Extended Application of Suffix Trees to Data Compression

N. Jesper Larsson[†]

Abstract

A practical scheme for maintaining an index for a sliding window in optimal time and space, by use of a suffix tree, is presented. The index supports location of the longest matching substring in time proportional to the length of the match. The total time for build and update operations is proportional to the size of the input. The algorithm, which is simple and straightforward, is presented in detail.

The most prominent lossless data compression scheme, when considering compression performance, is prediction by partial matching with unbounded context lengths (PPM*). However, previously presented algorithms are hardly practical, considering their extensive use of computational resources. We show that our scheme can be applied to PPM*-style compression, obtaining an algorithm that runs in linear time, and in space bounded by an arbitrarily chosen window size.

Application to Ziv–Lempel '77 compression methods is straightforward and the resulting algorithm runs in linear time.

1 Introduction

String matching is a central task in data compression. In particular, in string substitution methods—such as the original scheme of Ziv and Lempel [14]—the dominating part of computation is string matching. Also, statistical data compression, such as the PPM methods [3, 4, 7], includes the operation of finding *contexts*, which are defined by strings. In effect, this is a string matching operation, which, particularly when contexts are long, occupies a major part of computational resources.

The suffix tree [6, 11] is a highly efficient data structure for string matching. A suffix tree indexes all substrings of a given string and can be constructed in linear time. Our primary contribution is to present a scheme that enables practical use of suffix trees for PPM*-style statistical modeling methods, together with its necessary theoretical justification. Also, application to Ziv–Lempel compression is natural.

Some compression schemes [3, 9] require that each character, once read from the input, resides in primary storage until all of the input has been processed. This is not feasible in practice. We need a scheme that allows maintaining only a limited part of the input preceding the current position—a *sliding window*. Fiala and Greene [5] claim to have modified McCreight's suffix tree construction algorithm [6] for use with a sliding window, by presenting a method for making deletions at constant amortized cost. However, a careful investigation reveals that they do not consider the fact that McCreight's algorithm treats the

[†]Dept. of Computer Science, Lund University, Box 118, S-221 00 LUND, Sweden (`jesper@dna.lu.se`)

input right-to-left, and therefore does not support expanding the indexed string with characters added to the right. This property of McCreight's algorithm makes it unfit for sliding window use if linear time complexity is to be maintained.

Here, we show that Ukkonen's suffix tree construction algorithm [11] can be extended to obtain a straightforward on-line sliding window algorithm which runs in linear time. We utilize the update restriction technique of Fiala and Greene as part of our algorithm.

The most promising statistical compression method appears to be finite context modeling with unbounded context length, in the style of the PPM* algorithm presented by Cleary, Teahan, and Witten [3]. (Some refinements are given by Teahan [10].) However, as presented in the original paper, this algorithm uses too much computational resources (both time and space) to be practically useful in most cases. Observing that the *context trie* employed in PPM* is essentially a suffix tree, our algorithms can be used to accomplish a practical variant of PPM*, where space requirements are bounded by a window size, and time complexity is linear in the size of the input.

In a survey of string searching algorithms for Ziv–Lempel '77 compression, Bell and Kulp [1] rule out suffix trees because of the inefficiency of deletions. We assert that our method eliminates this inefficiency, and that suffix trees should certainly be considered for implementation of the Ziv–Lempel algorithm.

2 Suffix Trees

We consider strings of characters over a fixed alphabet. The length of a string α is denoted $|\alpha|$.

A *trie* is a tree data structure for representing strings. Each edge is labeled with a character, and each stored string corresponds to a unique path beginning at the root. By $\langle u \rangle$ we denote the string corresponding to the path from the root to a node u.

A *path compressed* trie is a trie where only nodes with more than one outgoing edge are represented. Paths of unary nodes are collapsed to single nodes, which means that edges must be labeled with strings rather than single characters. By $depth(u)$ we shall denote the length of $\langle u \rangle$ rather than the number of edges on the associated path.

A *suffix tree* is a path compressed trie representing all suffixes (and thereby also all other substrings) of a string T. The tree has at most n leaves (one for each suffix), and therefore, since each internal node has at least two outgoing edges, the number of nodes is less than $2n$. In order to ensure that each node takes constant storage space, an edge label is represented by pointers into the original string. Note that we do not (as is otherwise common) require that the last character of T is unique. Hence, our suffix trees are not guaranteed to have a one-to-one correspondence between leaves and suffixes.

We adopt the following convention for representing edge labels: Each node u in the tree holds the two values $pos(u)$ and $depth(u)$, where $pos(u)$ denotes a position in T where the label of the incoming edge of u is spelled out. Hence, the label of an edge (u, v) is the string of length $depth(v) - depth(u)$ that begins at position $pos(v)$ of T.

By $child(u, c) = v$, where u and v are nodes and c is a character, we denote that there is an edge (u, v) whose label begins with c. We call c the *distinguishing character* of (u, v).

In order to express tree locations of strings that do not have a corresponding node in the suffix tree (due to path compression), we introduce the following concept: For each substring α of T we define $point(\alpha)$ as a triple (u, k, c), where u is the node of maximum depth for which $\langle u \rangle$ is a prefix of α, $k = |\alpha| - |\langle u \rangle|$, and c is the $(|\langle u \rangle| + 1)$th character of α, unless $k = 0$ in which case c can be any character. Less formally, if we traverse the tree from the root following edges that together spell out α for as long as possible, u is the last node on that path, k is the number of remaining characters of α, and c is the distinguishing character that determines which of the outgoing edges of u spells out the last part of α.

3 Suffix Tree Construction

We give a slightly altered, and highly condensed, formulation of Ukkonen's suffix tree construction algorithm as a basis for discussions in subsequent sections. For a more elaborate description, see Ukkonen's original paper [11]. Ukkonen's algorithm has the advantage over the more well known algorithm of McCreight [6] that it builds the tree incrementally left-to-right. This is essential for our application.

3.1 Preliminaries

At each internal node u of the suffix tree, the algorithm stores a *suffix link*, pointing to another internal node v, such that $\langle u \rangle = c\langle v \rangle$ (where c is the first character of $\langle u \rangle$). This is denoted $suf(u) = v$. For convenience, we add a special node nil and define $suf(root) = nil$, $parent(root) = nil$, $depth(nil) = -1$, and $child(nil, c) = root$ for any character c. We leave $suf(nil)$ undefined. Furthermore, for a node u that has no outgoing edge with distinguishing character c, we define $child(u, c) = nil$.

We denote the individual characters of T (the string to be indexed) by t_i, where $1 \leq i \leq n$, i.e., $T = t_1 \cdots t_n$. We define T_i as the prefix of T of length i, and let $Tree(T_i)$ denote a suffix tree indexing the string T_i.

3.2 Construction Algorithm

Ukkonen's algorithm is incremental. In iteration i we build $Tree(T_i)$ from $Tree(T_{i-1})$, and thus after n iterations we have $Tree(T_n) = Tree(T)$. Hence, iteration i adds i strings αt_i to the tree for all suffixes α of T_{i-1}. For each αt_i precisely one of the following holds:

1. α occurs in only one position in T_{i-1}. This implies that $\langle s \rangle = \alpha$ for some leaf s of $Tree(T_{i-1})$. In order to add αt_i we need only increment $depth(s)$.

2. α occurs in more than one position in T_{i-1}, but αt_i does not occur in T_{i-1}. This implies that a new leaf must be created for αt_i, and possibly an internal node has to be created as well, to serve as parent of that leaf.

3. αt_i occurs already in T_{i-1} and therefore is already present in $Tree(T_{i-1})$.

Observe that if (in a specific suffix tree), for a given t_i, case 1 holds for $\alpha_1 t_i$, case 2 for $\alpha_2 t_i$, and case 3 for $\alpha_3 t_i$, then α_1 is longer than α_2, which in turn is longer than α_3.

For case 1, all work can be avoided if we represent $depth(s)$ implicitly for all leaves s: We represent the leaves as numbers and let $leaf(j)$ be the leaf representing the suffix beginning at position j of T. This implies that if $s = leaf(j)$, then after iteration i we have $depth(s) = i - j + 1$ and $pos(s) = j - depth(parent(s))$. Hence, neither $depth(s)$ nor $pos(s)$ needs to be stored.

Now, the point of greatest depth where the tree may need to be altered in iteration i is $point(\alpha)$, where α is the longest suffix of T_{i-1} that also occurs in some other position in T_{i-1}. We call this the *active point*. Before the first iteration, the active point is $(root, 0, c)$, where c is any character.

Other points that need modification can be found from the active point by following suffix links, and possibly some downward edges. Finally, we reach the point that corresponds to the longest αt_i string for which case 3 above holds, which concludes iteration i. We call this the *endpoint*. The active point for the next iteration is found simply by moving one character (t_i) downward from the endpoint.

We maintain a variable $front$ that holds the position to the right of the string currently included in the tree. Hence, $front = i$ before iteration i, and $front = i + 1$ after.

Two variables ins and $proj$ are kept so that $(ins, proj, t_{front-proj})$ is the *insertion point*, the point where new nodes are inserted. At the beginning of each iteration, the insertion point is set to the active point. The $Canonize$ function in Figure 1 is used to ensure that $(ins, proj, t_{front-proj})$ is a valid point after $proj$ has been incremented, by moving ins along downward edges.

Figure 2 shows the complete procedure for one iteration of the construction algorithm. This takes constant amortized time, provided that the operation to retrieve $child(u, c)$ given u and c takes constant time. (Proof given by Ukkonen [11].) For most realistic alphabet sizes, this requires that a hash coding representation is used, as suggested by McCreight [6].

4 Maintaining a Sliding Window

In this section, we assume that the string to be indexed is $T = t_{tail} \cdots t_{front}$, where $tail$ and $front$ are numbers such that at any point in time $tail \leq front$ and $front - tail \leq M$ for some maximum length M. For convenience, we assume that $front$ and $tail$ may grow indefinitely. However, in practice the indices should be represented as integers modulo M, and T stored in a circular buffer. This means that, e.g., $t_i = t_{i+M}$ and $leaf(i) = leaf(i+M)$ for any i.

The algorithm of Figure 2 can be viewed as a method to increment $front$. Below, we give a method to increment $tail$ without asymptotic increase in time complexity. Thus, we can maintain a suffix tree as an index for a sliding window of varying size at most M, while keeping time complexity linear in the number of processed characters. The storage space requirement is $\Theta(M)$.

4.1 Deletions

Removing the leftmost character of the indexed string involves removing the longest suffix of T, i.e. T itself, from the tree. It is clear that $Tree(T)$ must have a leaf v such that $\langle v \rangle = T$. Also, it is clear that $v = leaf(tail)$. It therefore appears at first glance to be a simple task to locate v and remove it from the tree in the following way:

Delete algorithm: Let $v = leaf(tail)$, $u = parent(v)$ and remove the edge (u, v). If u has at least two remaining children, then we are done.

Otherwise, let s be the remaining child of u; u and s should be contracted into one node. Let $w = parent(u)$. Remove edges (w, u) and (u, s) and create an edge (w, s). u has now been removed from the tree and can be marked unused. Finally, if s is not a leaf, $pos(s)$ should be updated by subtracting $depth(u) - depth(w)$ from it. □

However, this is not sufficient for a correct $tail$ increment procedure. We must first ensure that the variables ins and $proj$ are kept valid. This is violated if the deleted node u is equal to ins. Fortunately, it is an easy matter to check whether $u = ins$, and if so, let ins back up by changing it to w and increasing $proj$ by $depth(u) - depth(v)$.

Secondly, we must ensure that no other suffix than the whole of T is removed from the tree. This is violated if T has a suffix α that is also a prefix of T, and if $point(\alpha)$ is located on the incoming edge of the removed leaf; in this case α is lost from the tree. A solution to this problem can be found in the following lemma:

Lemma 1 *Assume that:*

1. *T and α are nonempty strings;*
2. *α is the longest string such that $T = \delta\alpha = \alpha\theta$ for nonempty strings δ and θ;*
3. *if T has a suffix $\alpha\mu$ for some nonempty μ, then μ is a prefix of θ.*

Then α is the longest suffix of T that also occurs in some other position in T.

Proof: Trivially, by assumptions 1 and 2, α is a suffix of T that also occurs in some other position in T. Assume that it is not the longest one, and let $\chi\alpha$ be a longer suffix that occurs in some other position in T. This implies that $T = \phi\chi\alpha = \beta\chi\alpha\gamma$, for some nonempty strings ϕ, χ, β, and γ.

Since $\alpha\gamma$ is a suffix of T, it follows from assumption 3 that that γ is a prefix of θ. Hence, $\theta = \gamma\theta'$ for some string θ'. Now observe that $T = \alpha\theta = \alpha\gamma\theta'$. Letting $\alpha' = \alpha\gamma$ and $\delta' = \beta\chi$ then yields $T = \delta'\alpha' = \alpha'\theta'$, where $|\alpha'| > |\alpha|$, which contradicts assumption 2. □

Let α be the longest suffix that would be lost. (This guarantees that the premises of the lemma are fulfilled.) If we ensure that α is kept, no suffixes can be lost, since all potentially lost suffixes are prefixes of T and therefore also of α.

From Lemma 1 we conclude that α is the longest suffix of T that occurs in some other position in T. Hence, $point(\alpha)$ is the insertion point. Therefore, before we delete v, we call $Canonize$ and check whether its returned value is equal to v. If so, instead of deleting v, we replace it by $leaf(front - |\alpha|)$.

Finally, we must ensure that edge labels do not become out of date when $tail$ is incremented, i.e. that $pos(u_i) \geq tail$ for all internal nodes u_i. Since each leaf corresponds to a suffix of T (and thereby a string contained in T), traversing the tree to the root to update position values for each added leaf could take care of this. However, this would yield quadratic time complexity, so we must restrict the number of updates.

The following scheme originates from Fiala and Greene [5]. We let each leaf contribute a *credit* to the tree. When a leaf is added, it issues one credit to its parent. Each internal node u has a credit counter $cred(u)$ that is initially zero. If a node receives a credit when the counter is zero, it sets the counter to one. When a node with its counter set to one receives a credit, it sets the counter to zero and issues, to its parent, the one of the two received credits that originates from the most recent leaf.

While $proj > 0$, do
$r \leftarrow child(ins, t_{front-proj})$.
$d \leftarrow depth(r) - depth(ins)$
If r is a leaf or $proj < d$, then return r,
else $proj \leftarrow proj - d$, $ins \leftarrow r$.
Return nil.

Figure 1: $Canonize$ function.

$v \leftarrow nil$
Repeat
$r \leftarrow Canonize$.
If $r \neq nil$, then
If $t_{pos(r)+proj} = t_{front}$, then endpoint found,
else
Assign u an unused node.
$depth(u) \leftarrow depth(ins) + proj$.
$pos(u) \leftarrow front - proj$.
Create edges (ins, u) and (u, r).
Remove edge (ins, r).
If r is an internal node, then
$pos(r) \leftarrow pos(r) + proj$.
else
If $child(ins, t_{front}) = nil$, then $u \leftarrow ins$,
else endpoint found.
If not endpoint found, then
$s \leftarrow leaf(front - depth(u))$.
Create edge (u, s).
$suf(v) \leftarrow u$, $v \leftarrow u$, $ins \leftarrow suf(ins)$.
until endpoint found.
$suf(v) \leftarrow ins$.
$proj \leftarrow proj + 1$, $front \leftarrow front + 1$.

Figure 2: One iteration of construction.

$r \leftarrow Canonize$, $v \leftarrow leaf(tail)$.
$u \leftarrow parent(v)$, remove edge (u, v).
If $v = r$, then
$k \leftarrow front - (depth(ins) + proj)$.
Create edge $(ins, leaf(k))$.
$Update(ins, k)$, $ins \leftarrow suf(ins)$.
else
If u has only one remaining child, then
$w \leftarrow parent(u)$.
$d \leftarrow depth(u) - depth(w)$.
If $u = ins$, then
$ins \leftarrow w$, $proj \leftarrow proj + d$.
Assign s the child of u.
If $cred(u) = 1$, then
$Update(w, pos(s) - depth(u))$.
Remove edges (w, u) and (u, s).
Create edge (w, s).
If s is an internal node, then
$pos(s) \leftarrow pos(s) - d$.
Make a note that u is unused.
$tail \leftarrow tail + 1$.

Figure 3: Deletion.

While $u \neq root$, do
$v \leftarrow parent(u)$.
$k \leftarrow \max\{k, pos(u) - depth(v)\}$.
$pos(u) \leftarrow k + depth(v)$.
$cred(u) \leftarrow 1 - cred(u)$.
If $cred(u) = 1$, then return,
else, $u \leftarrow v$.

Figure 4: $Update(u, k)$.

If an internal node scheduled for removal holds one credit, that is issued to its parent when the node is removed. Each time a credit is passed to a node u, $pos(u)$ is updated. We define a *fresh credit* as a credit originating from one of the leaves currently present. Hence, if a node u has received a fresh credit, then $pos(u) \geq tail$

The following lemma states that this scheme guarantees valid edge labels. (Fiala and Greene [5] make an analogous statement.)

Lemma 2 *Each node has issued at least one fresh credit.*

Proof: Trivially, all nodes of maximum depth are leaves, and have issued fresh credits, originating from themselves. Assume that all nodes of depth k have issued fresh credits. Let u be an internal node of depth $k - 1$, then each child v of u has issued a fresh credit. This has either been passed on to u immediately or to a node between u and v which has later been removed. Since remaining credits are sent upwards from nodes that are removed, that credit or an even more recent credit must have propagated to u. Thus, u has received fresh credits from all its children (at least two), and must therefore have issued at least one fresh credit.

Consequently, nodes of all depths have issued fresh credits. □

Figure 3 shows the final algorithm for advancing $tail$, and Figure 4 shows the routine $Update(u, k)$ for passing credits upwards in the tree, where the parameter u is a node to which a credit is issued and k is the starting position of a recently added suffix that starts with $\langle u \rangle$. The algorithm to advance $front$ is as in Figure 2, with the following additions to supply credits from inserted leaves:

After line 10: $Update(ins, front - depth(u))$, $cred(u) \leftarrow 1$.
After line 19: $Update(u, front - depth(u))$.

We can now state the following:

Theorem *The presented algorithm correctly maintains a sliding window in time linear in the size of the input.*

The proof follows from Lemma 1 and 2 together with the previous discussions. The details are omitted. If the alphabet size is not regarded as a constant this bound requires that hash coding is used, i.e., it concerns randomized time.

5 Statistical Modeling

The most effective results in data compression have been achieved by statistical modeling in combination with arithmetic coding. Specifically, prediction by partial matching (PPM) is the scheme that has generated the most notable results during the last decade. The original PPM algorithm was given by Cleary and Witten [4]. Moffat's variant PPMC [7] offers a significant improvement.

The most prominent PPM variant with respect to compression performance is a variant named PPM*, given by Cleary, Teahan, and Witten [3]. However, as originally presented, this is hardly practical due to very large demands in computational resources. We show that PPM*-style methods can be implemented in linear time and in space bounded by an arbitrarily chosen constant, using the techniques of the previous sections.

5.1 PPM and PPMC

The idea of PPM is to regard the last few characters of the input stream as a *context*, and maintain statistical information about each context in order to predict the upcoming character. The number of characters used as a context is referred to as the *order*.

For each context, a table of character counts is maintained. When a character c appears in context C, the count for c in C is used to encode the character: the higher the count, the larger the code space allocated to it. The encoding is most effectively performed with arithmetic coding [8, 13]. When a character appears in a context for the first time, its count in that context is zero, and the character can not be encoded. Therefore each context also keeps an *escape* count, used to encode a new event in that context. When a new event occurs, the algorithm "falls back" to the context of nearest smaller order. A (-1)-order context, where all characters are assumed to be equally likely, is maintained for characters that have never occured in the input stream. New contexts are added as they occur in the input stream.

How and when to update escape counts is an intricate problem. Witten and Bell [12] consider several heuristics.

5.2 PPM*

Previous to PPM*, the maximum order has usually been set to some small number. This is primarily to keep the number of states from growing too large, but also a decrease in compression performance can be observed when the order is allowed to grow large (to more than about six). This is because large-order contexts make the algorithm less stable; the chance of the current context not having seen the upcoming character is larger. However, the performance of PPM* demonstrates that with a careful strategy of choosing contexts, allowing the order to grow without bounds can yield a significant improvement.

In PPM* all substrings that have occured in the input stream are stored in a trie and each node in the trie is a context. A *context list* is maintained, holding all the contexts that match the last part of the input stream. Among these, the context to use for encoding is chosen. (Using the strategy of earlier PPM variants, the context to use would be the one of highest order, i.e. the node with greatest depth, but Cleary, Teahan, and Witten argue that this is not the best strategy for PPM*.) The tree is updated by traversing the context list, adding nodes where necessary. Escaping is also performed along the context list.

5.3 Using a Suffix Tree

Cleary, Teahan, and Witten observe that collapsing paths of unary nodes into single nodes, i.e. path compression, can save substantial space. We make some further observations that lead us to the conclusion that the suffix tree operations described in previous sections are suitable to maintain the data structure for a PPM* model.

1. A context trie is equivalent to a suffix tree. A path-compressed context trie *is* a suffix tree indexing the processed part of the input, according to the definition in Section 2.

2. Context list. The context list of the PPM* scheme corresponds to a chain of nodes in the suffix tree connected by suffix links. Using suffix links, it is not necessary to maintain a separate context list.

3. Storing the counts. The characters that have non-zero counts in a context are exactly the ones for which the the corresponding node in the trie has children. Hence, if $child(u, c) = v$, the count for character c in context u can be stored in v. There is no need for additional tables of counts for the contexts.

Cleary, Teahan, and Witten state that path compression complicates node storage, since different contexts may belong to the same node. However, if two strings (contexts) belong to the same node, this implies that one is the prefix of the other, and that there are no branches between them. Hence, they have always appeared in the input stream with one as the prefix of the other. Therefore, it appears that the only reasonable strategy is to let them have the same count, i.e., only one count needs to be stored.

We conclude that PPM modeling with unbounded context length for an input of size n can be done in time $O(n + U(n))$, where $U(n)$ is the time used for updating frequency counts and choosing states among the nodes. Thus, we are in a position where asymptotic time complexity depends solely on the count updating strategy. Furthermore, restraining update and state choice to a constant number of operations per visited node (amortized) appears to be a reasonable limitation.

5.4 Sliding Window

Using a suffix tree that is allowed to grow without bounds, until it covers the whole input, is still not a practical method. For large files, primary storage can not even hold the complete file, let alone a suffix tree for that file.

Clearly, it is necessary to bound the size of the data structure. We do that by letting the suffix tree index only the last M characters of the input, using the sliding window techniques of Section 4. In this way, contexts corresponding to strings occuring in the latest M characters are always maintained, while older contexts are “forgotten.” Note, however, that we do not lose all earlier information when deleting old contexts, since the counts of remaining contexts are influenced by earlier parts of the input.

Preliminary experiments indicate that this scheme, already with a simplistic strategy for updating and choosing states (choosing the active point as current state and updating nodes only when used), yield highly competitive results—e.g. an improvement over PPMC in most cases.

6 Ziv–Lempel Compression

Methods based on the original algorithm of Ziv and Lempel [14] operate by storing the latest part (typically several thousand characters) of the input seen so far, and for each iteration finding the longest match for the upcoming part of the input. It then emits the position in the buffer of that matching string, together with its length. If no match is found, the character is transferred explicitly.

The main part of Ziv–Lempel compression consists of searching for the longest matching string. This can be done in asymptotically optimal time by maintaining a suffix tree to index the buffer. For each iteration, the longest match can be located by traversing the tree from the root, following edges corresponding to characters from the input stream.

Rodeh and Pratt [9] show that it is possible to implement a linear-time Ziv–Lempel algorithm utilizing a suffix tree. However, since they do not allow deletions from the suffix tree, their algorithm requires $\Omega(n)$ space to process an input of length n. This implies that large inputs must be split into blocks, which decreases compression performance.

Our sliding window technique yields a natural implementation of Ziv–Lempel compression, which operates in linear time.

7 Conclusion

We conclude that, using our algorithm, PPM modeling with unbounded context length for an input of size n can be done in time $O(n + U(n))$, where $U(n)$ is the time used for updating frequency counts and choosing states among the nodes. Thus, asymptotic time complexity depends solely on the count updating strategy. Space requirements can be bounded by a constant, which is chosen depending on the application and available resources. Preliminary experiments indicate that this yields highly competitive results.

Furthermore, with our sliding window technique we obtain a natural implementation of Ziv–Lempel compression which runs in linear time.

It has been noted, e.g. by Bell and Witten [2], that there is a strong connection between string substituting compression methods and symbolwise (statistical) methods. Our assertion that the exact same data structure is useful in both these families of algorithms serves as a further illustration of this.

References

[1] T. Bell and D. Kulp. Longest-match string searching for Ziv–Lempel compression. *Software —Practice and Experience*, 23(7):757–771, July 1993.

[2] T. C. Bell and I. H. Witten. The relationship between greedy parsing and symbolwise text compression. *J. ACM*, 41(4):708–724, July 1994.

[3] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. In *Proc. IEEE Data Compression Conference*, pages 52–61, 1995.

[4] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, COM-32:396–402, Apr. 1984.

[5] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, Apr. 1989.

[6] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[7] A. Moffat. Implementing the PPM data compression scheme. *IEEE Trans. Commun.*, COM-38(11):1917–1921, Nov. 1990.

[8] A. Moffat, R. Neal, and I. H. Witten. Arithmetic coding revisited. In *Proc. IEEE Data Compression Conference*, pages 202–211, 1995.

[9] M. Rodeh and V. R. Pratt. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, Jan. 1981.

[10] W. J. Teahan. Probability estimation for PPM. In *Proc. Second New Zealand Comput. Sci. Research Students' Conference*, Apr. 1995.

[11] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, Sept. 1995.

[12] I. H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. Inf. Theory*, IT-37(4):1085–1094, July 1991.

[13] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, June 1987.

[14] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23(3):337–343, May 1977.