Algorithms and Data Structures

Daniel Sleator Editor

Data Compression with Finite Windows

EDWARD R. FIALA and DANIEL H. GREENE

ABSTRACT: *Several methods are presented for adaptive, invertible data compression in the style of Lempel's and Ziv's first textual substitution proposal. For the first two methods, the article describes modifications of McCreight's suffix tree data structure that support cyclic maintenance of a window on the most recent source characters. A percolating update is used to keep node positions within the window, and the updating process is shown to have constant amortized cost. Other methods explore the tradeoffs between compression time, expansion time, data structure size, and amount of compression achieved. The article includes a graph-theoretic analysis of the compression penalty incurred by our codeword selection policy in comparison with an optimal policy, and it includes empirical studies of the performance of various adaptive compressors from the literature.*

1. INTRODUCTION

Compression is the coding of data to minimize its representation. In this article, we are concerned with fast, one-pass, adaptive, invertible (or lossless) methods of digital compression which have reasonable memory requirements. Such methods can be used, for example, to reduce the storage requirements for files, to increase the communication rate over a channel, or to reduce redundancy prior to encryption for greater security.

By "adaptive" we mean that a compression method should be widely applicable to different kinds of source data. Ideally, it should adapt rapidly to the source to achieve significant compression on small files, and it should adapt to any subsequent internal changes in the nature of the source. In addition, it should achieve very high compression asymptotically on large regions with stationary statistics

All the compression methods developed in this article are *substitutional*. Typically, a substitutional compressor functions by replacing large blocks of text with shorter references to earlier occurrences of identical text [3, 5, 29, 34, 36, 39, 41–43]. (This is often called Ziv-Lempel compression, in recognition of their pioneering ideas. Ziv and Lempel, in fact, proposed two methods. The unqualified use of the phrase "Ziv-Lempel compression" usually refers to their second proposal [43]. In this article, we will be primarily concerned with their first proposal [42].) A popular alternative to a substitutional compressor is a statistical compressor. A symbolwise statistical compressor functions by accurately predicting the probability of individual symbols, and then encoding these symbols with space close to $-\log_2$ of the predicted probabilities. The encoding is accomplished with either Huffman compression [17] which has recently been made one-pass and adaptive [11, 22, 37], or with arithmetic coding, as described in [1, 14, 20, 25, 26, 31–33]. The major challenge of a statistical compressor is to predict the symbol probabilities. Simple strategies, such as keeping zero-order (single symbol) or first-order (symbol pair) statistics of the input, do not compress English text very well. Several authors have had success gathering higher-order statistics, but this necessarily involves higher memory costs and additional mechanisms for dealing with situations where higher-order statistics are not available [6, 7, 26].

It is hard to give a rigorous foundation to the substitutional vs. statistical distinction described above. Several authors have observed that statistical methods can be used to simulate textual substitution, suggesting that the statistical category includes the substitutional category [4, 24]. However, this takes no account of the simplicity of mechanism; the virtue of textual substitution is that it recognizes and removes coherence on a large scale, oftentimes ignoring the smaller scale statistics. As a result, most textual substitution compressors process their compressed representation in larger blocks than their statistical counterparts, thereby gaining a significant speed advantage. It was previously believed that the speed gained by textual substitution would necessarily cost something in compression achieved. We were surprised to discover that with careful attention to coding, textual substitution compressors can match the compression performance of the best statistical methods.

Consider the following scheme, which we will improve later in the article. Compressed files contain two types of codewords:

literal x	pass the next x characters directly into the uncompressed output
copy $x, -y$	go back y characters in the output and copy x characters forward to the current position.

So, for example, the following piece of literature:

```
IT WAS THE BEST OF TIMES,
                 IT WAS THE WORST OF TIMES
```

would compress to

(literal 26)`IT WAS THE BEST OF TIMES,`
  (copy 11 −26)**(literal** 3)`WOR`**(copy** 11 −27)

The compression achieved depends on the space required for the copy and literal codewords. Our simplest scheme, hereafter denoted **A1**, uses 8 bits for a literal codeword and 16 for a copy codeword. If the first 4 bits are 0, then the codeword is a literal; the next 4 bits encode a length x in the range [1 . . 16] and the following x characters are literal (one byte per character). Otherwise, the codeword is a copy; the first 4 bits encode a length x in the range [2 . . 16] and the next 12 bits are a displacement y in the range [1 . . 4096]. At each step, the policy by which the compressor chooses between a literal and a copy is as follows: If the compressor is idle (just finished a copy, or terminated a literal because of the 16-character limit), then the longest copy of length 2 or more is issued; otherwise, if the longest copy is less than 2 long, a literal is started. Once started, a literal is extended across subsequent characters until a copy of length 3 or more can be issued or until the length limit is reached.

A1 would break the first literal in the above example into two literals and compress the source from 51 bytes down to 36. **A1** is close to Ziv and Lempel's first textual substitution proposal [42]. One difference is that **A1** uses a separate literal codeword, while Ziv and Lempel combine each copy codeword with a single literal character. We have found it useful to have longer literals during the startup transient; after the startup, it is better to have no literals consuming space in the copy codewords.

Our empirical studies showed that, for source code and English text, the field size choices for **A1** are good; reducing the size of the literal length field by 1 bit increases compression slightly but gives up the byte-alignment property of the **A1** codewords. In short, if one desires a simple method based upon the copy and literal idea, **A1** is a good choice.

A1 was designed for 8-bit per character text or program sources, but, as we will see shortly, it achieves good compression on other kinds of source data, such as compiled code and images, where the word model does not match the source data particularly well, or where no model of the source is easily perceived. **A1** is, in fact, an excellent approach to general purpose data compression. In the remainder of this article, we will study **A1** and several more powerful variations.

2. OVERVIEW OF THE DATA STRUCTURE

The fixed window suffix tree of this article is a modification of McCreight's suffix tree [28] (see also [21, 34, 38]), which is itself a modification of Morrison's PATRICIA tree [30], and Morrison's tree is ultimately based on a Trie data structure [22, page 481]. We will review each of these data structures briefly.

A Trie is a tree structure where the branching occurs according to "digits" of the keys, rather than according to comparisons of the keys. In English, for example, the most natural "digits" are individual letters, with the lth level of the tree branching according to the lth letter of the words in the tree.

In Figure 1, many internal nodes are superfluous, having only one descendant. If we are building an index for a file, we can save space by eliminating the superfluous nodes and putting pointers to the file into the nodes rather than including characters in the data structure. In Figure 2, the characters in parentheses are not actually represented in the data structure, but they can be recovered from the (position, level) pairs in the nodes. Figure 2 also shows a suffix pointer (as a dark right arrow) that will be explained later.

Figure 2 represents some, but not all, of the innovations in Morrison's PATRICIA trees. He builds the trees with binary "digits" rather than full characters, and this allows him to save more space by folding the leaves into the internal nodes. Our "digits" are bytes, so the branching factor can be as large as 256. Since there are rarely 256 descendants of a node, we do not reserve that much space in each node, but instead hash the arcs. There is also a question about when the strings in parentheses are checked in the searching process. In what follows, we usually check characters immediately when we cross an arc. Morrison's scheme can avoid file access by skipping the characters on the arcs, and doing only one file access and comparison at the end of the search. However, our files will be in main memory, so this consideration is unimportant. We will use the simplified tree depicted in Figure 2.

For **A1**, we wish to find the longest (up to 16 character) match to the current string beginning anywhere in the preceding 4096 positions. If all preceding 4096 strings were stored in a PATRICIA tree with depth $d = 16$, then finding this match would be straightforward. Unfortunately, the cost of inserting these strings can be prohibitive, for if we have just descended d levels in the tree to insert the string starting at position i then we will descend at least $d - 1$ levels inserting the string at $i + 1$. In the worst case this can lead to $O(nd)$

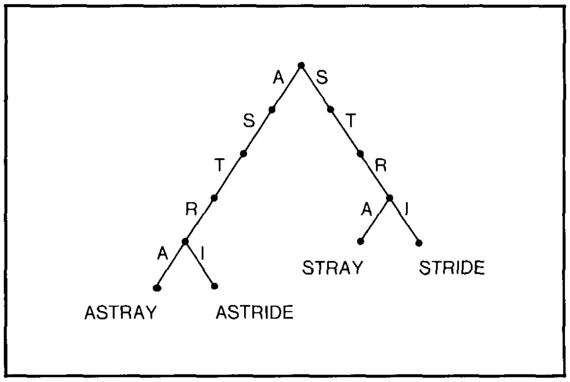

FIGURE 1. A Trie

insertion time for a file of size n. Since later encodings will use much larger values for d than 16, it is important to eliminate d from the running time.

To insert the strings in $O(n)$ time, McCreight added additional suffix pointers to the tree. Each internal node, representing the string aX on the path from the root to the internal node, has a pointer to the node representing X, the string obtained by stripping a single letter from the beginning of aX. If a string starting at i has just been inserted at level d we do not need to return to the root to insert the string at $i + 1$; instead, a nearby suffix pointer will lead us to the relevant branch of the tree.

Figure 3 shows how suffix links are created and used. On the previous iteration, we have matched the string aXY, where a is a single character, X and Y are strings, and b is the first unmatched character after Y. Figure 3 shows a complicated case where a new internal node, α, has been added to the tree, and the suffix link of α must be computed. We insert the next string XYb by going up the tree to node β, representing the string aX, and crossing its suffix link to γ, representing X. Once we have crossed the suffix link, we descend again in the tree, first by "rescanning" the string Y, and then by "scanning" from δ until the new string is inserted. The first part is called "rescanning" because it covers a portion of the string that was covered by the previous insert, and so it does not require checking the internal strings on the arcs. (In fact, avoiding these checks is essential to the linear time functioning of the algorithm.) The rescan either ends at an existing node δ, or δ is created to insert the new string XYb; either way we have the destination for the suffix link of α. We have restored the invariant that every internal node, except possibly the one just created, has a suffix link.

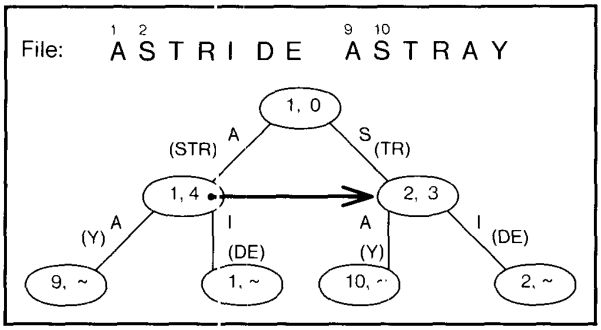

FIGURE 2. A PATRICIA Tree with a Suffix Pointer

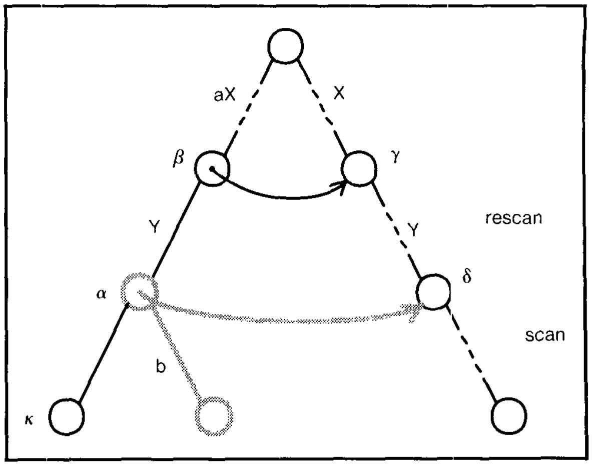

FIGURE 3. Building a Suffix Tree

For the **A1** compressor, with a 4096-byte fixed window, we need a way to delete and reclaim the storage for portions of the suffix tree representing strings further back than 4096 in the file. Several things must be added to the suffix tree data structure. The leaves of the tree are placed in a circular buffer, so that the oldest leaf can be identified and reclaimed, and the internal nodes are given "son count" fields. When an internal "son count" falls to one, the node is deleted and two consecutive arcs are combined. In Section 3, it is shown that this approach will never leave a "dangling" suffix link pointing to deleted nodes. Unfortunately, this is not the only problem in maintaining a valid suffix tree. The modifications that avoided a return to the root for each new insertion create havoc for deletions. Since we have not always returned to the root, we may have consistently entered a branch of the tree sideways. The pointers (to strings in the 4096-byte window) in the higher levels of such a branch can become out-of-date. However, traversing the branch and updating the pointers would destroy any advantage gained by using the suffix links.

We can keep valid pointers and avoid extensive updating by partially updating according to a percolating update. Each internal node has a single "update" bit. If the update bit is true when we are updating a node, then we set the bit false and propagate the update recursively to the node's parent. Otherwise, we set the bit true and stop the propagation. In the worst case, a long string of true updates can cause the update to propagate to the root. However, when amortized over all new leaves, the cost of updating is constant, and the effect of updating is to keep all internal pointers on positions within the last 4096 positions of the file. These facts will be shown in Section 3.

We can now summarize the operation of the inner loop, using Figure 3 again. If we have just created node α, then we use α's parent's suffix link to find γ. From γ we move down in the tree, first rescanning, and then scanning. At the end of the scan, we percolate an update from the leaf, moving toward the root, setting the position fields equal to the current position, and setting the update bits false, until we find a node with an update bit that is already false, whereupon we set that node's update bit true and stop the percolation. Finally, we go to the circular buffer of leaves and replace the oldest leaf with the new leaf. If the oldest leaf's parent has only one remaining son, then it must also be deleted; in this case, the remaining son is attached to its grandparent, and the deleted node's position is percolated upward as before, only at each step the position being percolated and the position already in the node must be compared and the more recent of these sent upward in the tree.

3. THEORETICAL CONSIDERATIONS

The correctness and linearity of suffix tree construction follows from McCreight's original paper [28]. Here we will concern ourselves with the correctness and the linearity of suffix tree destruction—questions raised in Section 2.

THEOREM 1. *Deleting leaves in FIFO order and deleting internal nodes with single sons will never leave dangling suffix pointers.*

PROOF. Assume the contrary. We have a node α with a suffix pointer to a node δ that has just been deleted. The existence of α means that there are at least two strings that agree for l positions and then differ at $l + 1$. Assuming that these two strings start at positions i and j, where both i and j are within the window of recently scanned strings and are not equal to the current position, then there are are two even younger strings at $i + 1$ and $j + 1$ that differ first at l. This contradicts the assumption that δ has one son. (If either i or j are equal to the current position, then α is a new node, and can temporarily be without a suffix pointer.)

There are two issues related to the percolating update: its cost and its effectiveness.

THEOREM 2. *Each percolated update has constant amortized cost.*

PROOF. We assume that the data structure contains a "credit" on each internal node where the "update" flag is true. A new leaf can be added with two "credits." One is spent immediately to update the parent, and the other is combined with any credits remaining at the parent to either: 1) obtain one credit to leave at the parent and terminate the algorithm or 2) obtain two credits to apply the algorithm recursively at the parent. This gives an amortized cost of two updates for each new leaf.

For the next theorem, define the "span" of a suffix tree to be equal to the size of its fixed window. So far we have used examples with "span" equal to 4096, but the value is flexible.

THEOREM 3. *Using the percolating update, every internal node will be updated at least once during every period of length "span."*

PROOF. It is useful to prove the slightly stronger result that every internal node (that remains for an entire period) will be updated twice during a period, and thus propagate at least one update to its parent. To show a contradiction, we find the earliest period and the node β farthest from the root that does not propagate an update to its parent. If β has at least two children that have remained for the entire period, then β must have received updates from these nodes: they are farther from the root. If β has only one remaining child, then it must have a new child, and so it will still get two updates. (Every newly created arc causes a son to update a parent, percolating if necessary.) Similarly, two new children also cause two updates. By every accounting, β will receive two updates during the period, and thus propagate an update—contradicting our assumption of β's failure to update its parent.

There is some flexibility on how updating is handled. We could propagate the current position upward before rescanning, and then write the current position into those nodes passed during the rescan and scan; in this case, the proof of Theorem 3 is conservative. Alternatively, a similar, symmetric proof can be used to show that updating can be omitted when new arcs are added so long as we propagate an update after every arc is deleted. The choice is primarily a matter of implementation convenience, although the method used above is slightly faster.

The last major theoretical consideration is the effectiveness of the **A1** policy in choosing between literal and copy codewords. We have chosen the following one-pass policy for **A1**: When the encoder is idle, issue a copy if it is possible to copy two or more characters; otherwise, start a literal. If the encoder has previously started a literal, then terminate the literal and issue a copy only if the copy is of length three or greater.

Notice that this policy can sometimes go astray. For example, suppose that the compressor is idle at position i and has the following copy lengths available at subsequent positions:

$$\begin{array}{cccccc} i & i+1 & i+2 & i+3 & i+4 & i+5 \\ 1 & 3 & 16 & 15 & 14 & 13 \end{array} \tag{1}$$

Under the policy, the compressor encodes position i with a literal codeword, then takes the copy of length 3, and finally takes a copy of length 14 at position $i + 4$. It uses 6 bytes in the encoding:

(literal 1)X(**copy** 3 − y)(**copy** 14 − y)

If the compressor had foresight it could avoid the copy of length 3, compressing the same material into 5 bytes:

(literal 2)XX(**copy** 16 − y)

The optimal solution can be computed by dynamic programming [36]. One forward pass records the length of the longest possible copy at each position (as in equation 1) and the displacement for the copy (not shown in equation 1). A second backward pass computes the optimal way to finish compressing the file from each position by recording the best codeword to use and the length to the end-of-file. Finally, another forward pass reads off the solution and outputs the compressed file. However, one would probably never want to use dynamic programming since the one-pass heuristic is a lot faster, and we estimated for several typical files that the heuristically compressed output was only about 1 percent larger than the optimum. Furthermore, we will show in the remainder of this section that the size of the compressed file is never worse than $\frac{5}{4}$ the size of the optimal solution for the specific **A1** encoding. This will require developing some analytic tools, so the non-mathematical reader should feel free to skip to Section 4.

The following definitions are useful:

Definition. *$F(i)$ is the longest feasible copy at position i in the file.*

Sample $F(i)$'s were given above in equation 1. They are dependent on the encoding used. For now, we are assuming that they are limited in magnitude to 16, and must correspond to copy sources within the last 4096 characters.

Definition. *$B(i)$ is the size of the best way to compress the remainder of the file, starting at position i.*

$B(i)$'s would be computed in the reverse pass of the optimal algorithm outlined above.

The following Theorems are given without proof:

Theorem. $F(i + 1) \geq F(i) - 1$.

Theorem. *There exists an optimal solution where copies are longest possible (i.e., only copies corresponding to $F(i)$'s are used).*

Theorem. *$B(i)$ is monotone decreasing.*

Theorem. *Any solution can be modified, without affecting length, so that* (**literal** x_1) *followed immediately by* (**literal** x_2) *implies that x_1 is maximum (in this case 16).*

We could continue to reason in this vein, but there is an abstract way of looking at the problem that is both clearer and more general. Suppose we have a non-deterministic finite automaton where each transition is given a cost. A simple example is shown in Figure 4. The machine accepts $(a + b)^*$, with costs as shown in parentheses.

The total cost of accepting a string is the sum of the transition costs for each character. (While it is not important to our problem, the optimal solution can be computed by forming a transition matrix for each letter, using the costs shown in parentheses, and then multiplying the matrices for a given string, treating the coefficients as elements of the closed semiring with operations of addition and minimization.) We can obtain a solution that approximates the minimum by deleting transitions in the original machine until it becomes a deterministic machine. This corresponds to choosing a policy in our original data compression problem. A policy for the machine in Figure 4 is shown in Figure 5.

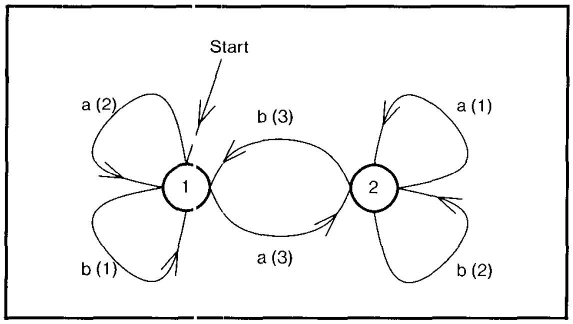

FIGURE 4. A Nondeterministic Automaton with Transition Costs

We now wish to compare, in the worst case, the difference between optimally accepting a string with the non-deterministic machine, and deterministically accepting the same string with the "policy" machine. This is done by taking a cross product of the two machines, as shown in Figure 6.

In Figure 6 there are now two weights on each transition; the first is the cost in the non-deterministic graph, and the second is the cost in the policy graph. Asymptotically, the relationship of the optimal solution to the policy solution is dominated by the smallest ratio on a cycle in this graph. In the case of Figure 6, there is a cycle from 1, 1′ to 1, 2′ and back that has cost in the non-deterministic graph of $2 + 1 = 3$, and cost in the policy graph of $3 + 3 = 6$, giving a ratio of $\frac{1}{2}$. That is, the policy solution can be twice as bad as the optimum on the string *ababababab*. . . .

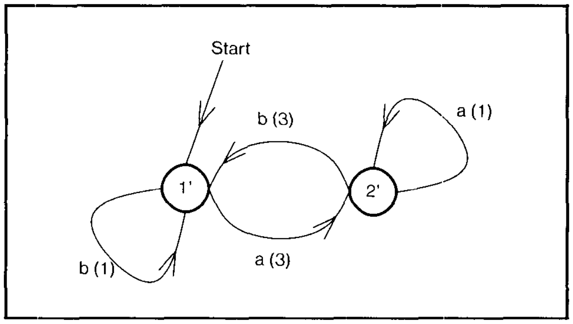

FIGURE 5. A Deterministic "Policy" Automaton for Figure 4

In general, we can find the cycle with the smallest ratio mechanically, using well known techniques [8, 27]. The idea is to conjecture a ratio r and then reduce the pairs of weights (x, y) on the arcs to single weights $x - ry$. Under this reduction, a cycle with zero weight has ratio exactly r. If a cycle has negative weight, then r is too large. The ratio on the negative cycle is used as a new conjecture, and the process is iterated. (Negative cycles are detected by running a shortest path algorithm and checking for convergence.) Once we have found the minimum ratio cycle, we can create a worst case string in the original automata problem by finding a path from the start state to the cycle and then repeating the cycle indefinitely. The ratio of the costs of ac-

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.