Splay Trees for Data Compression

Dennis Grinberg (*CMU*) *
Sivaramakrishnan Rajagopalan (*Boston U.*) †
Ramarathnam Venkatesan (*Bellcore*) ‡
Victor K. Wei (*Chinese U. Hong Kong*) §

Abstract

We present applications of *splay trees* to two topics in data compression. First is a variant of the move-to-front (mtf) data compression (of Bentley,Sleator Tarjan and Wei) algorithm, where we introduce *secondary list(s)*. This seems to capture higher-order correlations. An implementation of this algorithm with Sleator-Tarjan splay trees runs in time (provably) proportional to the entropy of the input sequence. When tested on some telephony data, compression ratio and run time showed significant improvements over original mtf-algorithm, making it competitive or better than popular programs. For stationary ergodic sources, we analyse the compression and output distribution of the original mtf-algorithm, which suggests why the secondary list is appropriate to introduce. We also derive analytical upper bounds on the average codeword length in terms of stochastic parameters of the source.

Secondly, we consider the compression (or coding) of source sequences where the codewords are required to preserve the alphabetic order of the source symbols. We describe the use of the semisplay tree, the regular splay tree, and splay trees with depth three and four for this application, and derive upper bounds on their compression efficiency. For example, the average codeword length of the semisplay tree is is no more than twice the source entropy plus some minor terms.

1 Introduction.

We study applications of the splay tree idea to two related topics in data compression: the compression of sequences sources with an alphabetic constraint and the compression of certain sources with memory.

Data compression is concerned with the shrinking of files before transmission or storage, in order to conserve communication or computer memory resources. We focus on *lossless* (*reversible, faithful*) compression where the original file can be exactly recovered from the compressed version. The objective of lossless compression is to achieve the smallest compressed file possible.

Huffman tree (or Huffman code) is a classic result in compression. It gives the best code for compressing a source file whose symbols are generated by an independent but identical distribution (i.i.d). If the source distribution is known, Huffman's algorithm [10] generates an optimal set of codewords for compressing this source.

First, we present a variant of the move-to-front algorithm of Bentley, et al. [1]. It maintains an additional list and has improved compression efficiency for certain sources with memory. This two-list variant of the move-to-front algorithm is implemented using splay trees for the lists and tested on a collection of telephony data. In most cases tested, this algorithm compresses as well or better than the popular Lempel-Ziv algorithm versions and its speed of execution is comparable as well. Simulation results are presented. The algorithm runs in time proportional to the entropy H of the source sequence. This turns out to be of importance: the average depth of access in a splay tree provably reduces by a factor of H, essentially making its runtime comptetitive. We also present upper bounds on the average codeword length in terms of stochastic parameters of the source. It would be of interest to see if additional lists help in other problems with move-to-front heuristic yield improvements in performance.

Secondly, we study adaptive data compression where the codewords are required to preserve the alphabetic ordering of their corresponding source symbols. The study of compression with an alphabetic ordering is fundamentally useful. Alphabetic coding tree minimizes encoding delays. Also in many applications of coding and searching, as well as data compression, the preservation of alphabetic ordering is imperative.

For example, Huffman code does not achieve this (assuming such an order exists). That is, if source symbol X is before source symbol Y, the (binary) codeword for X is not necessarily before that of Y. If the source statistics is known a priori, an algorithm due to Hu and Tucker [9][8] can be used to construct the optimal tree which preserves alphabetic order in

*School of Computer Science, Carnegie Mellon University. Pittsburgh, PA 15213-3891, dennis@cs.cmu.edu. Work partly performed for Bellcore.

†Department of Computer Science, Boston University, Boston MA 02215, raj@cs.bu.edu. Work partly performed for Bellcore.

‡Bellcore, 445 South St, Morristown, NJ 07960. venkie@wind.bellcore.com

§Department of Information Engineering, The Chinese University of Hong Kong, Shatin, N.T. Hong Kong, kwwei@ie.cuhk.hk. Work partly performed while with Bellcore

the codewords while minimizing the average codeword length (under the alphabetic constraint). For a source with unknown statistics, one needs to use an adaptive or dynamic algorithm. If there is no alphabetic order, adaptive versions of Huffman's algorithm [4][5][13][17] can be used. However, there are relatively fewer works on adaptive alphabetic coding trees.

We describe several applications of the *splay tree* idea to this problem and analyze their compression efficiency in information-theoretic terms. For example, the average codeword length of the semisplay tree is no more than twice the *source entropy* plus some minor terms. Subsequently, it was discovered that this result can be derived from a remark in [16]. Other splay tree versions we studied include the regular splay tree and splay trees with depth three and four. The bounds we derived on their performance are weaker than those for the semisplay tree.

2 Move-To-Front Compression with secondary lists

The original move-to-front data compression algorithm [BSTW] can be viewed as consisting of two modules. The first module accept as input a parsed stream of words. It translates these words, by means of a move-to-front list, to a stream of integers which we call *tokens*: when a word occurs it is encoded by its rank in the linear list and it is moved to the front of the list. (Exceptions for new words are ignored at this time for the sake of clarity.) The second module translates each token into a self-delimiting binary string (say using Elias integer representation scheme).

The two-list move-to-front data compression algorithm, or MTFS (Move-To-Front with a Secondary list), has three modules. The first module translates a parsed stream of words into a stream of tokens. The second module translates a stream of tokens into a different stream of integers. The third module translates each integer into a binary string, by another an integer represented in a self-delimiting scheme. Each of the first module and the second module maintains a move-to-front list. Each encodes an input item by its position in the list. Some other implementation details of MTFS will be provided later.

The mtf-algorithm exploits the *temporal locality of reference* in data – words often come into periods of frequent use before possibly falling into other periods of disuse. So, every word in a mtf-algorithm is coded by its interarrival times. But the inter-arrival times themselves may be correlated or clustered, exhibiting a *spatial locality of reference*, which is what MTFS-algorithm intends to exploit That is, tokens which appear near a given token t in the input stream have a good chance of appearing in the vicinity of future occurrences of t.

For example, assume that the words `diet` and `coke` frequently appear together and that `diet` is at position x and (hence) `coke` at position $x-1$ in the list. The MTF algorithm (without secondary lists) would output x and move `diet` to the front. Next when `coke` appears, it has rank x, and the output will be x once again. With a secondary mtf-list, we first find `diet` in the primary list and find its rank x. We then look up x in our secondary list and find its rank, say y there. We output y, move `diet` to the front of primary list and move x to the front of the secondary list. When we see `coke`, its rank is x in the primary list but now x has rank one in the secondary list. We output 1 and move `coke` to the beginning of its list. (We don't need to move anything in the secondary list as x is already at its front.) Thus instead of outputting x x as in the original MTF algorithm, we output y 1.

The power of MTFS is at its utmost clear on a periodic sequence. Assume the periodic sequence $(a, b, c, ..., a, b, c, ..., a, b, c, ...)$ is the input to the compression algorithm. Then MTF will have achieved essentially no compression while MTFS will have achieved very high compression ratio.

2.1 Analysis.

Let $x_1, x_2, ..., x_N$ be the source sequence. Let $\ell(j)$ be the length of Elias encoding of the positive integer j. Let $N(j)$ denote the number of appearances of the integer j as a token. Using analysis similar to that in [1], one can show

THEOREM 1. *The output length of the secondary list can be upper bounded by*

$$\sum_j N(j)\ell(j)+\sum_j N(j)(\log \frac{N}{N(j)}+2\log(1+\log \frac{N}{N(j)})+1)$$

It is an open problem to precisely characterize the class of sources for which the secondary list helps compression. But we give an analysis that gives a better picture still.

The output of a move to front compression algorithm consists of new words and indicies of old words in the list. Assuming large enough list szie and long enough history so that every future input is in the list. It would make sense to compress these indices, although asymptotically this additional effort would not yield an improvement since mft-compression acheives entropy. But in finite runs, this may make a difference: Lempel-Ziv algorithm (compress version) outperforms mtf-compression. One reason may be that there are differences in rates of convergence to optima.

We first show that the distribution of the indices have the following closure property: The distributions of the indices of any letter of the alphabet, with respect to the joint probability of observations made, are themselves stationary and ergodic, if the original source is and the observations are long enough and mixing occurs. This seems to be of interest on its own.

Let $\{X_i\}_{-\infty}^{\infty}$ be a stationary ergodic process, with $X_i \in \Sigma = \{a_1, a_2, ..a_N\}$.. This may be viewed as a standard extension (by sliding) of a single-ended source $X_i', i \geq 0$ which has the same distribution as X_i for $i \geq 0$. For a finite set of integers i, let *history h* denote an event $[X_i = b_i, i := 1, \ldots, r]$ Call a history complete if it includes the first occurrences of every $a \in \Sigma$. Below it may be assumed that that we have observed the process long enough and mixing occurs (i.e. for all events A and B, the conditional probability of B at time t, given that the event A occurred at time $t-k$, is exactly the stationary probability of A, as $k \to \infty$. For an $a \in \Sigma$, let the times of occurrence be $R_0^i R_1^a, R_2^a, \ldots$. It is known that R_j^a are almost surely finite and put $T_j^a := R_j^a - R_{j-1}^a$ be the inter arrival times, which correspond to the indicies produced by the mtf-compression. In the following, (b) is a well known fact[See Breiman], and has been used before in the compression analysis (see Wyner and Ziv). It is interesting to note that one can use (a) to get a qunatified analysis of splay trees: e.g. run time in terms of the entropy of the source for example in the "working set" theorem of Tarjan and Sleator. We omit details here.

LEMMA 1. **a** *For each a, the return times of a is stationary and ergodic w.r.t. the probability measure* $P(./X_0 = a)$.

b *(Kac) Expected value of* T_j^a *is* $1/P(X_0 = a)$.

To see why (a) motivates secondary lists, we first look at mtf-compression itself. Clearly the length of the compressed file is $\sum_a \sum_j \log T_j^a$, plus the overhead for first time occurrences and for making the representations self-delimiting-which can be ignored. An analysis of this for indpendent identically distributed sources, given in [1, 3] shows that the expected inter arrival time of a letter a is $1/p(a)$. Hence the expected length of the output per symbol is

$$1/L \sum_a \sum_j \log T_j^a = \sum_a p(a) \log 1/p(a) = H$$

where H is the entropy of the source and L is the length of the sequence. This can be extended to stationary ergodic sources using Kac's lemma, and the observation that any letter a will occur $p(a)$ fraction of the time L large enough L, since the source is stationary and ergodic. For stationary ergodic sources, the mtf compresses at H bits per symbol, where H is the entropy of the source, i.e. asymptototically optimal. Now it suggests itself that we may apply mtf to inter-arrival times as well. To arrive at reasonable estimate for for the primary list size note that average (over symbols a) of mean-interarrival times of each letter is N.

Similar assumptions that having observed the process long enough (and thus built a long enough primary list) so that future events become asymptotially independt of the observation are dealt with in [?] for the analysis of Lempel-Ziv algorithm's versions. The above assumption about mixing can be relaxed, but we will not deal with them here.

The analysis of inter arrival times is more complicated. First, we need to know that conditioned on the (long) observations of more than one symbol jointly earlier, the inter-arrival times are stationary and ergodic w.r.t. the conditional distribution induced by the observation.

THEOREM 2. *For a large enough history h, the inter arrival times of every letter a is stationary ergodic w.r.t.* $P(./h)$.

Th proof is in the same lines of the lemma above and for space constraints, is omitted here.

Hence it would seem appropriate to subject these indicies for another round of move-to-front compression. If the inter arrival times are clustered around a smaller interval of size $\sqrt{l}$, where l is the length of the primary list, then the compression will be achieved by secondary lists. Alternately, the probabilities of a subset of the letters are approximately equal, and the interarrival time of one letter is likely to be repeated by other letters, when secondary list helps. One may restate this as follows: Consider the interarrival times the letters T_j^a. Then one may require that the expectation and variance of these variables be within a constant factor of each other. (The algorithm need not know these parameters.) We detail this in the journal version.

3 Compression Efficiences of Various Splay Trees.

Let $s_1, s_2, s_3, \ldots, s_T$ be a sequence of symbols from a source S consisting of n symbols. Assume the source is memoryless, i.e. the symbols are generated independently according to a fixed probability distribution on S. Assume further that there is an alphabetic order among the members of S. The purpose of our study is to construct and maintain an adaptive code tree for S. The code tree is used to encode the symbol sequence. It is adapted after encoding each symbol, for the purpose of optimization for encoding of future symbols. The goal is to minimize the average length (in bits) per symbol of the encoder output for the entire sequence $s_1, s_2, \cdots, s_T$.

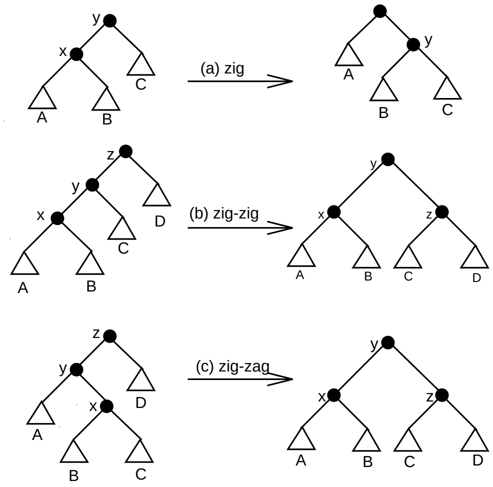

Figure 1: Semisplay steps. (a) Case 1, zig. (b) Case 2, zig-zig. (c) Case 3, zig-zag.

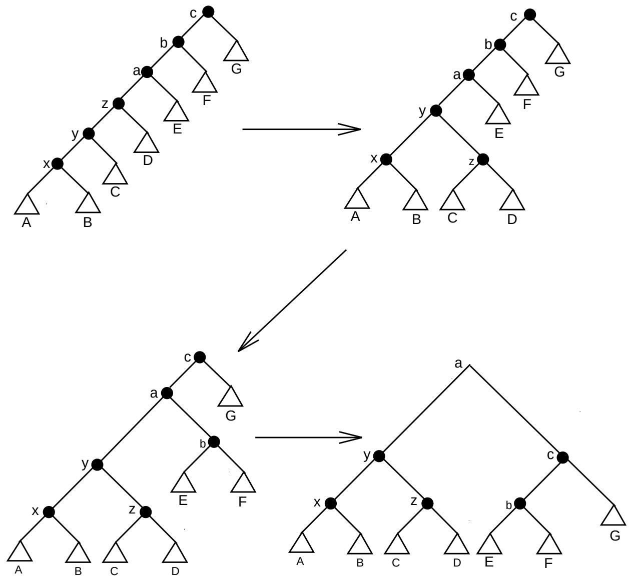

Figure 2: Semisplaying a tree at node x.

Additionally, it is required that the code tree preserves the alphabetic order among the source symbols, at all times.

We shall use several versions of the splay tree as the adaptive code tree. We restrict ourselves to binary trees throughout this paper. Our results also have applications in searching. An alphabetic code tree can be used as a search tree. The efficiency of the tree for searching is similar to its efficiency in compression. We do not address the details of searching applications in this paper.

3.1 An Adaptive Alphabetic Data Compression Algorithm. Given a sequence of source symbols $s_1, s_2, ..., s_T$, the algorithm proceeds as follows: (We assume there is an arbitrary initial tree whose leaves correspond to symbols of the alphabet S in alphabetic order at the beginning of the compression session.)

At time t, output the path from the root to the leaf corresponding to s_t as the codeword; then "splay" the tree at s_t's parent node. Naturally, the next symbol, s_{t+1}, will be encoded using the new tree, and so on.

We study the compression efficiency of the above adaptive alphabetic code tree. We present upper bounds on the average number of bits per symbol used in the encoding. There are several versions of "splaying" we consider.

3.2 Semisplaying.

3.2.1 Description.

A *semisplaying step* is shown in Figure 1. *Semisplaying a tree at a node x* means repeatedly applying the semisplaying steps to every two branches of the path from x towards the root of the tree. (*Splaying a tree at a node x* means repeatedly applying the splaying step to every two branches of the path from x towards the root.) If there are an odd number of branches between x and the root, then the last splay step applied is a zig. An example of semisplaying a tree at node x is shown in Figure 2.

D. Jones [11] suggested the use of semisplaying for the purpose of data compression with an alphabetic order. He also studies variations of splaying for compression without an alphabetic order. In the following, we analyze the efficiency of semisplaying for compression with an alphabetic order. The tools of our analysis are from information theory.

3.2.2 Compression Efficiency. Let p be the empirical distribution of the source symbols, i.e. $p_s = (1/T)\cdot$(number of occurrences of s. Let w be an arbitrary probability distribution on the source symbols. Let $p_{\min} = \min_s p_s$, $w_{\min} = \min_s w_s$. We assume $p_{\min} > 0$, $w_{\min} > 0$. Let $H(p) = \sum_s p_s \log(1/p_s)$ denote the entropy of the source. Let $D(p||w) = \sum_s p_s \log(1/w_s)$ denote the *divergence* of p and w. Because log is concave, by Jenson's inequality we have $\sum_s p_s \log(w_s/p_s) \leq \log \sum_s w_s = \log 1 = 0$. Therefore, $D(p||w) \geq H(p)$ and the equality holds when the two distributions $p = w$.

Theorem 1: For the semisplaying tree, the average codeword length per symbol in semisplaying is no more than $2D(p||w) + 2 + (n/T)\log(1/w_{min})$, for any probability distribution w on the source symbols.

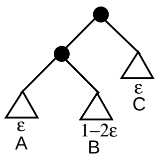

Figure 3: An optimal alphabetic tree.

Corrollary 1: The average codeword length per symbol in semisplaying is no more than $2H(p) + 2 + (n/T)\log(1/p_{min})$.

In contrast, the average codeword length achieved by Huffman code (for non-alphabetic applications) is upper bounded by $H(S) + 1$, and that achieved by Hu-Tucker algorithm (for alphabetic applications) is upper bounded by $H(S) + 2$. An extreme case of optimal alphabetic code tree is shown in Figure 3. The symbol "B" has probability close to one. The average codeword length is nearly two, and the entropy is nearly zero. Improvements to this bound based on properties of the source probability distribution exist, e.g. [19][14].

The proof is similar to that in Sleator and Tarjan [16] for the bound on the amortized cost of (regular) splaying. However, there are several specializations for compression studies. Each leaf node has a positive *weight* $w(i)$. For compression studies, the weight of each internal node is set to zero, and the sum of all weights is equal to 1. The *size* of a node x, denoted $s(x)$, is the sum of the individual weights of all items in the subtree rooted at x. Note that $s(t) = 1$ for the root t, $s(x) = w(x)$ if x is a leaf. The *rank* of a node x, denoted $r(x)$, equals $\log(s(x))$. The *potential* of the tree, denoted Φ, equals the sum of the ranks of all its nodes. All logarithms have base two in this paper.

The *length of the codeword* corresponding to the source symbol s_t equals the length of the path from node X, which s_t currently resides, to the root. The *amortized cost* of encoding s_t equals $\ell + \Phi' - \Phi$ where ℓ is the codeword length, Φ is the potential of the code tree which is used for encoding s_t, and Φ' is the potential of the code tree after adaptation corresponding to s_t.

Lemma 1: Assume the symbol encoded is at node X whose parent is node x. Let R_0 be the root of the entire tree. The amortized cost is at most $2r(R_0) - 2r(x) + 2$.

Proof: We analyze the amortized cost of a single semisplay step first. Let A, B, C, D denote the sizes of the subtrees as shown in Figure 1. Note that the size of an internal node equals the sum of the sizes of the subtrees below it, because the weight of an internal node is zero.

Case 2. The potential of the tree before this semisplay step is $\Phi = \log A + \log B + \log C + \log D + \log(A + B) + \log(A + B + C) + \log(A + B + C + D)$. The potential of the tree after this semisplay step is $\Phi' = \log A + \log B + \log C + \log D + \log(A+B) + \log(C+D) + \log(A + B + C + D)$. The amortized cost due to this individual step is $\ell + \Phi' - \Phi = \ell + \log\frac{(C+D)}{(A+B+C)}$. For technical reasons which will become clear shortly, we set the path length $\ell = 2$. We wish to show the amortized cost is no more than $2\log\frac{A+B+C+D}{A+B}$.

We have $2 + \log\frac{(C+D)}{(A+B+C)} - 2\log\frac{A+B+C+D}{A+B} = 2 + \log\frac{(A+B)(C+D)}{(A+B+C+D)^2} + \log\frac{A+B}{A+B+C} \leq \log\frac{4(A+B)(C+D)}{(A+B+C+D)^2} = \log\frac{(A+B+C+D)^2-(A+B-C-D)^2}{(A+B+C+D)^2} \leq \log 1 = 0$

Similarly, we can show the amortized cost in Case 3 is no more than $2\log\frac{A+B+C+D}{B+C}$, and in Case 1 (path length $\ell = 1$) it is no more than $1 + \log\frac{A+B+C}{A+B}$.

The amortized cost of semisplaying the node x (i.e. semisplaying the entire path from x to the root) is the sum of the amortized costs of individual semisplay steps along the entire path from x to the root. The above upper bounds can be used. The sum of the upper bounds for individual semisplay steps along the path telescopes to $1+2r(R_0)-2r(x)$. The term 1 corresponds to the case when the length of the path from x to root is an odd number.

There is an additional bit in the codeword for the branch from node X to node x, which is not accounted for in the above analysis. Thus the upper bound is $2r(R_0) - 2r(x) + 2$. □

Proof of Theorem 1. Let w_s be the weight of the leaf node corresponding to symbol s, where $\sum_s w_s = 1$. Then $\Phi_{\max} \leq \sum_s \log w_s$, $\Phi_{\min} \geq \sum_s \log w_s - n\log 1/w_{\min})$. Consider a leaf associated with symbol s. The amortized cost of encoding this symbol, which is associated with splaying at its parent x, is no more than $2r(R_0) - 2r(x) + 2 \leq 0 - 2\log w_s + 2$. (Note $r(R_0) = \log(\sum_s w_s) = \log 1$.)

The total length of the encoder output is no more than $T(2 + \sum_s(-2p_s \log w_s)) + \Phi_{\max} - \Phi_{\min} \leq T(2D(p||w) + 2) + n\log 1/w_{\min})$. □

3.3 Regular Splaying.

A regular splaying step is shown in Figure 4. Only case 2, the zig-zig, is different from semisplaying. Splaying a tree at node x means repeatedly applying the splaying steps to every two branches of the path from node x towards to root. If the length of the path is odd, then the last branched is splayed as a zig.

The splay tree can be used as the adaptive code tree for compression applications with an alphabetic ordering constraint, similar to the semisplay tree. Its efficiency is analyzed below.

Theorem 2: For the splay tree, the average

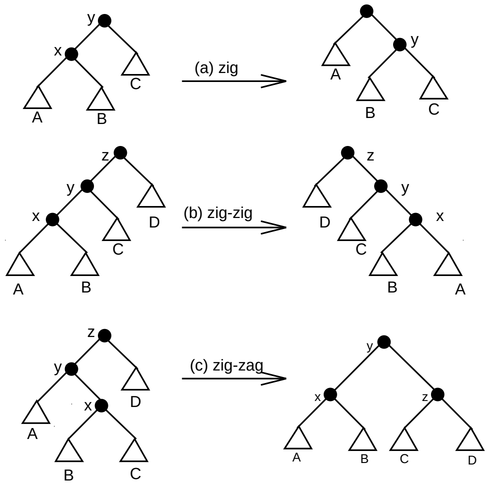

Figure 4: A regular splaying step

codeword length per source symbol is upper bounded by $3H(p)+2+(n/T)\log(1/p_{\min})$, where p is the empirical source probability distribution.

In [16], Lemma 1, the amortized access time of the splay tree is upper bounded by $3(r(R_0)-r(x))+1$. The same analysis implies that the amortized compression cost of the splay tree is upper bounded by $3(r(R_0)-r(x))+2$. The additional bit comes from the branch from node x to node X; it was not counted in [16] but needs to be counted here. Then the proof of Theorem 2 is similar to that of Theorem 1.

In the above analysis, the weights of leaf nodes remain constant throughout splayings. By varying the weights over time, we can obtain another result. Consider the input stream of symbols $s_1, s_2, ..., s_T$. For $1 \le j \le T$, let $t(j)$ be the positive integer such that s_j is the $t(j)$-th most recently appearing symbol. I.e. there are $t(j)$ different symbols appearing in the interval since, and including, the last appearance of the symbol s_j. For technicality, $t(j)=n$ for the first appearance of any symbol.

3.4 Splayings with Depth Three and Four.

In a splaying step, we can specify any transformation between trees which preserves the alphabetic order. As long as the decompression algorithm uses the same transformations as the compression algorithm, the original file can be faithfully recovered. The objective is to design these tree transformations in order to minimize the average number of bits used to transmit each symbol in the input sequence.

Both splaying and semisplaying are transformations among trees with four leaves or less. Let the **depth** be the maximum path length from the root to an internal

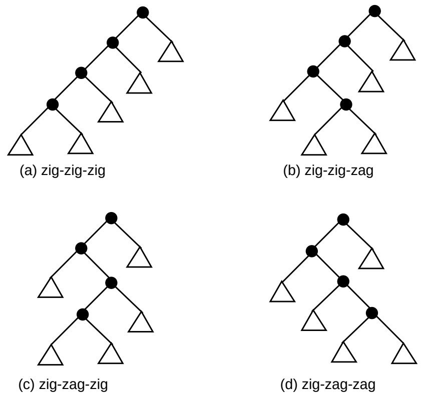

Figure 5: Beginning trees, depth-3 splay steps.

node. (We adopt this definition because we always splay at an internal code.) Then splaying and semisplaying deal with trees of depth two or less. We investigate the transformations, or "splaying", of trees of depth 3 or 4. Our goal is to find "deep" splaying steps whose efficiency is better than semisplaying. Through an exhaustive computer search, we report that no splayings of depth three or four were found whose performance is provably better than semisplaying with our analytical technique. This does not rule out the possibility that stronger analysis can produce stronger results.

3.4.1 Depth Three

There are four *beginning trees* to consider. As shown in Figure 5, they correspond to the cases (a) zig-zig-zig, (b) zig-zig-zag, (c) zig-zag-zig, and (d) zig-zag-zag. Each beginning tree can be "splayed" into any ending tree with the same number of leaves and with the alphabetic order preserved. There are 14 choices of *ending tree* for each beginning tree, they are shown in Figure 6. They are denoted $E_1, E_2, \cdots, E_{14}$. For each of the $4 \times 14 = 56$ "splaying" cases, we use numerical methods to calculate the smallest value α such that the amortized cost of a single "splaying step" is upper bounded as follows:

$$A(B_i \to E_j) = 3 + \Phi(E_j) - \Phi(B_i) \le \alpha_{i,j}(r(t) - r(x)),$$

for all possible weight assignments on the leaves. Note $r(t)$ is the rank of the top node in the "splay step" and x is the node to be splayed. Then for each beginning tree B_i, we choose to splay it into ending tree $E_{j(i)}$ such that $\alpha_i = \alpha_{i,j(i)} \le \alpha_{i,j'}$ for any j', $1 \le j' \le 14$.

Let $\alpha = \max_i\{\alpha_i\}$. Then we can show the cost of splaying the code tree at a node x is upper bounded by $\alpha(r(R_0) - r(x)) + 3$ by telescopic summation. This in turn enables us to prove that the compression efficiency of the algorithm is upper bounded by $\alpha H(S) + 3$.

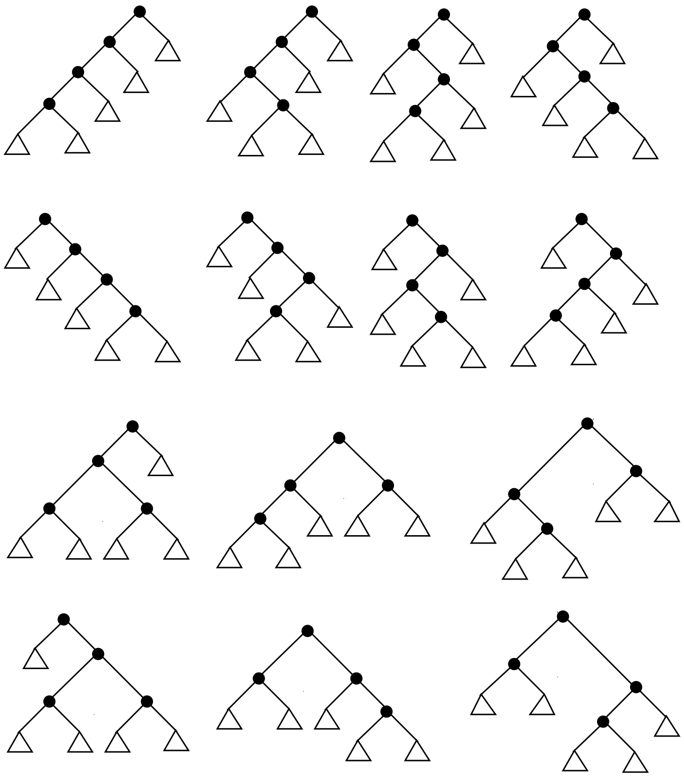

Figure 6: Ending trees, depth-3 splay steps.

The result of our calculations is disappointing. The value of α calculated from the above method is strictly greater than 2. Therefore, it does not yield improvement on semisplaying.

3.4.2 Depth Four

We repeat the search for splaying steps in trees with depth 4. There are 8 beginning trees and 42 ending trees. The search result on $8 \times 42 = 336$ cases is negative. The only way to obtain a value of $\alpha = 2$ is to use combinations of semisplaying on trees with depth 4. Therefore, this method does not yield improved bounds. However, this result does not preclude the possibility that better proof techniques can improve the bound.

3.5 Simulation Results.

The preliminary results reported below are from tests on formatted telephony data (see[BAF, FCIF]) and some C-programs. The tested file sizes varied from a few kilobytes to 70Megabytes. Usually the front-end that breaks an input stream into words is trivial and a few lines of C-code. We keep a large regular mtf list, usually 2^{14} entries, and a small secondary list, usually 63 or 511 entries. The output integer of the first module is sent directly to the third module, unless it is less than 512 – in this case it is processed by the second module. We used splay tree to implement the search tree. We chose a version called *simple splaying* [16] where the second rotation in the zig-zag case is omitted. Making a tradeoff between speed and compression, we used a variable-length encoding of the integers but we required our codewords to always be an integral number of nibbles (4 bits). Our ongoing experiments indicate that using alphabetic codes induced by splay trees (regular and semi-spaly) would improve the compression further. More optimizations and simulations will be needed before the best efficacy of the MTFS schemes can be known.

The compression ratios are competitive to or better than popular compression programs: Lempel-Ziv 77 based lharc, pkzip and Lempel-Ziv 78 based compress. In our experiments, of these algorithms, LZ-78,LZW based compress was best in terms of compression within a small error. MTFS compression was within a small error (approx. one percent) of compress or sometimes better when files were large (i.e. exceeding 10 Meg). In terms of running times our algorithm comparable or better than than lharc or pkzip algorithms, sometimes by a factor of 2 or more. When compared to compress, on large files, our run times varied a bit and was influenced by the computing environments. In a large set of runs, Compress and MTFS ran at essentially same speeds while compressing, and while de-compressing compress was often slower. In another set of runs, with an enhanced version of the file server, compress was faster than MTFS, by a factor of 1.5-2. More studies are needed to isolate and understand these runtime/buffering effects.

The splay tree implementation seemed to benefit from the fact that average depth of access in a splay tree is proportional to the entropy. For example, the average search depth is around 4 in a splay tree of 2^{14} items. Note that a dictionary data structure which is optimal in the the worst case and takes $\log n$ units of time, may slow down the algorithm considerably. Thus the amortized analysis principle behind splay trees is more relevant here. Section on alphabetic codes gives further analysis related to this observation.

Addition of a tertiary list improved the compression but this effect is in general was not significant.

Acknowledgements

Thanks to Danny Sleator for discussions in the initial phase,and Bill Aiello for many insightful discussions.

size	*method*	*packed size*
16384	MTFS	1535
	compress	2416
	lharc	1376
60883	MTFS	6092
	compress	7667
	lharc	4206
118890	MTFS	29823
	compress	28796
	lharc	16391
1923556	MTFS	134671
	compress	139649
	lharc	128606
2238763	MTFS	219632
	compress	221719
	lharc	161592
17187643	MTFS	1278075
	compress	2108551
	lharc	1980503

Compression of data type I.

size	*method*	*packed size*(t_c, t_d)
98,383,860 f1	MTFS	1,924,657 (340,1102)
	compress	1,801,775 (263,907)
	lharc	4,034,823(2733,970)
98,369,004 f2	MTFS	1,785,302(269,1373)
	compress	1,674,539 (236,1130)
	lharc	4006169 (2748,943)
20Meg	MTFS	427,516(23,632)
	compress	383,019 (24,1901)
	lharc	805,554 (480,2304)
5,065,896(*)	MTFS (*+)	149,682 (10,14)
	compress	96961 (5.4,12.4)
	gzip	89373(5.7,20.3)
f1 (*)	MTFS (*+)	(270,390)
	compress	(193,411)
	gzip	1,771,636(223,672)
[f2] (*)	MTFS(*+)	(223,406)
	compress	(202,376)
	gzip	164,697 (225,606)

Compression of Data Type 2

size	*method*	*packed size*(t_c, t_d)
1Meg (*)	MTFS (**)	317,101 (5.3,5.7)
	MTFS (***)	347,417 (5.9,4.6)
	compress	241,139 (4.5,2.6)
	gzip	227,630 (6.0,4.8)
39Meg	MTFS	13.5Meg (338,594)
	compress	12.2Meg(167,736)
	lharc	12.6Meg(2009,604)
50Meg	MTFS	(12.5meg) (380,686)
	compress	11.4Meg(261,1125)
	lharc	11.7meg(2344,1000)
20Meg(*)	MTFS (**)	5.1Meg (100,108)
	compress	4.7Meg (50.2,50.6)
	lharc	4.9Meg(375,59)
	gzip	4.4Meg(135,64)
20Meg	MTFS (**)	5.12Meg(113,2060)
	compress	4.7Meg(112,2180)
	lharc	4.9Meg(974,2693)

Compression of Data Type 3

(*)= With a faster network file Server;

(*+)= Primary list size=32K, Secondary =4K;

Tertiary list=63;

(**)= Primary list size=16K, Secondary =511;

(***)=Primary is 4K,secondary=511

t_c = Time to compress,

t_d = Time to decompress.

Variations of the list lengths are for illustrative purposes.

References

[BAF] *Bellcore Automatic Message Accounting Format (BAF) Requirements.* Technical Report TR-NWT-001100, Bell Communications Research, Issue 1, November 1990, Revision 7, March 1992.

[1] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, *A locally adaptive data compression scheme,* Comm. of the ACM, 29 (1986) 320–330.

[2] L. Breiman, *Probability*, SIAM.

[3] P. Elias, *Interval and recency rank source coding: two on-line adaptive variable-length*

schemes, IEEE Trans. on Info. Theory, 33 (1987) 3–10.

[FCIF] *FCIF—Flexible Computer Interface Format*, BellCore (Technical Advisory) TA-STS-000873, December 1987.

[4] N. Faller, *An adaptive system for data compression,* Record of the 7th Asilomar Conf. on Circuits, Systems, and Computers (1973) 593–597.

[5] R. G. Gallager, *Variations on a theme by Huffman,* IEEE Trans. on Information Theory, 24 (1978) 668–674.

[6] E. N. Gilbert and E. F. Moore, *Variable-length binary encodings,* Bell System Tech. J., 38 (1959) 933–968.

[7] R. N. Horspool and G. V. Cormack, *A general purpose data compression technique with practical applications,* Proceedings of the CIPS Session 84, (1984) 138–141.

[8] T. C. Hu, *A new proof of the T-C algorithm,* SIAM J. Appl. Math., 25 (1973) 83–94.

[9] T. C. Hu and A. C. Tucker, *Optimal computer search trees and variable length alphabetic codes,* SIAM J. Appl. Math., 21 (1971) 514–532.

[10] D. A. Huffman, *A method for the construction of minimum-redundancy codes,* Proc. of the IRE, 40 (1959) 933–968.

[11] D. W. Jones, *Application of splay trees to data compression,* Comm. of the ACM, 31 (1988) 996–1007.

[12] D. E. Knuth, *Optimum binary search trees,* Acta Informat., 1 (1971) 14–25.

[13] D. E. Knuth, *Dynamic Huffman coding,* J. Algorithms, 6 (1985) 163–180.

[14] N. Nakatsu, *Bounds on the redundancy of binary alphabetical codes,* IEEE Trans. Information Theory, 37 (1991) 1225–1229.

[15] B. Y. Ryabko, *Data compression by means of a 'book stack',* Prob. Inf. Transm., 16 (1980) 265–269.

[16] D. D. Sleator and R. E. Tarjan, *Self-adjusting binary search trees,* Comm. of the ACM, 32 (1985) 652–686.

[17] J. S. Vitter, *Design and analysis of dynamic Huffman coding,* Proc. 26th Symp. on Foundations of Computer Science, (1985) 293–302.

[18] A. D. Wyner and J. Ziv, *Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression,* IEEE Trans. on Information Theory, 35 (1989) 1250–1258.

[19] R. W. Yeung, *Alphabetic codes revisited,* IEEE Trans. on Information Theory, 37 (1991) 564–572.