_a occurrences of symbol a then

$$R_a(X) \leq f(t_1) + \sum_{i=2}^{N_a} f(t_i - t_{i-1}). \tag{7}$$

Noting the concavity of f and applying Jensen's inequality[2] we get

$$R_a(X) \le N_a f\left(\frac{1}{N_a}\left(t_1 + \sum_{i=2}^{N_a}(t_i - t_{i-1})\right)\right)$$
$$= N_a f\left(\frac{t_{N_a}}{N_a}\right) \le N_a f\left(\frac{N}{N_a}\right). \qquad (8)$$

The equality follows from the fact that the terms $t_i - t_{i-1}$ telescope, and the second inequality follows from the fact that f is monotonically increasing. Summing over all $a \in S$ and dividing by N gives Theorem 1. □

By combining Theorem 1 with a particular encoding scheme we can relate the efficiency of the move-to-front compression scheme on a particular sequence to the value of the "empirical entropy" of the sequence. This entropy, H^*, is defined as follows.

$$H^*(X) \equiv \sum_{a \in S} -\frac{N_a}{N} \log \frac{N_a}{N} \qquad (9)$$

COROLLARY 1.

$$\rho_{MF}(X) \le 1 + H^*(X) + 2 \log(1 + H^*(X)). \qquad (10)$$

PROOF.
We use the function f appropriate for the second scheme: $f(i) = 1 + \log i + 2 \log(1 + \log i)$.

Substituting this into Theorem 1 we get:

$$\rho_{Mf}(X) \le \sum_{a \in S} \frac{N_a}{N} + \sum_{a \in S} \frac{N_a}{N} \log \frac{N}{N_a}$$
$$+ \sum_{a \in S} \frac{N_a}{N} 2 \log\left(1 + \log \frac{N}{N_a}\right) \qquad (11)$$

The value of the first sum is 1. The second sum is just $H^*(X)$.

Because log is a concave function and $\Sigma\ (N_a/N) = 1$ we can apply Jensen's inequality to the third summation to bound it by

$$2 \log\left[\sum_{a \in S} \frac{N_a}{N}\left(1 + \log \frac{N}{N_a}\right)\right]. \qquad (12)$$

The summation in (12) is just $1 + H^*(X)$. Combining these results yields the corollary. □

We may now compare the performance of the move-to-front scheme with that of an optimum static prefix code for any particular sequence. One way of getting an optimum code for a particular sequence is to generate an optimum code for a source in which the probability of a symbol a occurring is N_a/N, which is just a Huffman code for this probability distribution. Let $\rho_H(X)$ be the average number of bits per symbol used by this code on the sequence X. A well known fact about an optimum static code is

$$H^*(X) \le \rho_H(X) \le H^*(X) + 1. \qquad (13)$$

(See Gallager [9, ch. 3].)

Substituting the left hand inequality into Corollary 1 gives us inequality (2) in Section 3, namely

$$\rho_{MF}(X) \le 1 + \rho_H(X) + 2 \log(1 + \rho_H(X)).$$

This means that the move-to-front scheme at its worst performs almost as well as a static optimum code, even though it has no advance knowledge of the sequence. Moreover, the move-to-front scheme will do much better than the static optimum on certain types of sequences.

We can also evaluate the average performance of our scheme when compressing a sequence of symbols generated independently according to a fixed distribution. (This is called a discrete memoryless source.)

THEOREM 2.
If the symbols are generated by a discrete memoryless source in which Prob$\{x_1 = a\} = P_a$, *then we have*

$$\langle \rho_{MF}(X) \rangle \le \sum_{a \in S} P_a f\left(\frac{1}{P_a}\right), \qquad (14)$$

where $\langle \cdot \rangle$ *denotes expected value over all sequences of length N.*

PROOF.
Taking expected values on both sides of Theorem 1, we have

$$\langle \rho_{MF}(X) \rangle$$
$$\le \sum_{a \in S} \left\langle \frac{N_a}{N} f\left(\frac{N}{N_a}\right) \right\rangle$$
$$= \sum_{a \in S} \sum_{i=1}^{N} \left(\text{Prob}\{N_a = i\} \frac{i}{N} f\left(\frac{N}{i}\right)\right)$$
$$= \sum_{a \in S} \sum_{i=1}^{N} \left(\binom{N}{i} P_a^i (1 - P_a)^{N-i} \frac{i}{N} f\left(\frac{N}{i}\right)\right) \qquad (15)$$
$$= \sum_{a \in S} P_a \sum_{i=1}^{N} \left(\binom{N}{i} P_a^{i-1} (1 - P_a)^{N-i} \frac{i}{N} f\left(\frac{N}{i}\right)\right).$$

The next step is to pull f out of the inner summation using Jensen's inequality. To do this we must verify that

$$\sum_{i=1}^{N} \binom{N}{i} P_a^{i-1} (1 - P_a)^{N-1} \frac{i}{N} = 1.$$

This follows immediately from the observation that

$$\binom{N}{i} \frac{i}{N} = \binom{N-1}{i-1}$$

and the binomial theorem.

[2] Jensen's inequality states that if f is a concave function, $\{w_i\}$ is a set of positive real weights whose sum is 1, and $\{p_i\}$ is a set of points in the domain of f, then $\sum_i w_i f(p_i) \le f(\sum_i w_i p_i)$ [11].

After applying Jensen's inequality we have

$$\begin{aligned}
&\langle \rho_{\mathrm{MF}}(X) \rangle \\
&\le \sum_{a \in S} P_a f\left(\sum_{i=1}^{N} \binom{N}{i} P_a^{i-1}(1-P_a)^{N-i} \frac{N}{N} \right) \\
&= \sum_{a \in S} P_a f\left(\frac{\left[\sum_{i=0}^{N} \binom{N}{i} P_a^i (1-P_a)^{N-i} \right] - (1-P_a)^N}{P_a} \right) \qquad (16) \\
&= \sum_{a \in S} P_a f\left(\frac{1-(1-P_a)^N}{P_a} \right) \\
&\le \sum_{a \in S} P_a f\left(\frac{1}{P_a} \right)
\end{aligned}$$

by the binomial theorem and the monotonicity of f. □

Note. Theorem 2 holds for all values of $N \ge 1$. □

Using Theorem 2 we can derive a corollary that bounds the expected performance of the move-to-front scheme in terms of the entropy of the source s. This is an expected-case version of Corollary 1. Let $H(s)$ denote the entropy of the source:

$$H(s) \equiv \sum_{a \in S} - P_a \log P_a \qquad (17)$$

COROLLARY 2.

$$\langle \rho_{\mathrm{MF}}(X) \rangle \le 1 + H(s) + 2 \log(1 + H(s)), \qquad (18)$$

where the average is taken over all sequences of length N.

This follows from Theorem 2 in much the same way that Corollary 1 follows from Theorem 1. A substitution is made for f into Theorem 2, then the three summations are bounded using Jensen's inequality.

Corollaries similar to 1 and 2 can be proven for all of the integer prefix codes. The bound achieved in each case is the same as the formula for f with an entropy replacing each log i. In particular, we can derive inequality (1) in Section 3 for the simplest code.

We can use Corollary 2 to prove Shannon's source coding theorem (see Gallager [9]). This theorem says that the number of bits per symbol needed to transmit the information from a discrete memoryless source can be made as close as desired to the entropy of the source. Let X be a sequence generated by a discrete memoryless source s. By grouping the symbols of X in blocks of size k and using the move-to-front scheme of Corollary 2 on these blocks, the average number of bits per block used is bounded by $1 + H(s^k) + 2 \log(1 + H(s^k))$, where $H(s^k) = kH(s)$ is the entropy of the block source s^k. Hence the average number of bits per symbol of X is bounded above by $1/k + H(s) + (2/k)\log(1 + kH(s))$. As k goes to infinity this number approaches the entropy of the source, proving the theorem.

The performance of the intermittent move-to-front scheme is very similar to that of the move-to-front scheme. Assume that an item is moved to the front every τth appearance. Recall that there are $|S| = n$ symbols.

THEOREM 3.

$$\rho_{\mathrm{IMF}}(X) \le \sum_{a \in S} \frac{N_a}{N} f\left(\frac{N + \tau n}{N_a} \right) \qquad (19)$$

PROOF.

Let $t_1, t_2, \ldots, t_{N_a}$ be the times when symbol a occurs in the sequence. For $1 \le i < N_a$, let y_i be the number of times move-to-front is applied between times t_i and t_{i+1} (exclusive). The position of the symbol a at time t_i, $i > \tau$, is at most $1 + \sum_{j=i-\tau}^{i-1} y_j$. The position of a at times $t_1, t_2, \ldots, t_\tau$ is at most n. Therefore the total number of bits used to transmit the N_a occurrences of symbol a is

$$R_a(X) \le \tau f(n) + \sum_{i=\tau+1}^{N_a} f\left(1 + \sum_{j=i-\tau}^{i-1} y_j \right). \qquad (20)$$

The sum of the y_j's is at most $(N - N_a)/\tau$, and each y_j is counted at most τ times. Hence

$$\sum_{i=\tau+1}^{N_a} \left(1 + \sum_{j=i-\tau}^{i-1} y_j \right) \le N. \qquad (21)$$

By Jensen's inequality and the concavity of the function f, we have

$$R_a(X) \le N_a f\left(\frac{N + \tau n}{N_a} \right). \qquad (22)$$

Summing over all $a \in S$ and dividing by N gives the theorem. □

Using the function f for the second integer coding scheme, we obtain

$$\rho_{\mathrm{IMF}}(X) \le 1 + H^*(X) + 2 \log(1 + H^*(X)) + O\left(\frac{\tau n}{N} \right). \qquad (23)$$

The last term is negligible for long sequences.

Similarly, we can prove

THEOREM 4.

If the symbols are generated by a discrete memoryless source in which $Prob\{x_1 = a\} = P_a$, then

$$\langle \rho_{\mathrm{IMF}}(X) \rangle \le \sum_{a \in S} P_a f\left(\frac{1 + \epsilon}{P_a} \right) \qquad (24)$$

where $\epsilon = \tau n / N$.

A direct consequence of Theorem 4 and the second integer coding scheme is

$$\langle \rho_{\mathrm{IMF}}(X) \rangle \le 1 + H(s) + 2\log(1 + H(s)) + O(\tau n/N) \tag{25}$$

where $H(s)$ is the entropy of the discrete memoryless source s.

REFERENCES

1. Bentley, J. L., and McGeoch, C. A. Worst-case analysis of self-organizing sequential search heuristics. In *Proceedings 20th Allerton Conference on Communication, Control, and Computing*. (Monticello, Ill., Oct. 6–8, 1982), Univ. of Illinois, 452–461.
2. Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. A locality adaptive data compression scheme. In *Proceedings 22nd Allerton Conference on Communication, Control, and Computing*. (Monticello, Ill., Oct. 3–5, 1984), Univ. of Illinois, 233–242.
3. Bentley, J. L., and Yao, A. C. An almost optimal algorithm for unbounded searching. *Inform. Process. Lett. 5*, 3 (Aug. 1976), 82–87.
4. Bitner, J. R. Heuristics that dynamically organize data structures. *SIAM J. Comput. 8*, 1 (Feb. 1979), 82–110.
5. Brown, M. R., and Tarjan, R. E. Design and analysis of a data structure for representing sorted lists. *SIAM J. Comput. 9*, 3 (Aug. 1980), 594–614.
6. Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
7. Elias, P. University codeword sets and representation of the integers. *IEEE Trans. Inform. Theory IT-21*, 2 (Mar. 1975), 194–203.
8. Elias, P. Interval and recency-rank source coding: Two on-line adaptive variable-length schemes. *IEEE Trans. Inform. Theory*, submitted for publication (1985).
9. Gallager, R. G. *Information Theory and Reliable Communication*. Wiley, New York, 1968.
10. Gallager, R. G. Variations on a theme by Huffman. *IEEE Trans. Inform. Theory IT-24*, 5 (Nov. 1978), 668–674.
11. Hardy, G. H., Littlewood, J. E., and Polya, G. *Inequalities*. Cambridge Univ. Press, Cambridge, England, 1967.
12. Huddleston, S. An efficient scheme for fast local updates in linear lists. Univ. of California, Irvine, 1981.
13. Huddleston, S., and Mehlhorn, K. A new data structure for representing sorted lists. *Acta Inform. 17*, 2 (June 1982), 157–184.
14. Huffman, D. A. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE 40*, (Sept. 1952), 1098–1101.
15. Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.
16. Knuth, D. E. Dynamic Huffman coding. *J. Algorithms 6*, 2 (June 1985), 163–180.
17. Kosaraju, R. Localized search in sorted lists. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. (San Francisco, Calif., May 5–7, 1982), ACM, 62–69.
18. Maier, D., and Salveter, S. C. Hysterical B-trees. *Inform. Processing Lett. 12*, 4 (Aug. 13, 1981), 199–202.
19. Sleator, D. D., and Tarjan, R. E. Self-adjusting binary trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. (Boston, Mass., Apr. 25–27, 1983), ACM, 235–245.
20. Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update rules. In *Proceedings of the Sixteenth Annual ACM Symposium on the Theory of Computing*. (Washington, D.C., Apr. 30–May 3, 1984), ACM, 488–492.
21. Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Commun. ACM 28*, 2 (Feb. 1985), 202–208.
22. Sleator, D. D., and Tarjan, R. E. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.
23. Tarjan, R. E. *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1983.
24. Tsakalidis, A. K. AVL-trees for localized search. In *Automata, Languages, and Programming, 11th Colloquium*. (Antwerp, Belgium, July 16–20, 1984); also *Lecture Notes in Computer Science 172*. G. Goos and J. Hartmanis, Eds., Springer-Verlag, Berlin, 1984, pp. 473–485.
25. Vitter, J. S. Design and analysis of dynamic Huffman coding. In *Proceedings of the Twenty-Sixth Annual IEEE Symposium on Foundations of Computer Science*. (Portland, Oreg., Oct. 21–23, 1985), IEEE Computer Society, 293–302.
26. Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory IT-24*, 5 (Sept. 1978), 530–536.

CR Categories and Subject Descriptors: E.4 [**Data**]: Coding and Information Theory—*data compaction and compression*
General Terms: Algorithms, Theory
Additional Key Words and Phrases: adaptive coding, move-to-front list rearrangement

Received 8/85; revised 12/85; accepted 12/85

Authors' Present Addresses: Jon Louis Bentley, and Daniel D. Sleator, AT&T Bell Laboratories, Murray Hill, NJ 07974. Robert E. Tarjan, Computer Science Department, Princeton University, Princeton, NJ 08544; and AT&T Bell Laboratories, Murray Hill, NJ 07974. Victor K. Wei, Bell Communications Research, Murray Hill, NJ 07974.