ATTACK OF THE MUTANT SUFFIX TREES

N. JESPER LARSSON

ABSTRACT. This is a thesis for the degree of *filosofie licentiat* (a Swedish degree between Master of Science and Ph.D.). It comprises three articles, all treating variations and augmentations of suffix trees, and the capability of the suffix tree data structure to efficiently capture similarities between different parts of a string. The presented applications are in the areas of data compression and pattern matching.

The first article presents a method of using a suffix tree as an index into a sliding window, in optimal time and space, with applications in probabilistic data compression schemes, such as PPM, as well as in Ziv-Lempel compression. The second article presents a generalized suffix tree which is able to index a select subset of the suffixes of a string, with a construction algorithm which is optimal in time and space. The final article deals with the connection between the explicit context trees of PPM and the implicit mechanism of the Burrows-Wheeler transform for block sorting compression, again utilizing the suffix tree to capture the repetitive characteristics of a string.

This document was revised on February 3 1998. The differences from the presented thesis are in corrections of trivial errors only.

Preface

This is a thesis for the degree of *filosofie licentiat*, a Swedish degree between Master of Science and Ph.D. It consists of a general introduction and three articles, which differ only in minor corrections and clarifications from the following publications:

1. N. Jesper Larsson, *Extended application of suffix trees to data compression*, Proceedings of the IEEE Data Compression Conference, March–April 1996, pp. 190–199.
2. Arne Andersson, N. Jesper Larsson, and Kurt Swanson, *Suffix trees on words*, Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 1075, Springer-Verlag, June 1996, pp. 102–115.
3. N. Jesper Larsson, *The context trees of block sorting compression*, to appear, Proceedings of the IEEE Data Compression Conference, March–April 1998.

Acknowledgments

I wish to thank my advisor Dr. Arne Andersson for his contributions, his help and encouragement—and also for allowing me substantial freedom in pursuing my own ideas; Kurt Swanson for collaboration in research as well as everyday matters; Dr. Stefan Nilsson for valuable comments and support during my first year of graduate studies; and the rest of my colleagues at the Department of Computer Science for contributing to a fertile research environment. Furthermore, I wish to thank Jan Åberg for many fruitful discussions about data compression.

Introduction

The *suffix tree* is a powerful data structure for many operations on strings, particularly useful for many applications since its construction can be accomplished in time that does not asymptotically exceed the time required to read the input. Its most elementary application is as an index which supports locating any substring of a longer string in time proportional to the length of the substring. Some of its more intricate applications have been surveyed by Apostolico [2].

The first linear-time suffix tree construction algorithm was presented in 1973 by Weiner [14]. Shortly thereafter, McCreight [11] presented a simpler and less space consuming construction algorithm which has been the standard since. Also notable are Ukkonen's on-line construction algorithm [13], as well as Farach's algorithm [7], which provides deterministic optimal time construction for large alphabets, and whose approach is quite different from that of the previous algorithms.

The efficiency at which the suffix tree supports access to repetitions of a string makes its application in data compression natural. The first article of this thesis [9] contributes to this area of application by presenting a sliding-window indexing algorithm that is an improvement over previous attempts [8, 12] as it is fully on-line and linear-time, and implements—as a single data structure—an index for a sliding window of variable size.

Application of this data structure to Ziv-Lempel compression [15] is natural and straightforward, as this family of compression algorithms is often formulated in a sliding window context. More elaborate is the application of this scheme to the probabilistic compression method family of PPM [6], more specifically PPM* [5]. Using the presented sliding window index, a PPM*-style algorithm can run in linear time, and in space limited by an arbitrarily chosen window size. The suffix tree approach to PPM compression has since the publication of this work been further developed by Bunton [3] (although not with as strict complexity considerations).

All the mentioned suffix tree construction algorithms rely heavily on *all* suffixes being inserted, i.e., all substrings are considered equally relevant. This does not apply to all uses of suffix trees for indexing strings: often one is interested only in substrings that begin at certain positions, *word boundaries*, of the text. The second article in this thesis [1] presents a data structure, the *word suffix tree*, whose construction space is proportional only to the number of words, while linear construction time is maintained. This article also illustrates the possibility of sublinear construction time in cases where the input string occupies $o(n)$ machine words.

The third article [10] discusses the *Burrows-Wheeler transform* (BWT) and *block sorting compression* [4]. The BWT procedure is intended for rearranging the input string so that repetitions are concentrated, as a first step in compression. The transform can be efficiently implemented using a suffix tree. In this article, the suffix tree is considered as the link between BWT and the *context tree* of the previously mentioned PPM* compression method. An efficient method of generating an explicit *pruned context tree* from the corresponding suffix tree is presented. Thus, the context tree model, which is otherwise implicitly assumed in BWT compression, is given

a concrete form. The relationship between the context tree and the *move-to-front* schemes commonly paired with BWT is considered in some detail, and an experimental compression program which utilizes the full context tree is presented. The conclusion is that a conscious treatment of the context tree makes a combination of BWT with some of the ideas from PPM a strong candidate for powerful general compression with moderate requirements on computational resources.

References

1. Arne Andersson, N. Jesper Larsson, and Kurt Swanson, *Suffix trees on words*, Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 1075, Springer-Verlag, June 1996, pp. 102–115.
2. Alberto Apostolico, *The myriad virtues of subword trees*, Combinatorial Algorithms on Words, NATO ISI Series (Alberto Apostolico and Zvi Galil, eds.), Springer-Verlag, 1985, pp. 85–96.
3. Suzanne Bunton, *On-line stochastic processes in data compression*, Ph.D. thesis, Department of Computer Science and Engineering, University of Washington, Seattle, December 1996.
4. Michael Burrows and David J. Wheeler, *A block-sorting lossless data compression algorithm*, Research Report. 124, Digital Systems Research Center, Palo Alto, California, May 1994.
5. John G. Cleary, W. J. Teahan, and Ian H. Witten, *Unbounded length contexts for PPM*, Proceedings of the IEEE Data Compression Conference, March 1995, pp. 52–61.
6. John G. Cleary and Ian H. Witten, *Data compression using adaptive coding and partial string matching*, IEEE Transactions on Communications **COM-32** (1984), 396–402.
7. Martin Farach, *Optimal suffix tree construction with large alphabets*, Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, October 1997, pp. 137–143.
8. Edward R. Fiala and Daniel H. Greene, *Data compression with finite windows*, Communications of the ACM **32** (1989), no. 4, 490–505.
9. N. Jesper Larsson, *Extended application of suffix trees to data compression*, Proceedings of the IEEE Data Compression Conference, March–April 1996, pp. 190–199.
10. ———, *The context trees of block sorting compression*, to appear, Proceedings of the IEEE Data Compression Conference, March–April 1998.
11. Edward M. McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM **23** (1976), no. 2, 262–272.
12. Micael Rodeh, Vaughan R. Pratt, and Shimon Even, *Linear algorithm for data compression via string matching*, Journal of the ACM **28** (1981), no. 1, 16–24.
13. Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.
14. Peter Weiner, *Linear pattern matching algorithms*, Proceedings of the 14th Annual IEEE Symposium on Foundations of Computer Science, 1973, pp. 1–11.
15. Jacob Ziv and Abraham Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory **IT-23** (1977), no. 3, 337–343.

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 input right-to-left (i.e. longest suffix first),

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 $\mathit{depth}(u)$ 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 $\mathit{pos}(u)$ and $\mathit{depth}(u)$, where $\mathit{pos}(u)$ 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 $\mathit{depth}(v) - \mathit{depth}(u)$ that begins at position $\mathit{pos}(v)$ 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.

```
1 While proj > 0, do
2    r ← child(ins, t_{front−proj}).
3    d ← depth(r) − depth(ins)
4    If r is a leaf or proj < d, then return r,
5    else proj ← proj − d, ins ← r.
6 Return nil.
```

FIGURE 1. *Canonize* function.

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 $T = t_{tail} \dots t_{front}$, 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

```
1  v ← nil
2  Repeat
3     r ← Canonize.
4     If r ≠ nil, then
5        If t_{pos(r)+proj} = t_front, then endpoint found,
6        else
7           Assign u an unused node.
8           depth(u) ← depth(ins) + proj.
9           pos(u) ← front − proj.
10          Create edges (ins, u) and (u, r).
11          Remove edge (ins, r).
12          If r is an internal node, then
13             pos(r) ← pos(r) + proj.
14    else
15       If child(ins, t_front) = nil, then u ← ins,
16       else endpoint found.
17    If not endpoint found, then
18       s ← leaf(front − depth(u)).
19       Create edge (u, s).
20       suf(v) ← u, v ← u, ins ← suf(ins).
21 until endpoint found.
22 suf(v) ← ins.
23 proj ← proj + 1, front ← front + 1.
```

FIGURE 2. One iteration of construction.

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 $\mathit{Tree}(T)$ must have a leaf v such that $\langle v \rangle = T$. Also, it is clear that $v = \mathit{leaf}(\mathit{tail})$. It therefore appears at first glance to be a simple task to locate v and remove it from the tree in the following way:

> Let $v = \mathit{leaf}(\mathit{tail})$, $u = \mathit{parent}(v)$ 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 $w = \mathit{parent}(u)$. Remove the 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, $\mathit{pos}(s)$ should be updated by subtracting $\mathit{depth}(u) - \mathit{depth}(w)$ 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 $u = \mathit{ins}$, and if so, let *ins* back up by changing it to w and increasing *proj* by $\mathit{depth}(u) - \mathit{depth}(v)$.

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 $\mathit{point}(\alpha)$ 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 some 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 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, $\mathit{point}(\alpha)$ 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 $\mathit{leaf}(\mathit{front} - |\alpha|)$.

Finally, we must ensure that edge labels do not become out of date when *tail* is incremented, i.e. that $\mathit{pos}(u_i) \geq \mathit{tail}$ 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 $\mathit{cred}(u)$ 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.

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, $\mathit{pos}(u)$ 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 $\mathit{pos}(u) \geq \mathit{tail}$

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

```
1  r ← Canonize, v ← leaf(tail).
2  u ← parent(v), remove edge (u, v).
3  If v = r, then
4     k ← front − (depth(ins) + proj).
5     Create edge (ins, leaf(k)).
6     Update(ins, k), ins ← suf(ins).
7  else
8     If u has only one remaining child, then
9        w ← parent(u).
10       d ← depth(u) − depth(w).
11       If u = ins, then
12          ins ← w, proj ← proj + d.
13       Assign s the child of u.
14       If cred(u) = 1, then
15          Update(w, pos(s) − depth(u)).
16       Remove edges (w, u) and (u, s).
17       Create edge (w, s).
18       If s is an internal node, then
19          pos(s) ← pos(s) − d.
20       Make a note that u is unused.
21 tail ← tail + 1.
```

FIGURE 3. Deletion.

```
1  While u ≠ root, do
2     v ← parent(u).
3     k ← max{k, pos(u) − depth(v)}.
4     pos(u) ← k + depth(v).
5     cred(u) ← 1 − cred(u).
6     If cred(u) = 1, then return,
7     else, u ← v.
```

FIGURE 4. *Update*(u, 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 1. *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 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 a 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, Pratt, and Even [9] show that it is possible to implement a linear-time Ziv-Lempel algorithm utilizing a suffix tree. However, with 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. To overcome this, they employ three coexisting suffix trees for overlapping sections of the input to emulate a sliding window index.

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. Timothy Bell and David Kulp, *Longest-match string searching for Ziv–Lempel compression*, Software—Practice and Experience **23** (1993), no. 7, 757–771.
2. Timothy C. Bell and Ian H. Witten, *The relationship between greedy parsing and symbolwise text compression*, Journal of the ACM **41** (1994), no. 4, 708–724.
3. John G. Cleary, W. J. Teahan, and Ian H. Witten, *Unbounded length contexts for PPM*, Proceedings of the IEEE Data Compression Conference, March 1995, pp. 52–61.
4. John G. Cleary and Ian H. Witten, *Data compression using adaptive coding and partial string matching*, IEEE Transactions on Communications **COM-32** (1984), 396–402.
5. Edward R. Fiala and Daniel H. Greene, *Data compression with finite windows*, Communications of the ACM **32** (1989), no. 4, 490–505.
6. Edward M. McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM **23** (1976), 262–272.
7. Alistair Moffat, *Implementing the PPM data compression scheme*, IEEE Transactions on Communications **COM-38** (1990), no. 11, 1917–1921.
8. Alistair Moffat, Radford Neal, and Ian H. Witten, *Arithmetic coding revisited*, Proceedings of the IEEE Data Compression Conference, 1995, pp. 202–211.
9. Micael Rodeh, Vaughan R. Pratt, and Shimon Even, *Linear algorithm for data compression via string matching*, Journal of the ACM **28** (1981), no. 1, 16–24.
10. W. J. Teahan, *Probability estimation for PPM*, Proceedings of the 2nd New Zealand Computer Science Research Students' Conference, April 1995.
11. Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.
12. Ian H. Witten and Timothy C. Bell, *The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression*, IEEE Transactions on Information Theory **IT-37** (1991), no. 4, 1085–1094.
13. Ian H. Witten, Radford M. Neal, and John G. Cleary, *Arithmetic coding for data compression*, Communications of the ACM **30** (1987), no. 6, 520–540.
14. Jacob Ziv and Abraham Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory **IT-23** (1977), no. 3, 337–343.

SUFFIX TREES ON WORDS

ARNE ANDERSSON, N. JESPER LARSSON, AND KURT SWANSON

Abstract. We discuss an intrinsic generalization of the suffix tree, designed to index a string of length n which has a natural partitioning into m multi-character substrings or *words*. This *word suffix tree* represents only the m suffixes that start at word boundaries. These boundaries are determined by *delimiters*, whose definition depends on the application.

Since traditional suffix tree construction algorithms rely heavily on the fact that *all* suffixes are inserted, construction of a word suffix tree is nontrivial, in particular when only $O(m)$ construction space is allowed. We solve this problem, presenting an algorithm with $O(n)$ expected running time. In general, construction cost is $\Omega(n)$ due to the need of scanning the entire input. In applications that require strict node ordering, an additional cost of sorting $O(m')$ characters arises, where m' is the number of distinct words. In either case, this is a significant improvement over previously known solutions.

Furthermore, when the alphabet is small, we may assume that the n characters in the input string occupy $o(n)$ machine words. We illustrate that this can allow a word suffix tree to be built in sublinear time.

1. Introduction

The *suffix tree* [18] is a very important and useful data structure for many applications [6]. Traditional suffix tree construction algorithms rely heavily on the fact that *all* suffixes are inserted, in order to obtain efficient time bounds. Little work has been done for the common case where only certain suffixes of the input string are relevant, despite the savings in storage and processing times that are to be expected from only considering these suffixes.

Baeza-Yates and Gonnet [7] have pointed out this possibility, by suggesting inserting only suffixes that start with a word, when the input consists of ordinary text. They imply that the resulting tree can be built in $O(n\mathcal{H}(n))$ time, where $\mathcal{H}(n)$ denotes the height of the tree, for n characters. While the expected height is logarithmic under certain assumptions [16], it is unfortunately linear in the worst case, yielding an algorithm that is quadratic in the size of the input.

One important advantage of this strategy is that it requires only $O(m)$ space for m words. Unfortunately, with a straightforward approach such as that of the aforementioned algorithm, this is obtained at the cost of a greatly increased time complexity. We show that this is an unnecessary tradeoff.

We formalize the concept of words to suit various applications and present a generalization of suffix trees, which we call *word suffix trees*. These trees store, for a string of length n in an arbitrary alphabet, only the m suffixes that start at word boundaries. Linear construction time is maintained, which in general is optimal, due to the requirement of scanning the entire input.

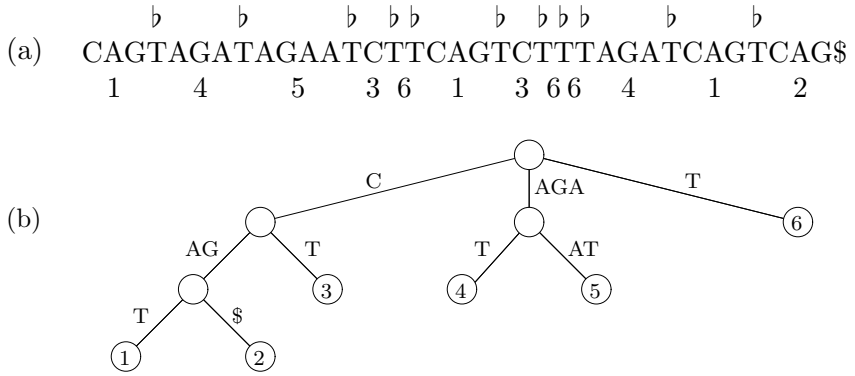

FIGURE 1. A sample string where ♭ = T, and its number string (a), created from its corresponding word trie (b).

Definitions and Main Results. We study the following formal problem: We are given an input string consisting of n characters from an alphabet of size k, including two, possibly implicit, special characters \$ and ♭. The \$ character is an end marker which must be the last character of the input string and may not appear elsewhere, while ♭ represents some *delimiting character* which appears in $m - 1$ places in the input string. We regard the input string as a series of *words*—the m non-overlapping substrings ending either with ♭ or \$. There may of course exist multiple occurrences of the same word in the input string. We denote the number of *distinct* words by m'. For convenience, we have chosen to regard each ♭ or \$ character as being contained in the preceding word. This implies that there are no empty words; the shortest possible word is a single ♭ or \$. The goal is to create a trie structure containing m strings, namely the suffixes of the input string that start at the beginning of words.

Figures 1 and 2 constitute an example where the input consists of a DNA sequence, and the character T is viewed as the word delimiter. (This is a special example, constructed for illustrating the algorithm, *not* a practical case.) Figure 2b shows the word suffix tree for the string shown in Figure 1a. These figures are more completely explained throughout this article.

Our definition can be generalized in a number of ways to suit various practical applications. The ♭ character does not necessarily have to be a single character, we can have a *set* of delimiting characters, or even sets of delimiting *strings*, as long as the delimiters are easily recognizable.

We define a *lexicographic trie* as a trie for which the strings represented at the leaves appear in lexicographical order in an in-order traversal. A *non-lexicographic* trie is not guaranteed to have this property. We now list the main results of this article:

- A word suffix tree for an input string of size n containing m words can be built in $O(n)$ expected time and $O(m)$ deterministic space.
- A lexicographic word suffix tree for an input string of size n containing m words of which m' are distinct can be built in $O(n + s(m'))$ expected

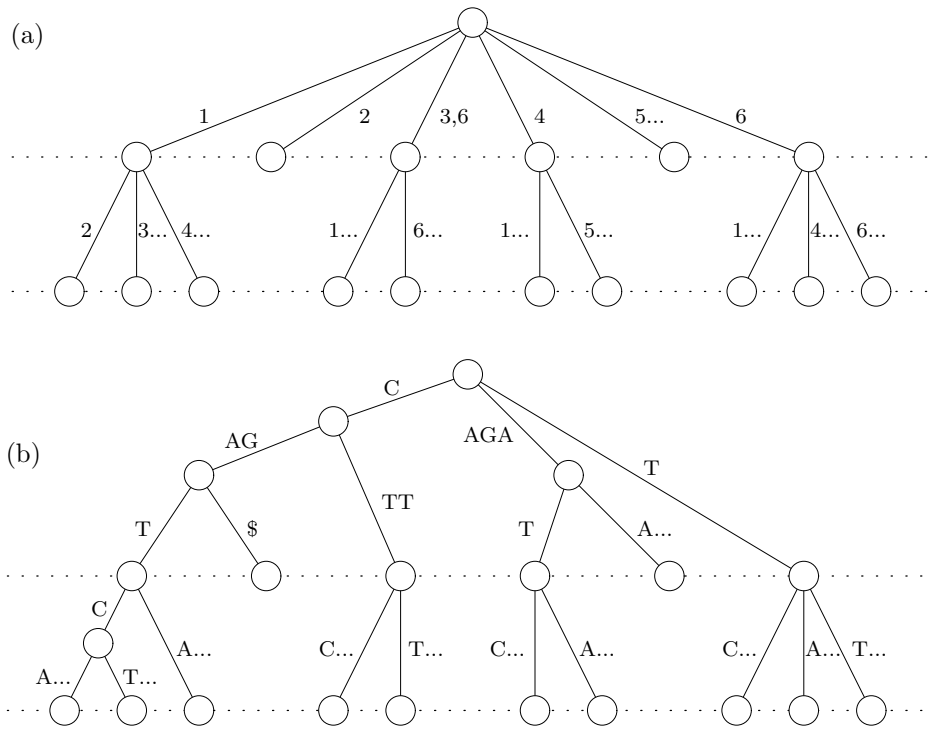

FIGURE 2. The number suffix tree (a) and its expansion into the final word suffix tree (b). (Dotted lines denote corresponding levels.)

time and $O(m)$ deterministic space, where $s(m')$ is the time required to sort m' characters.

- A lexicographic word suffix tree for an input string of size n containing m words of which m' are distinct can be built deterministically in $O(n+i(m,m'))$ time and $O(m)$ space, where $i(m,m')$ is the time required to insert m characters into ordered dictionaries each bounded in size by m'.

In addition, we obtain the following results, which show that the seemingly inherent lower bound of $\Omega(n)$ construction time can in some cases be surpassed when constructing word suffix trees:

- When the positions of all delimiters are known, a lexicographic word suffix tree on a string comprising m words of which m' are distinct, can be constructed in time

$$O\left(\frac{N}{b} + m + s_b(m')\right)$$

for some integer parameter $b \leq w$, where N is the number of bits in the input, w is the machine word length, and $s_b(m')$ is the time to sort m' b-bit integers.

- Given a Huffman coded input string of n characters coded in N bits where the alphabet size k satisfies $k = O(\sqrt{n}/\log^2 n)$, a word suffix

tree on m natural words can be constructed in time

$$O\left(\frac{N}{\log n} + m\right),$$

with construction space $O(m + \sqrt{n})$.

2. Data Structure Considerations

All tries discussed (the word suffix tree as well as some temporary tries) are assumed to be path compressed, i.e., they contain no nodes with out-degree one (other than possibly the root), and the edges of the trie are labeled with strings rather than characters. In order to reduce space requirements, these strings are represented by pointers into the original string. Thus, a trie with m leaves occupies $\Theta(m)$ space.

When choosing a trie implementation, it is important to be aware of which types of queries are expected. One important concept is the ordering of the nodes. Maintaining a lexicographic trie may be useful in some applications, e.g. to facilitate neighbor and range search operations. Note, however, that in many applications the alphabet is merely an arbitrarily chosen enumeration of unit entities with no tangible interpretation of *range* or *neighbor*, in which case a lexicographic trie has no advantage over its non-lexicographic counterpart.

Most of the work done on suffix tree construction seems to assume that a suffix tree is implemented as a lexicographic trie. However, it appears that the majority of the applications of suffix trees, for example all those given by Apostolico [6], do not require a lexicographic trie. Indeed, in his classic article on suffix tree construction, McCreight [15] concludes that the use of hash coding, which implies a non-lexicographic trie, appears to be the best representation.

Because of the factors of different applications, it is necessary to discuss several versions of tries. We consider specifically the following possibilities:

1. Each node is implemented as an array of size k. This allows fast searches, but consumes a lot of space for large alphabets.
2. Each node is implemented as a linked list or, preferably, as a binary search tree. This saves space at the price of a higher search cost, when the alphabet is not small enough to be regarded as constant.
3. The edges at each node are hash-coded (randomized construction). Using dynamic perfect hashing [8], we are guaranteed that searches spend constant time per node, even for a non-constant alphabet. Furthermore, this representation may be combined with variant 2.
4. Instead of storing pointers into the input string for each edge, the pointers are stored only at the leaves, and the character distinguishing each edge is stored explicitly. Thereby, the input string is not accessed while traversing internal nodes during search operations. Hence, an input string stored in secondary memory needs to be accessed only once per search operation.

In the following sections, we assume that the desired data structure is a *non-lexicographic* trie and that a randomized algorithm is satisfactory, except where otherwise stated. This makes it possible to use hash coding to

represent trees all through the construction. However, we also discuss the creation of lexicographic suffix trees, as well as deterministic construction algorithms.

An important fact is that a non-lexicographic trie can be made lexicographic at low cost by sorting all edges according to the first character of each edge, and then rebuilding the tree in the sorted order. We state this in the following observation:

Observation 1. *A non-lexicographic trie with l leaves can be made lexicographic in time $O(l+s(l))$, where $s(l)$ is the time required to sort l characters.*

3. Wasting Space: Algorithm A

We first observe the possibility of creating a word suffix tree from a traditional $\Theta(n)$ size suffix tree. This is relatively straightforward. Delimiters are not necessary when this method is used—the suffixes to be represented can be chosen arbitrarily. Unfortunately however, the algorithm requires much extra space during construction.

We refer to the procedure as *Algorithm A*:

1. Build a traditional *non-lexicographic* suffix tree for the input string in $O(n)$ time with a traditional algorithm [15, 17], using hashing to store edges.
2. Refine the tree into a word suffix tree: remove the leaves that do not correspond to any of the desired suffixes, and perform explicit path compression. The time for this is bounded by the number of nodes in the original tree, i.e. $O(n)$.
3. If so desired, make the trie lexicographic in time $O(m + s(m))$ (by Observation 1), where $s(m)$ denotes the time to sort m characters.

If the desired final result is a non-lexicographic tree, the construction time is $O(n)$, the same as for a traditional suffix tree. If a sorted tree is desired however, we have an improved time bound of $O(n + s(m))$ compared to the $\Theta(n + s(n))$ time required to create a lexicographic traditional suffix tree on a string of length n. (Note that while Farach [9] obtains linear construction time, his approach requires sorting as a preparatory step.) We state this in the following observation:

Observation 2. *A word suffix tree on a string of length n with m words can be created in $O(n)$ time, using $O(n)$ space, and made lexicographic at an extra cost of $O(m+s(m))$, where $s(m)$ denotes the time to sort m characters.*

The disadvantage of Algorithm A is that it consumes as much space as traditional suffix tree construction. Even the most space-economical implementation of Ukkonen's or McCreight's algorithm requires several values per node in the range $[0, n]$ to be held in primary storage during construction, in addition to the n characters of the string. While this is infeasible in many cases, it may well be possible to store the final word suffix tree of size $\Theta(m)$.

4. Saving Space: Algorithm B

In this section we present *Algorithm B*, the main word suffix tree construction algorithm, which in contrast to Algorithm A uses only $\Theta(m)$ space.

The algorithm is outlined as follows: First, a non-lexicographic trie with m' leaves is built, containing all distinct words: the *word trie*. Next, this trie is traversed and each leaf (corresponding to each distinct word in the input string) is assigned its in-order number. Thereafter, the input string is used to create a string of m numbers by representing every word in the input by its in-order number in the word trie. A *lexicographic* suffix tree is constructed for this string. This number-based suffix tree is then expanded into the final non-lexicographic word suffix tree, utilizing the word trie.

Below, we discuss the stages in detail.

1. Building the word trie. We employ a recursive algorithm to create a non-lexicographic trie containing all distinct words. Since the delimiter is included at the end of each word, no word can be a prefix of another. This implies that each word will correspond to a leaf in the word trie. We use hash coding for storing the outgoing edges of each node. The construction is performed top-down by the following algorithm, beginning at the root, which initially contains all words:

1. If the current node contains only one word, return.
2. Set the variable i to 1.
3. Check if all contained words have the same ith character. If so, increment i by one, and repeat this step.
4. Let the incoming edge to the current node be labeled with the substring consisting of the $i-1$ character long common prefix of the words it contains. If the current node is the root, and $i > 1$, create a new, unary, root above it.
5. Store all distinct ith characters in a hash table. Construct children for all distinct ith characters, and split the words, with the first i characters removed, among them.
6. Apply the algorithm recursively to each of the children.

Each character is examined no more than twice, once in step 3 and once in step 5. For each character examined, steps 3 and 5 perform a constant number of operations. Furthermore, steps 2, 4, and 6 take constant time and are performed once per recursive call, which is clearly less than n. Thus, the time for construction is $O(n)$.

2. Assigning in-order numbers. We perform an in-order traversal of the trie, and assign the leaves increasing numbers in the order they are visited, as shown in Figure 1. At each node, we take the order of the children to be the order in which they appear in the hash table. It is crucial for the correctness of the algorithm (Stage 5), that the following property holds:

Definition 1. *An assignment of numbers to strings is* semi-lexicographic *if and only if for all strings, α, β, and γ, where α and β have a common prefix that is not also a prefix of γ, the number assigned to γ is either less or greater than both numbers assigned to α and β.*

For an illustration of this, consider Figure 1b. The requirement that the word trie is semi-lexicographic ensures that consecutive numbers are assigned to the strings AGAT and AGAAT, since these are the only two strings with the prefix AGA.

The time for this stage is the same as for an in-order traversal of the word trie, which is clearly $O(m')$, where $m' \leq m$ is the number of distinct words.

3. Generating a number string. In this stage, we create a string of length m in the alphabet $\{1, \ldots, m'\}$.

This is done in $O(n)$ time by scanning the original string while traversing the word trie, following edges as the characters are read. Each time a leaf is encountered, its assigned number is output, and the traversal restarts from the root.

4. Constructing the number-based suffix tree. In this stage we create a traditional *lexicographic* suffix tree from the number string. For this, we use an ordinary suffix tree construction algorithm, such as McCreight's [15] or Ukkonen's [17]. Edges are stored in a hash table. The time needed for this is $O(m)$.

Since hash coding is used, the resulting trie is non-lexicographic. However, it follows from Observation 1 that it can be made lexicographic in $O(m)$ time using bucket sorting. In the lexicographic trie, we represent the children at each node with linked lists, so that the right sibling of a node can be accessed in constant time.

As an alternative, the suffix tree construction algorithm of Farach [9] can be used to construct this lexicographic suffix tree directly in $O(m)$ time, which eliminates the randomization element of this stage.

5. Expanding the number-based suffix tree. In this stage, each node of the number-based suffix tree is replaced by a local trie, containing the words corresponding to the children of that node. First, we preprocess the word trie for least common ancestor retrieval in $O(m')$ time, using for example the method of Harel and Tarjan [12]. This allows lowest common ancestors to be obtained in constant time. The local tries are then built left-to-right, using the fact that since the assignment of numbers to words is semi-lexicographic and the number-based suffix tree is lexicographic, each local trie has the essential structure of the word trie with some nodes and edges removed. We find the lowest common ancestor of each pair of adjacent children in the word trie, and this gives us the appropriate insertion point (where the two words diverge) of the next node directly.

More specifically, after preprocessing for computation of lowest common ancestors, we build the local trie at each node in the following manner. (The node expansion is illustrated in Figure 2a–b.)

1. Insert the first word.
2. Retrieve the next word in left-to-right order from the sorted linked list of children. Compute the lowest common ancestor of this word and the previous word in the word trie.
3. Look into the partially built trie to determine where the lowest common ancestor of the two nodes should be inserted, if it is not already there. This is done by searching up the tree from the last inserted word until reaching a node that has smaller height within the word trie.
4. If necessary, insert the internal (lowest common ancestor) node, and insert the leaf node representing the word.

5. Repeat from step 2 until all children have been processed.
6. If the root of the local trie is unary, remove it via path compression.

Steps 1 and 6 take constant time, and are executed once per internal node of the number-based suffix tree. This makes a total of $O(m')$ time for these steps. Steps 2, 4 and 5 also take constant time, and are executed once per node in the resulting word suffix tree. This implies that their total cost is $O(m)$. The total work performed in Step 3 is essentially an in-order traversal of the local subtree being built. Thus, the total time for Step 3 is proportional to the total size of the final tree, which is $O(m)$. Consequently, the expansion takes a total of $O(m)$ time.

The correctness of the algorithm can be verified in a straightforward manner. The crucial point is that the number-based suffix tree has the essential structure of the final word suffix tree, and that the expansion stage does not change this.

Theorem 1. *A word suffix tree for an input string of size n containing m words can be built in $O(n)$ expected time and $O(m)$ deterministic space.*

5. Extensions and Variations

Although the use of non-lexicographic suffix trees and randomization (hash coding) during construction is sufficient for a majority of practical applications, we describe extensions to Algorithm B in order to meet stronger requirements.

5.1. **Building a Lexicographic Trie.** Although applications commonly have no use for maintaining a lexicographic trie, there are cases where this is necessary. (A specialized example is the number-based suffix tree created in Stage 4 of Algorithm B).

If the alphabet size k is small enough to be regarded as a constant, it is trivial to modify Algorithm B to create a lexicographic tree in linear time: instead of hash tables, use any ordered data structure (most naturally an array) of size $O(k)$ to store references to the children at each node.

If hashing is used during construction as described in the previous section, Algorithm B can be modified to construct a lexicographic trie simply by requiring the number assignments in Stage 2 to be lexicographic instead of semi-lexicographic. Thereby, the number assignment reflects the lexicographic order of the words exactly, and this order propagates to the final word suffix tree. A lexicographic number assignment can be achieved by ensuring that the word trie constructed in Stage 1 is lexicographic. Observation 1 states that the trie can be made lexicographic at an extra cost which is asymptotically the same as for sorting m' characters, which yields the following:

Theorem 2. *A lexicographic word suffix tree for an input string of size n containing m words of which m' are distinct can be built in $O(n + s(m'))$ expected time and $O(m)$ deterministic space, where $s(m')$ is the time required to sort m' characters.*

For the general problem, with no restrictions on alphabet size, this implies an upper bound of $O(n \log\log n)$ by applying the currently best known upper bound for integer sorting [2].

5.2. **A Deterministic Algorithm.** A deterministic version of Algorithm B can be obtained by representing the tree with only deterministic data structures, such as binary search trees. Also, when these data structures maintain lexicographical ordering of elements (which is common, even for the data structures with the best known time bounds) the resulting tree becomes lexicographic as a side effect. We obtain a better worst case time, at the price of an asymptotically inferior expected performance.

We define $i(m, m')$ to denote the time to insert m characters into ordered dictionaries each bounded in size by m', where $m' \leq m$ is the number of distinct words. In a straightforward manner, we can replace the hash tables of Stages 1 and 4 with deterministic data structures. Since no node may have higher out-degree than m', the resulting time complexity is $O(n + i(m, m'))$.

Theorem 3. *A lexicographic word suffix tree for an input string of size n containing m words of which m' are distinct can be built deterministically in $O(n + i(m, m'))$ time and $O(m)$ space, where $i(m, m')$ is the time required to insert m characters into ordered dictionaries each bounded in size by m'.*

Using binary search trees, $i(m, m') = O(m \log m')$. There are other possibilities, for example we could implement each node as a fusion tree [10], which implies $i(m, m') = O(m \log m' / \log\log m')$, or as an exponential search tree [1], in which case $i(m, m')$ is $O(m\sqrt{\log m'})$ or $O(m \log\log m' \log\log k)$, the latter bound being the more advantageous when the alphabet size is reasonably small.

6. Sublinear Construction: Algorithm C

In some cases, particularly when the alphabet is small, we may assume that the n characters in the input string occupy $o(n)$ machine words. Then it may be possible to avoid the apparently inescapable $\Omega(n)$ cost due to reading the input.

This theme can be altered in many ways, the details depend on the application. The purpose of this (somewhat technical) section is to show that a cost of $\Omega(n)$ is not a theoretical necessity.

We start by studying the case when the positions of the delimiters are known in advance. Then we describe an application where the input string can be scanned and delimiters located in $o(n)$ time.

If the alphabet size is k, then each character occupies $\log k$ bits and the total length of the input is $N = n \log k$ bits stored in N/w machine words, where w is the number of bits in a machine word. (In this section, it is important to distinguish between "words" in the input string and hardware-dependent "machine words".)

We make the following observation:

Lemma 1. *A lexicographic trie containing strings of a-bit characters can be transformed into the corresponding lexicographic trie in a b-bit alphabet in linear time, where a and b are not larger than the size of a machine word.*

Proof (sketch). The transformation is made in two steps. First, we transform the trie of a-bit characters into a binary trie. The binary trie is then transformed into the final b-bit trie. For the first part, the essential observation is that the lowest common ancestor of two neighboring strings can be computed by finding the position of their first differing bit. This position can be found in constant time [10]. Using this information, we can construct a binary path compressed trie in the same manner as the node expansion stage of Algorithm B. The binary trie, in turn, can be trivially transformed into a trie of the desired degree in linear time. (For a more detailed discussion, we refer to [2]). □

The following algorithm, which we refer to as *Algorithm C*, builds a word suffix tree, while temporarily viewing the string as consisting of n' b-bit pseudo-characters, where $n' = o(n)$. It is necessary that this transformation does not cause the words to consist of fractions of pseudo-characters. Therefore, in the case where a word ends at the ith bit of a pseudo-character, we pad this word implicitly with $b - i$ bits at the end, so that the beginning of the next word may start with an unbroken pseudo-character. This does not influence the structure of the input string, since each distinct word can only be replaced by another distinct word. Padding may add at most $m\,(b-1)$ bits to the input. Consequently,

$$n' = O\left(\frac{N + m\,(b-1)}{b}\right) = O\left(\frac{N}{b} + m\right).$$

We are now ready to present Algorithm C:

1. Construct a non-lexicographic word trie in the b-bit alphabet in time $O(n')$, as in Stage 1 of Algorithm B. The padding of words does not change the important property of direct correspondence between the words and the leaves of the word trie.
2. Sort the edges of this trie, yielding a lexicographic trie in the b-bit alphabet in $O(m' + s_b(m'))$ time, by Observation 1, where $s_b(m')$ is the time to sort m' b-bit integers.
3. Assign in-order numbers to the leaves, and generate the number string in time $O(n')$, in the same manner as Stage 3 of Algorithm B.
4. Convert this word trie into a word trie in the original k-size alphabet, utilizing Lemma 1. (This does not affect the in-order numbers of the leaves).
5. Proceed from Stage 4 of Algorithm B.

The first four steps take time $O(n' + s_b(m'))$, and the time for the completion of the construction from Stage 4 is $O(m)$. Thus the complexity of Algorithm C is $O(n' + m + s_b(m'))$. Thereby we obtain the following theorem:

Theorem 4. *When the positions of all delimiters are known, a lexicographic word suffix tree on a string comprising m words of which m' are distinct, can be constructed in time*

$$O\left(\frac{N}{b} + m + s_b(m')\right)$$

for some integer parameter $b \leq w$, where N is the number of bits in the input, w is the machine word length, and $s_b(m')$ is the time to sort m' b-bit integers.

Note that Theorem 4 does not give a complete solution to the problem of creating a word suffix tree. We still have to find the delimiters in the input string, which may take linear time. Below, we illustrate a possible way around this for one application.

Example: Huffman Coded Text. Suppose we are presented with a Huffman coded text and asked to generate an index on every suffix starting with a word. Furthermore, suppose that word boundaries are defined to be present at every position where a non-alphabetic character (a space, comma, punctuation etc.) is followed by an alphabetic character (a letter), i.e. we have implicit ♭ characters in these positions. The resulting word suffix tree may be a binary trie based on the Huffman codewords, or a trie based on the original alphabet. Here we assume the former.

We view the input as consisting of b-bit pseudo-characters, where $b = \frac{\log n}{2}$. The algorithm is divided in two main parts.

1. Create a code table. We start by creating a table containing 2^b entries, each entry corresponding to one possible pseudo-character. For each entry, we scan the corresponding pseudo-character and examine its contents by decoding the Huffman codewords contained in it. If there is an incomplete Huffman codeword at the end, we make a note to the length of this codeword. We denote the decodable part of the pseudo-character as a *chunk*. While decoding the contents of a table entry, we check if any word boundaries are contained in the decoded chunk. If so, this is noted in the table entry. Furthermore, we check if the last character in the chunk is non-alphabetic, in which case we note that this character, together with the first character in the next chunk, may define a word boundary.

The time to create and scan the table is at most proportional to the total number of bits it contains, which is $2^b \cdot b$.

2. Scan the input and locate delimiters. We use p as a pointer into the input string.

1. Set p to 1.
2. Read a pseudo-character (b bits), starting at position p.
3. Use the pseudo-character as an address in the code table. Examine if any word boundaries are contained in the corresponding decoded chunk. Let i be the length of the chunk. We have two cases:
   (a) $i \geq b/2$. Update p to point at the first bit after the chunk and go to 2.
   (b) $i < b/2$. Continue reading bits in the input string one at a time while traversing the Huffman tree until the end of a character is found. Update p to point at the first bit after this character and go to 2.

Assuming that b consecutive bits can be read in $O(1)$ time, the time consumption for Step 2 is constant. This step is performed a total number of $O(\lceil N/b \rceil)$ times.

Case 3a takes constant time plus the number of found word boundaries. Hence the total cost of this case is $O(N/b + m)$.

Case 3b occurs when more than the last $b/2$ bits are occupied by a single character. It consumes time proportional to the number of bits in the character's codeword each time it occurs. Hence, the total cost of Case 3b equals the total number of bits occupied by codewords of length more than $b/2$.

The length of a Huffman coded text asymptotically approaches the entropy of the text. Therefore, we may assume that the length of the codeword for a character with frequency f approaches $-\log f$. This yields the following:

Observation 3. *Given a Huffman coded input string of n characters, a character whose Huffman codeword occupies i bits occupies a total of $O(ni/2^i)$ bits in the coded string.*

Since the number of characters occupying i bits can not exceed the alphabet size, k, the total number of bits taken by codewords of length i or longer is $O(kNi/2^i)$. Hence, the total number of bits taken by codewords of length $b/2$ or longer is $O(kNb/2^b)$, which gives us a bound on the cost of Case 3b above.

The total cost for finding delimiters becomes

$$O\left(2^b \cdot b + \frac{N}{b} + m + \frac{kNb}{2^b}\right) = O\left(\sqrt{n}\log n + \frac{N}{\log n} + m + \frac{kN\log n}{\sqrt{n}}\right)$$

The first term can be canceled since $N \geq n$ and the last term can be canceled if $k = O(\sqrt{n}/\log^2 n)$. We then get a cost of

$$O\left(\frac{N}{\log n} + m\right).$$

Next, applying Theorem 4 with the same choice of b, we find that $s_b(m') = O(m' + 2^b) = O(m' + \sqrt{n})$ by using bucket sorting; this cost is negligible. The space used by this algorithm is $O(m + \sqrt{n})$, the last term being due to the table. This yields:

Observation 4. *Given a Huffman coded input string of n characters coded in N bits where the alphabet size k satisfies $k = O(\sqrt{n}/\log^2 n)$, a word suffix tree on m natural words can be constructed in time*

$$O\left(\frac{N}{\log n} + m\right),$$

with construction space $O(m + \sqrt{n})$.

It should be noted that even if the alphabet is very large, the complexity of our algorithm would be favorable as long as characters with long Huffman codewords are rare, i.e. when the entropy of the input string is not too high.

7. A Comment on the Time–Space Tradeoff in Practice

Asymptotically, our space requirement is better than that of alternative data structures, like the traditional suffix tree and the suffix array [14] or PAT array [11]. In practice, however, an asymptotic advantage may sometimes be neutralized by high constant factors.

There are two aspects of space efficiency: the space required to construct the data structure and the space required by the data structure after construction.

First, we study the construction space. Recall that we have n characters, m words, and m' distinct words. The space taken by our construction algorithm equals the space required to construct a traditional suffix tree of m characters, plus the space required to store m' words in the word trie, (including lowest common ancestor links). In many practical cases, see Figure 3 for example, m' is considerably smaller than m and we can neglect the space required by the word trie.

Next, we discuss the space required by the final data structure. As a very natural and space-efficient alternative to a tree structure, one may store the suffix references in a plain sorted array, denoted suffix array [14] or PAT array [11]. Such an array may be searched through binary search, but in order to speed up the search, Manber and Myers [14] suggest using an additional bucket array. The idea is to distribute the suffixes among b buckets according to their first $\log b$ bits, as in traditional bucket sorting. Manber and Myers suggested $b = n/4$ as a suitable value, but other values are possible. This method is very helpful is cases when the suffix array is stored in secondary memory; by storing a small bucket array in internal memory, we may decrease the number of disk accesses significantly. One disadvantage with the bucket array, however, is that the number of suffixes may vary a lot between the buckets, many buckets may even be empty. Therefore, Andersson and Nilsson suggested an improvement [5]: instead of the bucket array, a small, efficiently implemented suffix tree may be used to index the array of pointers. It is experimentally demonstrated that this data structure uses less space than the bucket array while the number of disk accesses is smaller. The reason for this advantage is that a trie in general, and a level compressed trie in particular [3, 4], adapts more nicely to the input distribution than a bucket array. For the same reasons, we conjecture that an efficiently implemented word suffix tree will offer a better time–space tradeoff than a bucket array.

Of course, an obvious advantage compared with the suffix array is the construction cost; we need only $\Theta(n)$ time (in some cases we also need to sort m' characters) while Manber and Myers' suffix array algorithm requires $\Theta(n \log n)$ time.

8. Final Comments

The word suffix tree is indeed a natural data structure, and it is surprising that no efficient construction algorithm has previously been presented. We now discuss several practical cases where word suffix trees would be desirable.

With natural languages, a reasonable word partitioning would consist of standard text delimiters: space, comma, carriage return, etc. We could also use implicit delimiters, as in the example in the preceding section. Using word suffix trees, large texts can be manipulated with a greatly reduced space requirement, as well as increased processing speed [7]. Figure 3 indicates that the number of words, m, in common novels, is much less than the length of

	n	m	m'
Mark Twain's *Tom Sawyer*	387922	71457	7389
August Strindberg's *Rda rummet*	539473	91771	13425

FIGURE 3. Examples of natural language text

the work in bytes, n. This difference is even greater when one considers the number of distinct words, m'.

In the study of DNA sequences, we may represent a large variety of genetic substructures as words, from representations of single amino acids, up to entire gene sequences. In many such cases, the size of the overlying DNA string is substantially greater than the number of substructures it contains. As an example, there are merely tens of thousands of human genes, whilst the entire length of human DNA contains approximately three billion nucleotides.

The word suffix tree is of particular importance when the indexed string is not held in primary storage while the tree is utilized. Using the representation of case 4 in Section 2 our data structure allows search operations with a single access to secondary storage, using only $O(m)$ cells of primary storage, regardless of the length of the search string.

The related problem of constructing *evenly spaced suffix trees* has been treated by Krkkinen and Ukkonen [13]. Such trees store all suffixes for which the start position in the original text are multiples of some constant. We note that our algorithm can produce this in the same complexity bounds by assuming implicit word boundaries at each of these positions.

It should be noted that one open problem remains, namely that of removing the use of delimiters—finding an algorithm that constructs a trie of arbitrarily selected suffixes using only $O(m)$ construction space.

REFERENCES

1. Arne Andersson, *Faster deterministic sorting and searching in linear space*, Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science, October 1996, pp. 135–141.
2. Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman, *Sorting in linear time?*, Proceedings of the 27th Annual ACM Symposium on the Theory of Computing, ACM Press, 1995, pp. 427–436.
3. Arne Andersson and Stefan Nilsson, *Improved behaviour of tries by adaptive branching*, Information Processing Letters **46** (1993), 295–300.
4. ———, *Faster searching in tries and quadtrees—an analysis of level compression*, Proceedings of the 2nd Annual European Symposium on Algorithms, Lecture Notes in Computer Science, vol. 855, Springer-Verlag, September 1994, pp. 82–93.
5. ———, *Efficient implementation of suffix trees*, Software—Practice and Experience **25** (1995), no. 2, 129–141.
6. Alberto Apostolico, *The myriad virtues of subword trees*, Combinatorial Algorithms on Words, NATO ISI Series (Alberto Apostolico and Zvi Galil, eds.), Springer-Verlag, 1985, pp. 85–96.
7. Ricardo Baeza-Yates and Gaston H. Gonnet, *Efficient text searching of regular expressions*, Proceedings of the 16th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 372, Springer-Verlag, 1989, pp. 46–62.

8. Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, SIAM Journal on Computing **23** (1994), 738–761.
9. Martin Farach, *Optimal suffix tree construction with large alphabets*, Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, October 1997, pp. 137–143.
10. Michael L. Fredman and Dan E. Willard, *Surpassing the information theoretic bound with fusion trees*, Journal of Computer and System Sciences **47** (1993), 424–436.
11. Gaston H. Gonnet and Ricardo A. Baeza-Yates, *Handbook of algorithms and data structures*, Addison-Wesley, 1991, ISBN 0-201-41607-7.
12. Dov Harel and Robert E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM Journal on Computing **13** (1984), 338–355.
13. Juha Krkkinen and Esko Ukkonen, *Sparse suffix trees*, Proceedings of the 2ndAnnual International Conference on Computing and Combinatorics, Lecture Notes in Computer Science, vol. 1090, Springer-Verlag, June 1996, pp. 219–230.
14. Udi Manber and Gene Myers, *Suffix arrays: A new method for on-line string searches*, SIAM Journal on Computing **22** (1993), no. 5, 935–948.
15. Edward M. McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM **23** (1976), 262–272.
16. Wojciech Szpankowski, *A generalized suffix tree and its (un)expected asymptotic behaviors*, SIAM Journal on Computing **22** (1993), no. 6, 1176–1198.
17. Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.
18. Peter Weiner, *Linear pattern matching algorithms*, Proceedings of the 14th Annual IEEE Symposium on Foundations of Computer Science, 1973, pp. 1–11.

THE CONTEXT TREES OF BLOCK SORTING COMPRESSION

N. JESPER LARSSON

ABSTRACT. The *Burrows-Wheeler transform* (BWT) and *block sorting compression* are closely related to the *context trees* of PPM. The usual approach of treating BWT as merely a permutation is not able to fully exploit this relation.

We show that an explicit context tree for BWT can be efficiently generated by taking a subset of the corresponding suffix tree, identify the central problems in exploiting its structure, and trace the influence of the context tree on the common *move-to-front* schemes. We experimentally obtain limits for compression using the constructed trees, and, as an attempt at utilizing the full context tree, present a compression scheme that represents the context tree explicitly as part of the compressed data.

We argue that a conscious treatment of the context tree should be able to achieve the full compression performance of PPM while maintaining the computational efficiency of BWT. Thus, BWT with explicit context trees is a strong candidate for powerful general compression, especially for large data files.

1. INTRODUCTION

Block sorting compression was originally presented by Burrows and Wheeler in 1994 [3]. The method consists of a transform (henceforth referred to as the Burrows-Wheeler transform, BWT) which reorganizes the input string to concentrate repetitions. The transformed string can then be compressed by a simple locally adaptive statistical compression scheme to yield compression ratios close to the best known modelling schemes.

While BWT may at first glance appear to be a magical new algorithm, Cleary, Teahan, and Witten [4] observed that its effect is quite similar to PPM [4, 5, 11]. We take that similarity one step further in giving the *context tree*, which is implicit in BWT, a concrete form. We present a computationally efficient method to construct the tree, explore its power of capturing characteristics of the source, identify the central points in using it for compression, and finally suggest a possible direction towards an efficient complete compression algorithm, presenting a description of an experimental program, with preliminary compression results.

This text is organized as follows: first, we describe the BWT transform, including its time complexity, and discuss previous work. Section 3 identifies the implicit context tree of BWT and gives an efficient algorithm to make it explicit. Section 4 reconsiders the common *move-to-front* encoding from the perspective of the context tree. In Section 5 we discuss explicit use of the context tree in BWT compression and exemplify by presenting an experimental algorithm. We conclude that compression using the context tree is a good candidate for achieving the full benefits of tree model methods such as PPM, while maintaining tight complexity bounds.

2. Background

We first present the basics of BWT as a starting point for the following text, as well as discussions of previous work. Although our formulations are somewhat different, the basis of this section is primarily Burrows and Wheeler [3].

2.1. **Block Sorting Transform.** We assume that the last symbol of the input is a special *end-of-file* symbol \$. With this assumption, sorting *cyclic shifts* of the input, as in the common formulation of BWT, is equivalent to sorting *suffixes* of the same string.

Let the input be a string $t = t_1 \dots t_n$ of symbols. We assume that symbols are represented as integers in the range $[\,1, k\,]$. The output of the transform is the pair $(t', \mathbf{i})$ where t' is a string of length n and $\mathbf{i}$ an integer in the range $[\,1, n\,]$. The transform is performed as follows:

1. Sort all the suffixes of t. Represent the sorted sequence as a vector $S = (S_1, \dots, S_n)$ of numbers in the range $[\,1, n\,]$ such that i precedes j in S iff the suffix that begins in position i of t lexicographically precedes that which begins in position j.
2. Let $\mathbf{i}$ be the number such that $S_{\mathbf{i}} = 1$.
3. For $i \in [\,1, n\,]$, let $t'_i = t_{S_i - 1}$, where we define $t_0 = t_n = \$$.

The effect of the transform is that symbols followed by the same substrings in t are placed in consecutive positions in t'. Referring to the suffix following a position in t as the *context* of that position, we can say that the more similar the contexts of two positions, the closer the symbols in those positions in t. (In PPM, the symbols *preceding* a context define its context. If desired, this can be emulated in BWT, simply by reversing t. However, the difference is normally of no importance.)

Consequently, if t contains repeating patterns, some parts of t'—that originate from similar contexts—comprise only symbols from a small part of the input alphabet. By transferring t' instead of t to the decompressor, we can exploit its regularities efficiently with a simple locally adaptive compression method.

The decompression program needs to reverse the transform to obtain the original string. This remarkably fast and simple procedure can be formulated as follows:

1. For $c \in [\,1, k\,]$, let n_c be the number of occurrences of symbol c in t'.
2. Set $C[\,1\,]$ to 0. For $i = 1, \dots, k-1$, set $C[\,i+1\,]$ to $C[\,i\,] + n_i$.
3. For $i = 1, \dots, n$, set $P\big[C[\,t'_i\,]\big]$ to i and increment $C[\,t'_i\,]$.
4. Set i to $\mathbf{i}$. For $j = 1, \dots, n$, let $t_j = t'_i$ and advance i to $P[\,i\,]$.

2.2. **Sorting Algorithms and Time Complexity.** The key advantage of BWT compression is its moderate requirements on computational resources compared to methods with similar compression performance. Throughout this work, we make an effort to maintain this advantage and avoid processes that notably increase time or space complexity. We now discuss the time complexity of BWT itself.

The computionally critical part of the transform is the suffix sorting. It is important to note that since the elements to be sorted are strings, comparisons potentially take linear time. Therefore, a normally commendable comparison based algorithm may well require $\Omega(n^2 \log n)$ time, so a specialized method is needed.

Existing BWT implementations typically use ad hoc combinations of sorting algorithms, often paired with a run length encoding scheme to handle common degeneration cases [3, 7, 12, 14]. However, as noted by Burrows and Wheeler [3], this can

be improved upon by building a suffix tree (see Section 3.1), which is then traversed in sorted order and the sorted sequence is obtained from the leaves.

The suffix tree implementation yields linear time for the transform. If the alphabet size is too large to be realistically regarded as constant, this time bound can be achieved by representing the edges of the tree with hashing [10]. This requires an additional sorting step, which is done in linear time by bucket sorting the edges and rebuilding the tree in sorted order [1].

Furthermore, even if hashing can not be used, the recently presented algorithm by Farach [6] provides linear time suffix tree construction for all alphabets relevant for BWT. Thus, the time complexity of BWT is deterministically $\Theta(n)$.

2.3. **Move-to-front and Related Coding.** A large majority of the previous work on BWT relies on *move-to-front coding* to exploit the local repetitiveness of the transformed string [3, 7, 12, 14]. The symbols of the input alphabet are placed in a conceptual list, and the position of a symbol in this list, counting from the head, is used to encode the symbol when encountered. Encoded symbols are immediately moved to the head of the list.

This subsidiary transformation of t' produces another string t'' of integers in the range $[\,1, k\,]$, for which the distribution is highly skewed (provided that t is compressible): low numbers are more common than high numbers. Now, t'' can be compressed with simple zero-order statistical compression, such as Huffman or arithmetic coding.

Arnavut and Magliveras [2] devised a slightly different technique named *inversion frequencies*. While move-to-front coding replaces each symbol c with the number of *distinct* symbols encoded since the last occurrence of c, inversion-frequency coding replaces c with the *total* number of symbols *greater than* c encoded since the last occurrence of c. The results were shown to be similar to move-to-front coding.

3. Context Trees

We elaborate the properties of the reorganization performed in BWT by relating to context trees, known from PPM (sometimes referred to as context *tries*). The close relation between PPM and BWT was briefly noted by Cleary, Teahan, and Witten [4].

3.1. **More on Suffix Trees.** A context tree can be viewed as a trie (also known as digital tree) storing substrings of the input string t. Edges of the tree are labelled with symbols, and each node (state) corresponds to the string spelled out by the labels on the path from the root to that node. When the trie contains *all* substrings of the input string (as in PPM* [4]), and, in addition, is path compressed (i.e. all paths of single-child nodes contracted) this results in what is commonly known as a *suffix tree* [6, 9, 10, 13] (see Figure 1).

As noted in Section 2.2, a suffix tree can be used to produce the BWT string t': the tree is traversed left to right, and for each leaf encountered, the symbol preceding the corresponding position of t is emitted as the next symbol of t'. However, the suffix tree is not only a useful tool for the transform, it is also an excellent hierarchical model of similarities between contexts. The leaves of the tree correspond to the contexts. The lowest common ancestor of a pair of nodes, particularly the depth of that ancestor, manifests the similarity between the corresponding pair of contexts, i.e. the length of their common prefix.

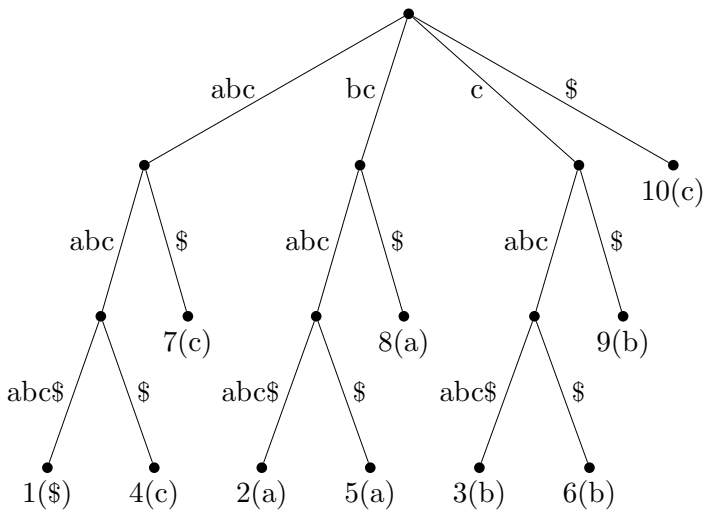

FIGURE 1. The suffix tree of the string 'abcabcabc$'. Below each leaf is shown the number of the suffix it corresponds to and, in parenthesis, the corresponding symbol that would be emitted by the BWT.

For each internal node, we consider the set of frequency counts for the symbols of the input alphabet emitted by the BWT for leaves in its subtree. The root holds the counts for the whole string, which would be used in a simple zero-order encoding, while an internal node corresponding to a string w (where w is the string spelled out by the labels on the path from the root) holds the counts for symbols occurring in the context w. Thus, the suffix tree incorporates exactly the structure of a context tree.[1]

3.2. **Pruning the Tree.** Frequency counts in each internal node as described in the previous section means keeping an absolute maximum of statistics about the context properties of the string. This is generally much more than what is actually needed to fully characterize the source.

As an extreme example, consider a single-state source—there is obviously no gain in using more than one set of counts in modelling this. We should recognize this condition, and remove all internal nodes of that context tree, except the root. Generally, we should remove all internal nodes that do not exhibit any significant change in distribution compared to its parent (see Figure 2). Eventually, for large n, the number of internal nodes should converge towards a number that reflects the number of states in a tree model of the source (provided that there is some tree model that captures the source).

To find an approximation of the optimal context tree, we use a greedy method that recursively prunes the tree bottom-up and left-to-right. This has the advantage of being simple and fast, and consuming little space. At each point in time, we only need to maintain frequency counts for nodes on the path from the root to the node currently being processed. This limits space requirements to the height of the tree times the size of the alphabet. We can limit space requirements even further by simply removing all nodes below a certain, constant, depth. This does not notably

[1]Note however, that since our contexts are the strings *after* each position, the tree representation is "backwards" compared to most PPM descriptions.

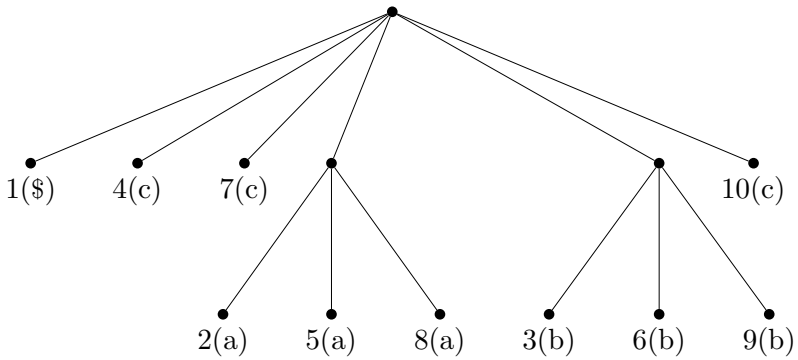

FIGURE 2. A pruned context tree corresponding to the suffix tree in Figure 1.

infer on the final product (it is extremely rare that nodes below a depth of about seven are maintained), but yields an important improvement in worst case space complexity.

In principal, the pruning algorithm works as follows: At each node, we calculate the optimal code length for encoding symbols both including and excluding that node. If keeping the node does not yield a smaller total code length, we remove it.

In addition to maintaining counts over the input alphabet, we also need to take into account the discrepancy in which symbols are used in different subtrees. Again in terms borrowed from PPM, we employ an *escape* mechanism to account for the cost of introducing new events in a state. The first time a symbol occurs, it charges an increase of the escape count instead of the count of the symbol itself.

More specifically, the greedy pruning algorithm prunes the subtree rooted at an internal node u as follows:

1. For each leaf child of u, check which symbol the transform should produce corresponding to that leaf. Then for each symbol c, set $n_{c,u}$ to the number of times c was encountered in this process.
2. Repeat steps 3 to 8 for each internal-node child v of u.
3. Recursively prune the subtree rooted at v.
4. For each symbol c, let $e_c = 1$ if c occurs in the subtree rooted at v, and $e_c = 0$ otherwise. This is to account for escape events in the subtree.
5. Calculate the optimal code length $\mathbf{h}_u$ for encoding, as an independent sequence, the symbols corresponding to leaf children of u, using $n_{c,u} + e_c$ as frequency counts.
6. Analogously calculate the optimal code length $\mathbf{h}_\Sigma$ for encoding symbols in the combination of u and v, using $n_{c,u} + n_{c,v}$ as frequency counts.
7. If $\mathbf{h}_\Sigma < \mathbf{h}_u + \mathbf{h}_v$, then delete v and let all children of v become children of u. Update the $n_{c,u}$ by adding to them their corresponding $n_{c,v}$.
8. Otherwise, update the $n_{c,u}$ by adding to them their corresponding e_c.

Calculation of code lengths is expressed as follows: Let $U = \{\, c \mid n_{c,u} + e_c > 0 \,\}$, and $n_U = \sum_{c \in U} n_{c,u} + e_c$. Summing code length for escapes and symbols in u, we have

$$\begin{aligned}\mathbf{h}_u &= |U| \log \frac{n_U}{|U|} + \sum_{c \in U} (n_{c,u} + e_c - 1) \log \frac{n_U}{n_{c,u} + e_c - 1} \\ &= l(n_U) - l(|U|) - \sum_{c \in U} l(n_{c,u} + e_c - 1),\end{aligned}$$

	size	nodes	bits
bib	111261	10248	1.80
book1	768770	59701	2.19
book2	610856	50286	1.87
geo	102400	5442	4.37
news	377109	40568	2.28
obj1	21504	1972	3.45
obj2	246814	27037	2.27
paper1	53161	6265	2.26
paper2	82199	8338	2.22
pic	513216	7499	0.78
progc	39611	4939	2.29
progl	71646	7958	1.59
progp	49379	5922	1.62
trans	93695	10328	1.42

TABLE 1. Results of the pruning algorithm for the Calgary corpus. *Size* is the original file size in bytes, *nodes* the number of internal nodes maintained by the pruning, and *bits* is the calculated code length in bits per symbol.

where $l(n) \equiv n \log n$. The calculation of $\mathbf{h}_\Sigma$ is analogously reduced to a sum of $l(n)$ terms. The function $l(n)$ can be efficiently implemented through a simple halving procedure, which can be speeded up further by a lookup table. We may therefore realistically assume that these calculations are dominated by set operations, which yields a worst case complexity of $O(n \log k)$ for the greedy pruning algorithm (where k is the alphabet size), with a straightforward implementation using (possibly implicit) binary trees.

3.3. **Code Length Measurements.** To illustrate how the context tree captures the statistics of a file, Table 1 shows experimental results of code length, using the files of the Calgary corpus as input. Note that these are not compression results, since information about the tree structure is not included (see Section 5), but rather lower bounds for what can be achieved by the greedy-pruned context tree.

The measurements show how much redundancy the context tree is able to capture for different kinds of data. Perhaps the most interesting point is that since the pruned context tree approximates the optimal context tree, this can be regarded as an estimate of the actual entropy for the corresponding source, assuming that the tree model assumption holds. Thus, the values are an approximation of the lower bound for *any* tree model based compression method, including PPM, when compressing data with these characteristics.

4. The Relationship between Move-to-front Encoding and Context Trees

We now review the move-to-front encoding described in Section 2.3 from a context tree perspective, in order to shed light on some important points regarding its performance.

The transform of t' into t'' serves to replace the local repetitions of t' by a globally skewed distribution that would ultimately submit to compression using static frequency counts. However, static encoding is a poor choice. While lower numbers are indeed generally more common than high numbers in t'', their probabilities vary due to the following facts:

- The move-to-front process has no notion of depth changes in the context tree. While BWT places similar contexts close to each other, many not so similar contexts still end up in consecutive positions. The extreme case occurs when all the contexts beginning with a particular symbol are exhausted—the next position corresponds to a completely different context, e.g., a character followed by 'baaa' may be placed directly after a character followed by 'azzz'.
- The degree of regularity varies between contexts. As an example, in English text the characters followed by the string 'the ' are extremely regular (almost all spaces), while the characters followed by ' the' are much less predictable. In information theoretic terms: different states of the source have different entropy. Again, a simple left-to-right view is unable to take context changes into account.

Existing implementations essentially all deal with these inherent disadvantages in the same way: they employ highly adaptive statistics. The simplest method is the common approach of periodically scaling down frequency counts, typically halving them. This gives local probability distributions an advantage over old statistics.

Despite the apparent crudeness of this approach—throwing away large amounts of the collected statistics—it can give quite astonishing results. Fenwick [7] reports the same average as the PPM* algorithm [4] for the Calgary corpus. The key to this performance lies in the extreme degree of repetition in t' for some files, which produces long runs of zeroes in the move-to-front transform. This is a global property of those files, which remains in spite of the loss of detail in the estimates.

5. Context Tree BWT Compression Schemes

The ultimate goal of our exploration of the BWT context tree is of course to find a competitive compression scheme. However, while the possibilities appear to be immense, it is far from clear what is the best way of exploiting the context tree.

An interesting option, that has a clear potential of competing with move-to-front encoding in computational requirements, is to include a representation of the structural properties of the tree as part of the compressed data, and then encoding the BWT transformed string left-to-right, dynamically updating frequency counts as in PPM. Below we discuss a simple implementation using this strategy. It works well for large files (where the tree representation comprises a small part of the data), but it appears that a more sophisticated tree encoding is required for this method to be a general improvement over move-to-front encoding.

5.1. **Further Pruning.**

When the tree is to be explicitly represented we need to reconsider the pruning strategy. Now, the tree that models the data optimally is not necessarily the best choice, since the size of the tree is a factor. We need to weigh the cost of representing each node against the gain of utilizing that node.

Consequently, the pruning algorithm should be modified so that it maintains a node only if the gain in code length is larger than the cost of representing that node. However, the cost of representing a node is not easily predicted. It depends, naturally, on our choice of representation of the tree, but also on the structure of the

whole tree. Our experimental algorithm employs the simplest possible strategy: the cost of representing each node is estimated as a constant, whose value is empirically determined. Furthermore, we impose a lower limit on the number of leaves in a subtree; all nodes with less than some constant number of leaves below are removed.

5.2. **Encoding the Tree.** A pruned context tree has properties which makes it highly compressible. One quickly noted attribute that is easy to take advantage of, is that a large majority of the nodes are leaves. Less obviously exploitable are the structural repetitions in the tree: small subtrees are essentially copies of larger subtrees with some nodes removed.

In the current implementation we use the following simplistic encoding method: we traverse the tree in order, obtaining the number of children of each node. These numbers are encoded as exponent-mantissa pairs, where the exponents are compressed with a first-order arithmetic encoder whose state is based on the size of the parent.

5.3. **Encoding the Symbols.** In encoding the symbols corresponding to the leaves of the tree, we have to choose a strategy for transferring the frequency counts to the decoder. One possibility is to encode them explicitly, as we do the structure of the tree. Another, which we have chosen in the current implementation, is to use the tree only for state selection and encode new symbols by escaping to shorter contexts, as in PPM compression.

The crucial difference compared to PPM is that of computational efficiency and simplicity: Since we encode left-to-right in the tree, we only need to maintain frequency count for one branch of the tree at a time. Furthermore, escaping to a shorter context is simple, since the shorter context is the *parent* of each node—we do not need the suffix links, or escape lists, of PPM implementations.

Now, we have the same choices as in PPM regarding strategies of escape probability estimation, inheritance, exclusion etc. Again because the tree is traversed in order, most conceivable choices are easily and efficiently implemented, which opens extensive possibilities for refinement. Our current implementation uses no inheritance, an escape estimate similar to PPMD [8], full exclusion, and update exclusion.

5.4. **Preliminary Results.** Table 2 shows the current results of our experimental compression program. The limits chosen for the pruning algorithm were five bits as the minimum gain to retain a node, and a minimum of eight leaves for subtrees rooted at internal nodes.

It is clear from the table that for these files our current experimental program is no general improvement over the best known BWT implementations—only the largest file, book1, yields a total improvement over the move-to-front results achieved by Fenwick [7]. In particular, the the tree encoding scheme must be improved in order to achieve favourable compression ratios for files as small as these (although for a few files, Fenwick's implementation performs better even disregarding the *tree* part). The small number of internal nodes retained by the pruning indicates that this improvement should certainly be possible through a more sophisticated tree encoding.

For very large files, the representation of the tree should eventually be negligible, provided that the source can be represented as a tree model, and that the number of internal nodes of the context tree converges towards a constant that reflects the number of states in this model (see Section 3.2).

	size	nodes	bits (tree + sym)
bib	111261	2308	2.26 (0.28 + 1.98)
book1	768770	7777	2.37 (0.15 + 2.22)
book2	610856	8793	2.17 (0.21 + 1.96)
geo	102400	899	4.69 (0.13 + 4.56)
news	377109	7350	2.76 (0.26 + 2.50)
obj1	21504	516	4.27 (0.32 + 3.95)
obj2	246814	6303	2.94 (0.33 + 2.61)
paper1	53161	1384	2.84 (0.35 + 2.49)
paper2	82199	1634	2.65 (0.28 + 2.37)
pic	513216	652	0.80 (0.02 + 0.78)
progc	39611	1071	2.92 (0.36 + 2.56)
progl	71646	1778	2.13 (0.33 + 1.80)
progp	49379	1256	2.22 (0.34 + 1.88)
trans	93695	2775	2.06 (0.37 + 1.69)

TABLE 2. Results of the experimental compression program. *Size* is the original file size in bytes, and *nodes* the number of internal nodes maintained. *Bits* is the average number of bits per compressed symbol, divided into *tree* and *sym* (symbol) to show the relative code space for tree encoding used by the program for these files.

6. Final Comments

Data compression using BWT has an advantage over other tree model based methods in its moderate requirements on computational resources. We assert that this advantage can be maintained with a much more sophisticated modelling method than the move-to-front transform. Our context tree approach reveals the possibility of using BWT to obtain a tight time complexity while taking advantage of sophisticated techniques developed for PPM.

However, finding the optimal combination of these two approaches remains as an open problem. Particularly, if the approach of representing the context tree explicitly is used, the structure of the tree must be further analyzed, and a sophisticated encoding scheme designed, if the method is to be competitive for small files.

It should be noted that while we have approached the context trees of BWT with suffix trees as a starting point, the process of pruning the suffix tree to obtain a useful context tree is by no means the only possibility. On the contrary, the small number of internal nodes maintained by the extended pruning indicates that a top-down method of constructing the tree (which could be made to consume less memory) should certainly be considered. This is particularly the case when large blocks of data are treated, since the suffix tree may then require considerable (although linear) storage space.

In conclusion, we conjecture that BWT compression with an explicit treatment of context trees is a strong candidate for achieving the full compression performance of PPM and similar methods, with a program that requires only moderate computational resources.

References

1. Arne Andersson, N. Jesper Larsson, and Kurt Swanson, *Suffix trees on words*, Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 1075, Springer-Verlag, June 1996, pp. 102–115.
2. Ziya Arnavut and Spyros S. Magliveras, *Block sorting and compression*, Proceedings of the IEEE Data Compression Conference, March 1997, pp. 181–190.
3. Michael Burrows and David J. Wheeler, *A block-sorting lossless data compression algorithm*, Research Report. 124, Digital Systems Research Center, Palo Alto, California, May 1994.
4. John G. Cleary, W. J. Teahan, and Ian H. Witten, *Unbounded length contexts for PPM*, Proceedings of the IEEE Data Compression Conference, March 1995, pp. 52–61.
5. John G. Cleary and Ian H. Witten, *Data compression using adaptive coding and partial string matching*, IEEE Transactions on Communications **COM-32** (1984), 396–402.
6. Martin Farach, *Optimal suffix tree construction with large alphabets*, Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, October 1997, pp. 137–143.
7. Peter Fenwick, *Block sorting text compression*, Proceedings of the 19th Australasian Computer Science Conference (Melbourne, Australia), January–February 1996.
8. Paul Glor Howard, *The design and analysis of efficient lossless data compression systems*, Ph.D. thesis, Deptartment of Computer Science, Brown University, Providence, Rhode Island, June 1993, CS-93-28.
9. N. Jesper Larsson, *Extended application of suffix trees to data compression*, Proceedings of the IEEE Data Compression Conference, March–April 1996, pp. 190–199.
10. Edward M. McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM **23** (1976), no. 2, 262–272.
11. Alistair Moffat, *Implementing the PPM data compression scheme*, IEEE Transactions on Communications **COM-38** (1990), no. 11, 1917–1921.
12. Julian Seward, *`bzip2` program*, `http://www.muraroa.demon.co.uk/`, 1997.
13. Esko Ukkonen, *On-line construction of suffix trees*, Algorithmica **14** (1995), no. 3, 249–260.
14. David Wheeler, *An implementation of block coding*, Tech. report, Cambridge University Computer Laboratory, October 1995.