Edgar H. Sibley
Panel Chair

The splay-prefix algorithm is one of the simplest and fastest adaptive data compression algorithms based on the use of a prefix code. The data structures used in the splay-prefix algorithm can also be applied to arithmetic data compression. Applications of these algorithms to encryption and image processing are suggested.

APPLICATION OF SPLAY TREES TO DATA COMPRESSION

DOUGLAS W. JONES

Data compression algorithms can improve the efficiency with which data is stored or transmitted by reducing the amount of redundant data. A compression algorithm takes a source text as input and produces a corresponding compressed text, while an expansion algorithm takes the compressed text as input and produces the original source text as output.[1] Most compression algorithms view the source text as consisting of a sequence of letters selected from an alphabet.

The redundancy of a representation of a string S is $L(S)$–$H(S)$, where $L(S)$ is the length of the representation, in bits, and $H(S)$ is its entropy—a measure of information content, also expressed in bits. No compression algorithm can compress a string to fewer bits than its entropy without information loss. If the source text is drawn one letter at a time from a random source using alphabet A, the entropy is given by:

$$H(S) = C(S) \sum_{c \in A} p(c) \log_2 \frac{1}{p(c)}$$

where $C(S)$ is the number of letters in the string, and $p(c)$ is the static probability of obtaining any particular letter c. If the frequency of each letter c in the string S is used as an estimate of $p(c)$, $H(S)$ is called the self-entropy of S. In this paper, $H_s(S)$ will be used to signify the self-entropy of a string, computed under the assumption that it was produced by a static source.

[1] Such algorithms are *noiseless*; in this paper, approximate or *noisy* algorithms will not be considered.

Static probability models do not provide very good characterizations of many sources. For example, in English text, the letter u is less common than e, so a static probability model would incorrectly predict that qe would be more common than qu. Markov probability models allow very good characterization of such sources. A Markov source has many states, and undergoes a random state change as each letter is drawn. Each state is associated with a probability distribution that determines the next state and the next letter produced. When a Markov source producing English-like text emits a q, it would enter a state in which u is the most likely output. Further discussion of entropy, static sources, and Markov sources can be found in most books on information theory [2].

Although there are a number of ad hoc approaches to data compression, for example, run-length encoding, there are also a number of systematic approaches. Huffman codes are among the oldest of the systematic approaches to data compression. Adaptive Huffman compression algorithms require the use of tree balancing schemes which can also be applied to the data structures required by adaptive arithmetic compression algorithms. There is sufficient similarity between the balancing objectives of these schemes and those achieved by splay trees to try splay trees in both contexts.

Splay trees are usually considered forms of lexicographically ordered binary search trees, but the trees used in data compression need not have a static order. The removal of the ordering constraint allows the basic splaying operation to be considerably simplified. The

resulting algorithms are extremely fast and compact. When applied to Huffman codes, splaying leads to a locally adaptive compression algorithm that is remarkably simple as well as fast, although it does not achieve optimal compression. When applied to arithmetic codes, the result is near optimal in compression and asymptotically optimal in time.

PREFIX CODES

The most widely studied data compression algorithms are probably those based on Huffman codes. In a Huffman code, each source letter is represented in the compressed text by a variable length code. Common source letters are represented by short codes, while uncommon ones are represented by long codes. The codes used in the compressed text must obey the prefix property, that is, a code used in the compressed text may not be a prefix of any other code.

Prefix codes may be thought of as trees, with each leaf of the tree associated with one letter in the source alphabet. Figure 1 illustrates a prefix code tree for a 4 letter alphabet. The prefix code for a letter can be read by following the path from the root of the tree to the letter and associating a 0 with each left branch followed and a 1 with each right branch followed. The code tree for a Huffman code is a weight balanced tree, where each leaf is weighted with the letter frequency and internal nodes have no intrinsic weight. The example tree would be optimal if the frequencies of the letters A, B, C, and D were 0.125, 0.125, 0.25, and 0.5, respectively.

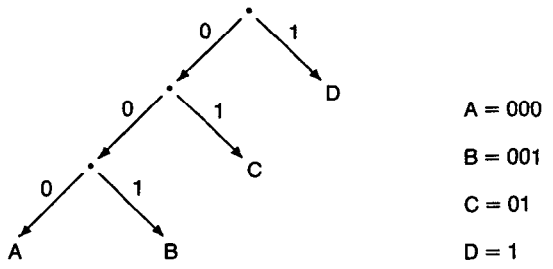

FIGURE 1. A Tree Representation of a Prefix Code

Conventional Huffman codes require either prior knowledge of the letter frequencies or two passes through the data to be compressed—one to obtain the letter frequencies, and one to perform the actual compression. In the latter case, the letter frequencies must be included with the compressed text in order to allow for later expansion. Adaptive compression algorithms operate in one pass. In adaptive Huffman codes, the code used for each letter in the source text being compressed is based on the frequencies of all letters up to but not including that letter. The basis for efficient implementation of adaptive Huffman codes was established by Gallager [3]; Knuth published a practical version of an adaptive algorithm [5]; and Vitter has developed an optimal adaptive Huffman algorithm [10].

Vitter's optimal adaptive Huffman code always comes within one bit per source letter of the optimal static Huffman code, and it is usually within a few percent of H_s. Furthermore, static Huffman codes are always within one bit per source letter of H_s (Huffman codes achieve this limit only when, for all letters, $p(c) = 2^{-i}$). These same bounds can be applied to Markov sources if a different (static or adaptive) Huffman tree is used for each source state inferred from the source text. There are compression algorithms that can improve on these bounds. The Ziv–Lempel algorithm, for example, assigns fixed length words in the compressed text to varying length strings from the source [11], while arithmetic compression can, in effect, utilize fractional bits in the encoding of source letters [12].

Applying Splaying to Prefix Codes

Splay trees were first described in 1983 [8], and more details were presented in 1985 [9]. Splay trees were originally intended as a form of self-balancing binary search trees, but they have also been shown to be among the fastest known priority queue implementations [4]. When a node in a splay tree is accessed, the tree is splayed; that is, the accessed node becomes the root, and all nodes to the left of it form a new left subtree, while all nodes to the right form a new right subtree. Splaying is accomplished by following the path from the old root to the target node, making only local changes along the way, so the cost of splaying is proportional to the length of the path followed.

Tarjan and Sleator [9] showed that splay trees are statically optimal. In other words, if the keys of the nodes to be accessed are drawn from a static probability distribution, the access speeds of a splay tree and of a statically balanced tree optimized for that distribution should differ by a constant factor when amortized over a sufficiently long series of accesses. Since a Huffman tree is an example of a statically balanced tree, this suggests that splaying should be applicable to data compression, and that the compressed code resulting from a splayed prefix code tree should be within a constant factor of the size achievable by using a Huffman code.

As originally described, splaying applies to trees where data is stored in the internal nodes, not the leaves. Prefix code trees carry all of their data in the leaves, with nothing in the internal nodes. There is a variant of splaying, however, called semi-splaying, which is applicable to prefix code trees. In semi-splaying, the target node is not moved to the root, nor are its children modified; instead, the path from the root to the target is simply shortened by a factor of two. Semi-splaying has been shown to achieve the same theoretical bounds as splaying, within a constant factor.

Both splaying and semi-splaying are complicated in a lexicographic tree when a zig-zag path is followed in the interior of the tree, but they are easy when the path to the target node stays entirely on the left or right edge

of the tree (called the zig-zig case in [8] and [9]). This simple case is illustrated in Figure 2. The effect of semi-splaying along the path from the root (node w) to leaf node A is to rotate each successive pair of internal nodes so that the path length from the root to the leaf node is halved. In the process, the nodes in each pair that were farthest from the root stay on the new path (nodes x and z), while those that were closest move off the path (nodes w and y).

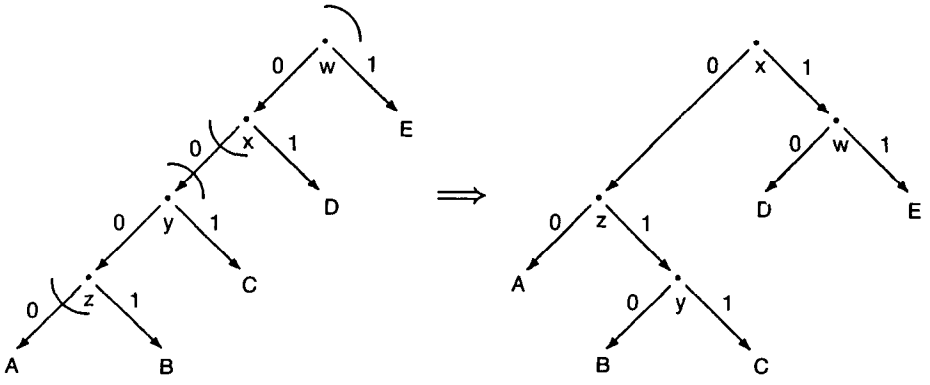

FIGURE 2. Semi-Splaying Around the Leftmost Leaf in a Code Tree

While the semi-splaying operation preserves the lexicographic ordering of all nodes in the tree, this is not important in a prefix code tree. With prefix codes, all that matters is that the tree used by the compression routine to compress any letter of the source text exactly match the tree used by the expansion routine to expand that letter. Any transformation of the tree is allowed between successive letters, as long as both routines perform the same transformations in the same order.

The lack of a lexicographic ordering constraint allows a great simplification to the semi-splaying operation by eliminating the need to consider the zig-zag case. This can be done by inspecting the nodes on the path from the root to the target leaf and exchanging those which are right children with their siblings. This will be called *twisting* the tree. After this modification, the new prefix code for the target leaf is all zeros and the target leaf is the leftmost leaf. In Figure 3, the tree has been twisted to allow easy semi-splaying around leaf C. Fortunately, this change does not disturb any of the performance bounds that have been proven for semi-splaying. The proof of this follows trivially since the potential function used in [9] to prove these performance bounds does not depend on the order of the subtrees of a node.

A second simplification arises when we consider that not only can left and right siblings be exchanged at will, but all internal nodes in the prefix code tree are anonymous and carry no information. This allows the rotations used in semi-splaying to be replaced by operations requiring the exchange of only two links in the tree; we will call these operations *semi-rotations*. Figure 4 shows a semi-rotation. A semi-rotation has the same effect on the distances of each leaf from the root as a full rotation, but it destroys the lexicographic ordering and involves cutting and grafting only 2 branches of the tree, while a full rotation involves cuts and grafts on 4 branches.

There is actually no need to twist the tree prior to applying semi-rotations. Instead, the semi-rotations can be applied along the path from the root to the target leaf as if that path were the left-most path. For exam-

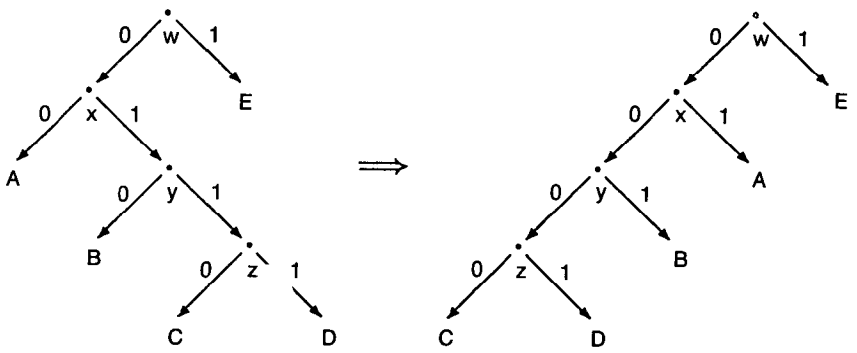

FIGURE 3. Twisting About C to Eliminate the Need for Zig-Zag Semi-Splaying

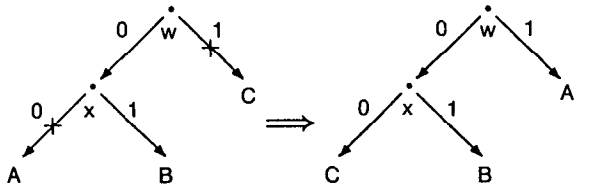

FIGURE 4. Semi-Rotation About A

ple, the tree shown in Figure 3 can be directly splayed as shown in Figure 5. In this example, the tree is semi-splayed around leaf C using semi-rotations at each pair of internal nodes along the path from C to the root. Note that the effect of this transformation on the tree puts each leaf node at the same distance from the root as it would have been if the tree was first twisted so that C was the left-most leaf, and then semi-splayed in the conventional way. The trees that result differ only in the labeling of their internal nodes and in the exchange of children of some of the internal nodes.

It should be noted that there are two ways to semi-splay a tree around a node; these differ when the path from that node to the root has an odd length. In this case, a node on the path cannot be paired to participate in a rotation or semi-rotation. If pairs are constructed from the bottom up, the root may be left out, while, if constructed from the top down, the last internal node on the path may be left out. The presentation given here will focus on the bottom-up approach.

access to two children and the parent of that node. The easiest way to allow for this is to use three pointers per node, a left pointer, a right pointer, and an up pointer. The triangular representation discussed in [9] was used with only two pointers per node,[2] but the savings in storage would be offset by an increase in run-time and code complexity. The basic data structures required are:

```
const
  maxchar = ...   {maximum source character code};
  succmax = maxchar + 1;
  twicemax = 2 × maxchar + 1;
  root = 1;
type
  codetype = 0 .. maxchar   {source character code range};
  bit = 0 .. 1;
  upindex = 1 .. maxchar;
  downindex = 1 .. twicemax;
var
  left, right: array [upindex] of downindex;
  up: array [downindex] of upindex;
```

The index types *upindex* and *downindex* are used for pointers up and down in the code tree. Down pointers must be able to point to either leaves or internal nodes, while up pointers only reference internal nodes. Internal nodes will be stored below leaves, so index values between 1 and *maxchar* (inclusive) will be used to reference internal nodes, while index values between *maxchar* + 1 and (2 × *maxchar*) + 1 (inclusive) will be

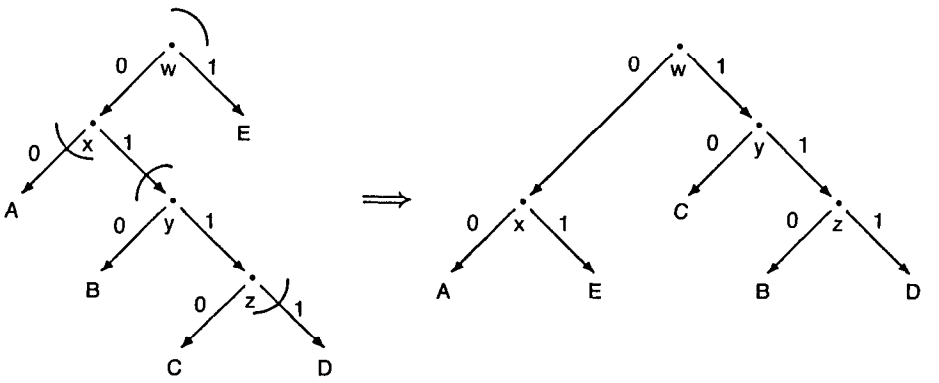

FIGURE 5. Semi-Splaying About C Using Semi-Rotations

The Splay-Prefix Algorithm

The code presented here will be in the style of Pascal, with constant valued expressions substituted for constants where that improves readability. The data structures required by this code will be constructed using only arrays, even though the logical structure might be more clearly expressed using records and pointers. This is in keeping with the form of presentation used in earlier work in this area [5, 10]. It allows easy expression in older but widely used languages such as Fortran, and it allows compact pointer representations.

Each internal node in the code tree must allow used to reference leaves. Note that the root of the tree always has an undefined parent, and is always stored at location 1. The letter corresponding to a leaf can be computed from the index of that leaf by subtracting *maxchar* + 1.

If the end of a source document can be inferred from context, the source alphabet can be encoded directly in the range *codetype*, and the largest code allowed in a

[2] In [9], an extra bit was needed per node in the triangular representation to distinguish left only children from right only children; since a prefix code tree is a complete binary tree, this bit is not needed here.

source document can be *maxchar*. If this is not the case, the range *codetype* must be expanded by one to include a special *end-of-file* character; this means that *maxchar* will be one greater than the largest character representation.

The following routine will initialize the code tree. This builds a balanced code tree, but in fact, any initial tree would suffice as long as the same initial tree is used for both compression and expansion.

```
procedure initialize;
var
  i: downindex;
  j: upindex;
begin
  for i := 2 to twicemax
  do up[i] := i div 2;
  for j := 1 to maxchar do begin
      left[j] := 2 × j;
      right[j] := 2 × j + 1;
  end;
end  {initialize};
```

After each letter is compressed or expanded, using the current version of the code tree, the tree must be splayed around the code for that letter. The following procedure does this, using bottom-up splaying.

```
procedure splay(plain: codetype):
var
  a, b: downindex   {children of nodes to semi-rotate};
  c, d: upindex     {pair of nodes to semi-rotate};
begin
  a := plain + succmax;
  repeat   {walk up the tree semi-rotating pairs}
    c := up[a];
    if c ≠ root then begin   {a pair remains}
      d := up[c];
      {exchange children of pair}
      b := left[d];
      if c = b then begin
        b := right[d];
        right[d] := a;
      end else begin
        left[d] := a;
      end;
      if a = left[c] then begin
        left[c] := b;
      end else begin
        right[c] := b;
      end;
        up[a] := d;
        up[b] := c;
        a := d;
      end else begin   {handle odd node at end}
        a := c;
      end;
    until a = root;
  end  {splay};
```

To compress a letter from the source text, the letter must be encoded using the code tree, and then transmitted. Since encoding is done by following a path from a leaf to the root of the tree, the code bits are produced in the reverse order from the order in which they must be transmitted. To correct this, the *compress* routine uses a local stack from which bits are popped one at a time and passed to the *transmit* routine.

```
procedure compress(plain: codetype);
var
  sp: 1 . . succmax;
  stack: array[upindex] of bit;
  a: downindex;
begin
  {encode}
  a := plain + succmax;
  sp := 1;
  repeat   {walk up the tree pushing bits}
    stack[sp] := ord(right[up[a]] = a);
    sp := sp + 1;
    a := up[a];
  until a = root;
  repeat   {transmit}
    sp := sp − 1;
    transmit(stack[sp]);
  until sp = 1;
  splay(plain);
end  {compress};
```

To expand a letter, successive bits must be read from the compressed text using the *receive* function. Each bit determines one step on the path from the root of the tree to the leaf representing the expanded letter.

```
function expand: codetype;
var
  a: downindex;
begin
  a := root;
  repeat   {once for each bit on the path}
    if receive = 0
      then a := left[a]
      else a := right[a];
  until a > maxchar;
  splay(a − succmax);
  uncompress := a − succmax;
end  {expand};
```

The main programs for compression and expansion are trivial, consisting of a call to the *initialize* routine, followed by successive calls to *compress* or *expand* for each letter processed.

Performance of the Splay-Prefix Algorithm

In practice, splay-tree based prefix codes are not optimal, but they have some useful properties. Primary among these are speed, simple code, and compact data structures. The splay-prefix algorithm requires only 3 arrays, while Vitter's Algorithm Λ for computing an optimal adaptive prefix code requires 11 arrays [10]. Assuming that the source character set uses 8 bits per character and that end-of-file must be signalled by a

character outside the 8 bit range, *maxchar* = 256 and all array entries can be directly represented in either 9 or 10 bits (two bytes on most machines).[3] The static storage requirements for the splay-prefix algorithm are about 9.7k bits (or 2k bytes on most machines). A similar approach to storing the arrays used by Algorithm Λ requires about 57k bits (or 10k bytes on most machines).

Other commonly used compression algorithms require even more memory; for example, Welch recommends using a 4096 entry hash table with 20 bits per entry to implement Ziv–Lempel compression [11], for a total of almost 82k bits (or 12k bytes on most machines). The widely used compress command on Berkeley UNIX systems uses a Ziv–Lempel code based on a table of up to 64k entries of at least 24 bits each, for a total of 1572k bits (196k bytes on most machines).

Table I shows how Vitter's Algorithm Λ and the splay-prefix algorithm performed when used on a variety of test data. In all cases, an alphabet of 256 distinct letters was used, augmented with a reserved end-of-file mark. For all files, compressed output of Algorithm Λ was within 5 percent of H_s, and was usually within 2 percent. For all files, the compressed output of the splay algorithm was never more than 20 percent larger than H_s, and was sometimes much smaller.

The test data includes a C program (file 1), two Pascal programs (files 2, 3), and an early draft of this text (file 4). All 4 text files use the ASCII character set, with tabs replacing most groups of 8 leading blanks, and few if any trailing blanks. For all of these files, Algorithm Λ produced results that were about 60 percent of the original size, and the splay algorithm produced results that were about 70 percent of the original size. This was the worst compression performance observed for the splay-prefix algorithm relative to Algorithm Λ.

Two M68000 object files were compressed (files 5, 6), as well as a file of TEX output in DVI format (file 7). These files have less redundancy than the text files, and thus, neither compression method was able to reduce their size as effectively. Nonetheless, both compression methods managed to usefully compress the data, and the splay algorithm produced results that were about 10 percent larger than those produced by Algorithm Λ.

Three digitized images of human faces were compressed (files 8, 9, 10); these have varying numbers of pixels, but all were digitized using 16 grey levels, and stored one pixel per byte. For these files, Algorithm Λ produced results that were about 40 percent of the original size, while the splay-prefix algorithm produced results only 25 percent of the original size, or about 60 percent of H_s. At first, this may appear to be impossible, since H_s is an information theoretic limit, but the splay-prefix algorithm passes this limit by exploiting the Markov characteristics of some sources.

The final 3 files were artificially created to explore the class of sources where the splay-prefix algorithm excels (files 11, 12, 13); all contain equal numbers of each of the 256 character codes, so H_s is the same for all three, and is equal to the length of the string in bits. In file 11, the entire character set is repeated 64 times; the splay-prefix algorithm performed marginally better than H_s. In file 12, the character set is repeated 64 times but the bits of each character are reversed; this prevents splaying from improving on H_s. The key difference between these two is that in file 11, successive characters are likely to come from the same subtree of the code tree, while in file 12, this is unlikely. In file 13, the character set is repeated 7 times, but in each copy of the character set after the second, each character is repeated twice as many times as in the previous copy; the file ends with a run of 32 *a*'s followed by a run of 32 *b*'s, and so forth. Here, the splay-prefix algorithm was able to exploit long runs of repeated characters, so the result was only 25 percent of H_s; on the other hand, algorithm Λ never found any character to be more than twice as common as any other, so equal length codes were used throughout.

When a character is repeated, the splay-prefix algorithm assigns successively shorter codes to each repetition; after at most $\log_2 n$ repetitions of a letter from an

[3] Changes to the coding standards allowing array indices to run from 0 to 255 instead of 1 to 256 would reduce the storage requirements of both the splay-prefix algorithm and Algorithm Λ.

TABLE 1. Results for Algorithm Λ and the Splay-Prefix Algorithm

file	type	bytes	bits	H_s	Λ bits	splay bits
1	C	12090	96720	58880.2	60176	66344
2	Pascal	3632	29056	16882.0	17544	19608
3	Pascal	9720	77760	45788.6	46704	53552
4	text	55131	441048	270814.9	274032	309496
5	object	32207	257656	193665.3	196760	206280
6	object	41456	331648	249270.7	252312	263744
7	.DVI	41881	335048	257542.3	260592	282304
8		46187	369496	147296.7	149056	94936
9	images	60141	481128	183023.7	186032	115576
10		144981	1159848	506817.1	515304	262376
11		16385	131080	131080.2	132552	122296
12	test	16385	131080	131080.2	132592	144544
13		16385	131080	131080.2	132552	32424

n letter alphabet, splaying will assign a 1 bit code to that letter. This explains the excellent results of splaying applied to file 13. Furthermore, if letters from one subtree of the code tree are repeatedly referenced, splaying will shorten the codes for all letters in that subtree. This explains why splaying performed well when applied to file 11.

In the image data, it was rare for more than a few consecutive pixels of any scan line to have the same intensity, but within each textured region of the image, a different static probability distribution could be used to describe the distribution of intensities. As the splay-prefix algorithm compresses successive pixels in a scan line, it assigns short codes to the pixel intensities which are common in the current context. When it crosses from one textured region to another, short codes are quickly assigned to intensities common in the new region, while the codes for now-unused intensities slowly grow longer. As a result of this behavior, the splay-prefix algorithm is *locally adaptive*. The splay-prefix algorithm and the similar locally adaptive algorithms should be able to achieve reasonable compression results for any Markov source that stays in each state long enough for the algorithm to adapt to that state.

Other locally adaptive data compression algorithms have been proposed by Knuth [5] and by Bentley, et al. [1]. Knuth proposed a locally adaptive Huffman algorithm where the code used for any letter was determined by the n most recent letters; this approach is computationally slightly more difficult than simple adaptive Huffman algorithms, but the appropriate value of n depends on the frequency of state changes in the source. Bentley, et al. propose using the move-to-front heuristic to organize a list of recently used words (assuming that the source text has lexical structure) in conjunction with a locally adaptive Huffman code for encoding slot numbers in the list. This locally adaptive Huffman code involves periodically reducing the weights on all letters in the Huffman tree by multiplying by a constant less than one. A similar approach is used [12] in the context of arithmetic codes. In many respects, the periodic reduction of the weights of all letters in an adaptive Huffman or arithmetic code should result in adaptive behavior very similar to that of the splay compression algorithm described here.

The small data structures required by the splay-prefix algorithm allow Markov models to be constructed with a relatively large number of states; for example, models with more than 90 states can be represented in the 196k byte space used by the compress command under Berkeley UNIX. Furthermore, the code presented here can be converted to a Markov model by adding one variable, *state*, and by adding a state dimension to each of the 3 arrays representing the code tree. The code trees for all of the states can be identically initialized, and one statement needs to be added at the end of the splay routine to change the state based on the previous letter (or in more complex models, on the previous letter and the previous state).

For a system with n states, and where the previous letter was c, it is easy to use the value c mod n to determine the next state. This Markov model blindly lumps every nth letter in the alphabet into one state. Values of n varying from 1 to 64 were tried in compressing a text file, an object code file, and a digitized image (file 8). The results of these experiments are presented in Figure 6. For object code, a 64 state model was sufficient to outperform the Ziv–Lempel based compress command and a 4 state model was sufficient to pass H_s. For the text file, a 64 state model came close to the performance of the compress command, and an 8 state model was sufficient to pass H_s. For the image data (file 8), a 16 state model was sufficient to outperform the compress command and all models significantly outperformed H_s. Markov models with fewer than 8 states were less effective than a simple static model applied to the image data, with the worst case being 3 states. This is because the use of a Markov model interferes with the locally adaptive behavior of the splay-prefix algorithm.

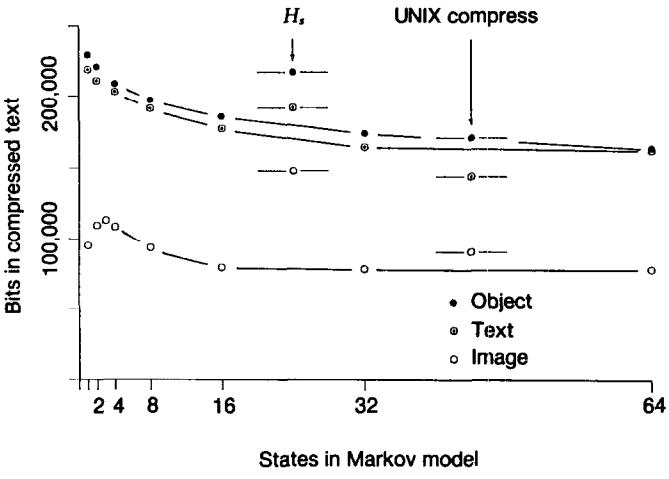

FIGURE 6. Performance of the Splay-Prefix Algorithm with a Markov Model

Both Algorithm Λ and the splay-prefix algorithm have run-times proportional to the size of the output, and in both cases, the output is of worst-case length $O(H_s)$; thus both should run in worst-case time $O(H_s)$. The constant factors differ because the splay-prefix algorithm performs less work per bit of output, but produces more bits of output in the worst case. For the 13 files in Table I, Algorithm Λ produced output at an average rate of 2k bits per second, while the splay-prefix algorithm produced output at better than 4k bits per second; thus, the splay algorithm was always significantly faster. These times were measured on an M68010 based Hewlett Packard Series 200 9836CU workstation under the HP-UX operating system, with both algorithms written in Pascal to similar coding standards.

ARITHMETIC CODES

The compressed text resulting from arithmetic data compression is viewed as a binary fraction, and each letter in the alphabet is associated with a different subrange of the half open interval [0, 1). The source text can be viewed as a textual representation of this fraction using a number system where each letter in the alphabet is used as a digit, but the range of values associated with each letter has a width depending on the frequency of that letter. The first letter of the compressed text (the most significant "digit") can be decoded by finding the letter associated with the subrange bounding the fraction that represents the text. After determining each letter of the source text, the fraction can be rescaled to remove that letter; this is done by subtracting the base of the letter's subrange and dividing by the width of the subrange. Once this is done, the next letter can be decoded.

As an example of an arithmetic code, consider the 4 letter alphabet (A, B, C, D) with the probabilities (0.125, 0.125, 0.25, 0.5). The interval [0, 1) could be subdivided as follows:

$$A = [0, 0.125), \quad B = [0.125, 0.25),$$

$$C = [0.25, 0.5), \quad D = [0.5, 1)$$

This subdivision is easily derived from the cumulative probabilities of each letter and its predecessors in the alphabet. Given the compressed text 0.6 (represented as a decimal fraction), the first letter must be D because it is in the range [0.5, 1). Rescaling gives:

$$(0.6 - 0.5)/0.5 = 0.2$$

Thus, the second letter must be B because it is in the range [0.125, 0.25). Rescaling gives:

$$(0.2 - 0.125)/0.125 = 0.6$$

This implies that the third letter is D, and that, lacking any information about the length of the message, it could be the repeating string DBDBDB . . . .

The primary problem with arithmetic codes is the high precision arithmetic required by interpreting the entire bit pattern that represents the compressed text as a number. This problem was solved in 1979 [6]. The compression efficiency of a static arithmetic code will equal H_s only if infinite precision arithmetic is used. The finite precision of most machines, however, is sufficient to allow extremely good compression. Integer variables 16 bits long, with 32 bit products and dividends, are sufficient to allow adaptive arithmetic compression to within a few percent of the limit, and the result is almost always slightly better than Vitter's optimal adaptive Huffman code.

As with Huffman codes, static arithmetic codes require either two passes or prior knowledge of the letter frequencies. Adaptive arithmetic codes require an efficient algorithm for maintaining and updating the running frequency and cumulative frequency information as letters are processed. The simplest way of doing this is to associate a counter with each letter that is incremented each time the letter or any of its successors in the alphabet are encountered. With this approach, the frequency of a letter is the difference between its counter and its predecessor's counter. This simple approach can take $O(n)$ time to process a letter from an n letter alphabet. In Witten, Neal and Cleary's C implementation of an arithmetic data compression algorithm [12], the average performance was improved by using a move-to-front organization, thus reducing the number of counters that must be updated each time a letter is processed.

Further improvement in the worst-case performance for updating the cumulative frequency distribution requires a radical departure from the simple data structures used in [12]. The requirements that this data structure must meet are best examined by expressing it as an abstract data type with the following five operations: *initialize*, *update*, *findletter*, *findrange*, and *maxrange*. The *initialize* operation sets the frequency of all letters to one; any nonzero value would do, as long as the encode and decode algorithms use the same initial frequencies. An initial frequency of zero would assign an empty range to a character, thus preventing it from being transmitted or received.

The *update*(*c*) operation increments the frequency of the letter *c*. The *findletter* and *findrange* functions are inverses, and *update* may perform any reordering of the alphabet as long as it maintains this inverse relationship. At any point in time, *findletter*(*f*, *c*, *min*, *max*) will return the letter *c* and the associated cumulative frequency range [*min*, *max*), where this range contains *f*. The inverse function, *findrange*(*c*, *min*, *max*) will return the values for *min* and *max* when given the letter *c*. The *maxrange* function returns the sum of the frequencies of all letters in the alphabet, and is needed to scale the cumulative frequencies into the interval [0, 1).

Applying Splaying to Arithmetic Codes

The key to implementation of the cumulative frequency data structures, with worst case behavior better than $O(n)$ per operation on an n letter alphabet, is to organize the letters of the alphabet as leaves in a tree. Each leaf in this tree can be weighted with the frequency of the corresponding letters, and each internal node can be weighted with the sum of the weights of all children. Figure 7 illustrates such a tree for the

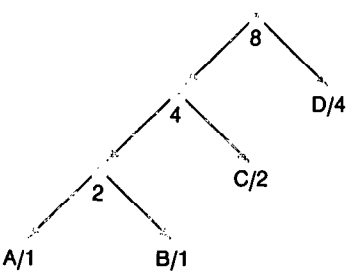

FIGURE 7. A Cumulative Frequency Tree

4 letter alphabet (A, B, C, D) with the probabilities (0.125, 0.125, 0.25, 0.5) and the frequencies (1, 1, 2, 4). The *maxrange* function is trivial to compute on such a tree; it simply returns the weight on the root. The *update* and *findrange* functions can be computed by traversing a path in the tree from a leaf to the root, and the *findletter* function can be computed by traversing a path from the root to a leaf.

The data structures for representing the cumulative frequency tree are essentially the same as those already presented for representing a prefix code tree, with the addition of an array to hold the frequency of each node in the structure:

```
const
  maxchar := ...   {maximum source character code};
  succmax = maxchar + 1;
  twicemax = 2 × maxchar + 1;
  root = 1;
type
  codetype = 0 .. maxchar   {source character code range};
  bit = 0 .. 1;
  upindex = 1 .. maxchar;
  downindex = 1 .. twicemax;
var
  left, right: array [upindex] of downindex;
  up: array [downindex] of upindex;
  freq: array [downindex] of integer;
```

Initialization of this structure involves not only building the tree data structure, but initializing the frequencies of each leaf and internal node as follows:

```
procedure initialize;
var
  d: downindex;
  u: upindex;
begin
  for d := succmax to twicemax do freq[d] := 1;
  for u := maxchar downto 1 do begin
    left[u] := 2 × u;
    right[u] := (2 × u) + 1;
    freq[u] := freq[left[u]] + freq[right[u]];
    up[left[u]] := u;
    up[right[u]] := u;
  end;
end   {initialize};
```

To find a letter and its cumulative frequency range when given a particular cumulative frequency, the tree must be entered at the root and traversed towards that letter, keeping a running account of the frequency range represented by the current branch of the tree. The range associated with the root is [0, *freq*[*root*]), which must contain *f*. When at a particular node *i* in the tree associated with range [*a*, *b*), where $a - b = freq[i]$, the ranges associated with the two subtrees will be [*a*, $a + freq[left[i]]$) and [$a + freq[left[i]]$, *b*); these subranges are disjoint and the path down the tree will be such that *f* is contained in the subranges associated with each node on the path. This leads to the following code:

```
procedure findsymbol(f: integer; var c: codetype;
                     var a, b: integer);
var
  i: downindex;
  t: integer;
begin
  i := root;
  a := 0;
  b := freq[root];
  repeat
    t := a + freq[left[i]];
    if f < t then begin   {left turn}
      i := left[i];
      b := t;
    end else begin   {right turn}
      i := right[i];
      a := t;
    end;
  until i > maxchar;
  c := i - succmax;
end   {findsymbol}
```

To find the cumulative frequency range associated with a letter, the process illustrated for *findsymbol* must be reversed. Initially, the only information known about the letter at node *i* in the tree is the frequency of that letter, *freq*(*i*). From this, the range [0, *freq*(*i*)) can be inferred; this would be the range associated with the letter if it were the only letter in the alphabet. Given that the range [*a*, *b*) is associated with some leaf in the context of the subtree rooted at *i*, the range associated with that leaf in the context of the *up*[*i*] can be computed. If *i* is a left child, this is simply [*a*, *b*); if *i* is a right child, this is [$a + d$, $b + d$), where $d = freq[up[i]] - freq[i]$, or equivalently, $d = freq[left[up[i]]]$. This leads to the following:

```
procedure findrange(c: codetype; var a, b: integer);
var
  i: downindex;
  d: integer;
begin
  i := c + succmax;
  a := 0;
  b := freq[i];
  repeat
    if right[up[i]] = i then begin   {i is right child}
      d := freq[left[up[i]]];
      a := a + d;
      b := b + d;
    end;
    i := up[i];
  until i = root;
end   {findrange};
```

If not for the problem of maintaining appropriate balance in the cumulative frequency tree, the update

function would be trivial; consisting of a walk from the leaf being updated to the root, incrementing the weights of each node visited. If this were done, starting with an initially balanced tree, the time per *findletter*, *findrange*, or *update* operation would be $O(\log_2 n)$ for an n letter alphabet. This is better than the worst case $O(n)$ achieved by a linear data structure (with or without a move-to-front organization), but it can be improved.

Note that each letter compressed by the arithmetic data compression algorithm requires a call to *findrange* followed by a call to *update*, and that each letter expanded requires a call to *findletter* followed by a call to *update*. Thus, the path from the root to a letter in the cumulative frequency tree will be traversed twice during compression, and twice during expansion. Minimizing the total time taken to compress or expand a message requires minimizing the total length of all paths followed in the tree. If the letter frequencies are known in advance, a static Huffman tree will minimize this path length! The path length for message S will be bounded by $2(H_s(S) + C(S))$, where $C(S)$ is the number of letters in the string and the factor of 2 is due to the fact that each path is followed twice.

There is no point in using a cumulative frequency tree if all probabilities are known in advance, since this would allow the use of a simple table lookup to find the probabilities. If they are not known, Vitter's optimal Algorithm Λ could easily be modified to manage the cumulative frequency tree to obtain a bound on the path length followed during compression or expansion of $2(H_s(S) + 2C(S))$. Similarly, the splay-prefix algorithm could be used, giving a bound of $O(H_s(S))$ on the path

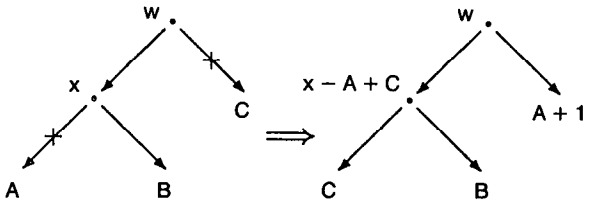

FIGURE 8. Semi-Rotation in a Cumulative Frequency Tree

length, but with larger constant factors. The empirical results presented earlier suggest that these constant factors are more than compensated for by the simplicity of the splay-prefix algorithm.

In the context of the splay-prefix algorithm, the splay operation did not need to manipulate any information in the internal nodes of the tree. When splaying is used as part of the *update* operation, each semi-rotate operation must preserve the invariants governing the weights of nodes in the tree. In Figure 8, the tree is semi-rotated about A; as a result, the weight of x is reduced by the weight of A and increased by the weight of C. At the same time, since this is part of an iterative traversal of the path from A to the root, the weight of A is incremented. The resulting code is shown as:

```
procedure update(c: codetype);
var
  a, b: downindex   {children of nodes to semi-rotate};
  c, d: upindex     {pair of nodes to semi-rotate};
begin
  a := c + succmax;
  repeat   {walk up tree, rotating and incrementing}
    c := up[a];
    if c ≠ root then begin   {a pair remains}
      d := up[c];
      {exchange children of pair}
      b := left[d];
      if c = b then begin
        b := right[d];
        right[d] := a;
      end else begin
        left[d] := a;
      end;
      if a = left[c] then begin
        left[c] := b;
      end else begin
        right[c] := b;
      end;
      up[a] := d;
      up[b] := c;
      freq[c] := (freq[c] − freq[a]) + freq[b];
      freq[a] := freq[a] + 1;
      a := d;
    end else begin {handle odd node at end of path}
      freq[a] := freq[a] + 1;
      a := up[a];
    end;
  until a = root;
  freq[root] := freq[root] + 1;
end   {update};
```

The code ignores the problem of overflow in the frequency counters. Arithmetic data compression repeatedly uses computations of the form $a*b/c$, and as a result, the limit on the precision of computation is set by the storage allowed for intermediate products and dividends, not for integer variables. Many 32 bit machines impose a limit of 32 bits on products and dividends, and thus impose an effective 16 bit limit on the integers a, b, and c in the above expression. When this constraint is propagated through the code for arithmetic data compression, the net effect is a limit of 16383 on the maximum value returned by *maxrange*, or *freq*[*root*]. As a result, unless all files being compressed are shorter than 16383 bytes, all frequencies in the data structure must be periodically rescaled to force them into this range. An easy way to do this is to divide all frequencies by a small constant such as two, and rounding up to prevent any frequencies from dropping to zero.

Leaves in the cumulative frequency tree can be easily rescaled by division by two, but internal nodes are not as easily rescaled because of the difficulty of propagating rounding decisions up the tree. As a result, the easiest thing to do is rebuild the tree, as shown in the following code:

```
procedure rescale;
var
  d: downindex;
  u: upindex;
begin
  for d := succmax to twicemax
    do freq[d] := (freq[d] + 1) div 2;
  for u := maxchar downto 1 do begin
    left[u] := 2 × u;
    right[u] := (2 × u) + 1;
    freq[u] := freq[left[u]] + freq[right[u]];
    up[left[u]] := u;
    up[right[u]] := u;
  end;
end {rescale};
```

Performance of Arithmetic Codes

The above routines were incorporated into a Pascal transliteration of the Witten, Neal and Cleary's algorithm [12]. As expected, there was no significant difference between the compressed text resulting from the original and from the modified arithmetic compression algorithm. Usually, the compressed texts that resulted from the two algorithms were exactly the same length.

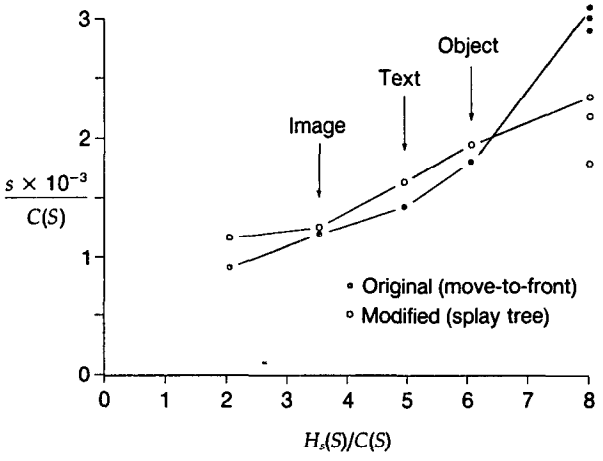

FIGURE 9. Performance of Arithemtic Compression Algorithms

Figure 9 shows the speed of the two arithmetic compression algorithms as a function of H_s. Time is shown in milliseconds per source byte, and entropy is shown in bits per source byte. The files with 2 bits/byte and 8 bits/byte were artificially created; the others were an image file digitized using 16 grey levels (3.49 bits/byte), a text file (4.91 bits/byte), and an M68000 object file (6.02 bits/byte). Time was measured on an HP9836CU workstation under HP-UX.

As shown in Figure 9, splaying applied to a cumulative frequency tree only outperforms the move-to-front algorithm employed by Witten, Neal, and Cleary [12] when the data to be compressed has an entropy of more than about 6.5 bits/byte. Below this, the move-to-front method always slightly outperforms splaying. Thus, splaying or other approaches to balance the cumulative frequency tree are probably not justified for compressing data using a 256 letter alphabet. For larger alphabets, on the other hand, this data suggests that splaying may well be the best approach.

CONCLUSIONS

The splay-prefix algorithm presented here is probably the simplest and fastest adaptive data compression algorithm based on the use of a prefix code. Its outstanding characteristics are a very small storage requirement and locally adaptive behavior. When large amounts of memory are available, use of the splay-prefix algorithm with a Markov model frequently allows more effective data compression than competing algorithms that use the same amount of memory.

The advantages of the splay-prefix algorithm were most effectively demonstrated when it was applied to compressing image data. The locally adaptive character of the algorithm allowed it to compress an image to fewer bits than the self-entropy of the image measured, assuming a static source. Finally, a simple Markov model using the splay-prefix algorithm frequently allowed compression superior to the widely used Ziv–Lempel algorithm which uses a comparable amount of memory.

Arithmetic data compression algorithms can be made to run in $O(H_s)$ time by using a cumulative frequency tree balanced by the splaying heuristic for the statistical model required by the algorithm. This bound is new, but the simple move-to-front heuristic is more effective for the small alphabets typically used.[4]

Both the splay-prefix algorithm and the use of splaying to manage the cumulative frequency tree provide useful illustrations of the utility of splaying to manage trees which are not lexicographically organized. The notion of twisting a tree prior to splaying to eliminate the need for the zig-zag case may be applicable to other nonlexicographic trees, as may the notion of semi-rotation for balancing such a tree. As an example, these techniques should be applicable to merge trees where a binary tree of 2-way merges is used to construct an n-way merge; Saraswat appears to have used similar ideas in developing his Prolog implementation of merge trees [7].

It is interesting to note that, as with other adaptive compression schemes, the loss of one bit from the stream of compressed data is catastrophic! This suggests that it would be interesting to search for ways of recovering from such a loss; yet it also suggests the use of such compression schemes in cryptography. It is well known that compressing a message before it is encrypted increases the difficulty of breaking the code simply because code breaking relies on redundancy in the encrypted text and compression reduces this redundancy. The new possibility, introduced by the compression algorithms described here, is to use the initial state of the prefix code tree or the initial state of the

[4] Alistair Moffat of the University of Melbourne has independently achieved the same performance using a data structure derived from the implicit heap of heapsort.

cumulative frequency tree as a key for direct encryption during compression. The arithmetic compression algorithm could further complicate the work of a code breaker because letter boundaries to not necessarily fall between bits.

The key space for such an encryption algorithm is huge. For an n letter alphabet, there are $n!$ permutations allowed on the leaves of the tree, times C_{n-1} trees with $n-1$ internal nodes, where $C_i = (2i)!/i!\,(i+1)!$, the ith Catalan number. This product simplifies to $(2(n-1))!/(n-1)!$. For $n = 257$ (a 256 letter alphabet augmented with an end-of-file character), this is 512!/256!, or somewhat less than 2^{2200}. A compact integer representation of a key from this space would occupy 675 eight-bit bytes; clearly, such large keys may pose problems. One practical solution would simply involve starting with an initial balanced tree, as in the compression algorithms presented here, and then splaying this tree about each of the letters in a key string provided by the user; most users are unlikely to provide key strings as long as 675 bytes, and it takes keys longer than this to allow splaying to move the tree into all possible configurations, but even short key strings should provide a useful degree of encryption.

REFERENCES

1. Bentley, J.L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr. 1986), 320–330.
2. Gallager, R.G. *Information Theory and Reliable Communication.* John Wiley & Sons, New York, 1968.
3. Gallager, R.G. Variations on a theme by Huffman. *IEEE Trans. Inform. Theory IT-24*, 6 (Nov. 1978), 668–674.
4. Jones, D.W. An empirical comparison of priority queue and event set implementations. *Commun. ACM* 29, 4 (Apr. 1986), 300–311.
5. Knuth, D.E. Dynamic Huffman coding. *J. Algorithms 6*, 2 (Feb. 1985), 163–180.
6. Rubin, F. Arithmetic stream coding using fixed precision registers. *IEEE Trans. Inform. Theory IT-25*, 6 (Nov. 1979), 672–675.
7. Saraswat, V. Merge trees using splaying—or how to splay in parallel and bottom-up. *PROLOG Digest 5*, 22 (Mar. 27, 1987).
8. Sleator, D.D., and Tarjan, R.E. Self-adjusting binary trees. In *Proceedings of the ACM SIGACT Symposium on Theory of Computing* (Boston, Mass., Apr. 25–27). ACM, New York, 1983, pp. 235–245.
9. Tarjan, R.E., and Sleator, D.D. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.
10. Vitter, J.S. Two papers on dynamic Huffman codes. Tech. Rep. CS-85-13. Brown University Computer Science, Providence, R.I. Revised Dec. 1986.
11. Welch, T.A. A technique for high-performance data compression. *IEEE Comput. 17*, 6 (June 1984), 8–19.
12. Witten, I.H., Neal, R.M., and Cleary, J.G. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June 1987), 520–540.

CR Categories and Subject Descriptors: E.1 [**Data Structures**]: *trees*; E.2 [**Data Storage Representation**]: *linked representations*; E.4 [**Coding and Information Theory**]: *data compaction and compression*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Adaptive algorithms, arithmetic codes, data compression, splay trees, prefix codes

Received 11/87; accepted 2/88

Author's Present Address: Douglas W. Jones, Dept. of Computer Science, University of Iowa, Iowa City, IA 52242. Internet address: jones@cs.uiowa.edu