1985

The Myriad Virtues of Subword Trees

Alberto Apostolico

Report Number:

85-540

THE MYRIAD VIRTUES OF SUBWORD TREES

Alberto Apostolico

CSD-TR-540
October 1985

THE MYRIAD VIRTUES OF SUBWORD TREES

ALBERTO APOSTOLICO
Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

Several nontrivial applications of subword trees have been developed since their first appearance. Some such applications depart considerably from the original motivations. A brief account of them is attempted here.

INTRODUCTION

Subword trees fit in the general subject of digital search indexes [KN]. In fact their earliest conception is somewhat implicit in Morrison's 'PATRICIA' tries [MO]. Several linear time and space subword tree constructions are available today [MC, PR, SL] (see also [AH]), following the pioneering work by Weiner [WE]. More compact alternate versions have been introduced recently in [BL, BE, CS2]. The data structures developed in this endeavor are variously referred to as B-trees, position trees, suffix (or prefix) trees, subword trees, repetition finders, directed acyclic word graphs, etc. A concise account of the similarities and discrepancies among the various approaches is presented in [SE1, CS1]. On line (though not linear time) constructions are discussed in [MR]. In this paper, we choose to refer mostly to the version in [MC], to which we also conform as much as possible as for basic definitions and notations. However, the properties presented here are to a large extent independent of the particular incarnation of a subword tree, and, from the conceptual standpoint, so are indeed the associated criteria and constructions. This paper addresses itself to a reader with scarce previous exposure to the subject, but it does assume some familiarity with elementary facts and concepts in combinatorics on words. The paper is also self-contained in the description of the various applications presented. However, some proofs are only sketched; the reader is also pointed to the referenced literature when it comes to constructions too elaborate to be given here in full details. Finally, the list given here is not meant to be exhaustive. In particular, it reflects some recent involvements of this author, and his personal perspective.

The paper is organized as follows. Basic properties and applications of subword trees are outlined in the next section. In Section 2, such trees are treated as a unifying framework for the description of a class of linear time sequential data compression techniques that is becoming increasingly popular. In Section 3, we take steps from one such data compression paradigm and use subword trees to decide whether a word contains a square subword, in linear time. We show next how subword trees can be used also to spot all such squares, as well as to establish bounds on the number of cube subwords in a string. Augmented subword trees are suited to allocate the statistics without overlap of all subwords of a textstring, as highlighted in Section 4. In Section 5, we mention two applications in which subword trees are outperformed by other approaches.

1. PRELIMINARIES

We shall deal with *strings (words)* of *symbols* from a finite alphabet I. If x is a word, $|x|$ will denote the *length* (i.e., the number of symbols) of x. Sometimes we will implicitly assume $|x|=n$. The set of all distinct nonempty substrings of x (*subwords*) is called the *vocabulary* of x, denoted V_x. We say that $x_i x_{i+1} \cdots x_{i+|w|-1}$ is an *occurrence* of $w \in V_x$ in x if $x_{i+k}=w_k$ ($k=0,1,\ldots,|w|-1$). Let $\$ \notin$ I be a special end-marker. For each i in the set $P=\{1,2,\ldots,n+1\}$ of *positions* of $x\$$, suf_i denotes the ith *suffix* of $x\$$. Since $\$ \notin$ I, it is always possible to write $\text{suf}_i=\text{head}_i \cdot \text{tail}_i$, with tail_i nonempty and head_i the longest prefix of suf_i which is also a prefix of suf_j for some $j<i$. The *subword tree* T_x associated with x is defined here as the digital search tree with $n+1$ leaves and at most n interior vertices such that: each edge is labeled with an occurrence of a subword of x via a pair of pointers to a common, randomly accessible, copy of x; each leaf is labeled with a position in P; the labels on the path from the root to leaf labeled i describe suf_i. This labeling policy enables to maintain an $O(n)$ space allocation for any subword tree. Figure 1 displays a portion (i.e., all suffixes starting with a) of T_x for $x=abaababaabaababaababa$.

Any vertex α of T_x distinct from the root describes a subword $w(\alpha)$ of x in a natural way: vertex α is called the *proper locus* of $w(\alpha)$. In general, the *locus* of $w \in V_x$ in T_x is the unique vertex of T_x such that w is a prefix of $w(\alpha)$ and $w(\text{FATHER}(\alpha))$ is a proper prefix of w.

The obvious approach to the construction of T_x is to start with the empty tree T_0 and inserts suffixes in succession into an increasingly updated version of the tree, as follows.

$$\textbf{for } i:=1 \textbf{ to } n+1 \textbf{ do } T_i \leftarrow \text{ insert } (T_{i-1}, \text{suf}_i)$$

A brute force implementation of *insert* would lead to an algorithm taking $O(n^2)$ time in the worst case. The time consuming subtask of *insert* is that of finding the locus of head_i ($i=1,2,\ldots,n+1$) in T_{i-1} (head_i might not have a proper locus in T_{i-1}, but it certainly will in T_i). McCreight's construction [MC] exploits auxiliary "suffix links" to retrieve the locus of head_i ($i=1,2,\ldots,n+1$) in overall linear time. Basically, this is made possible by the simple fact that if $\text{head}_i=aw$ ($i=1,2,\ldots,n$) with $a \in I$, then w is a prefix of head_{i+1}. All clever variations of subword trees are built in linear time by resorting to similar properties.

The original motivation behind Wiener's construction of the first subword tree [WE] was that of transmitting and/or storing a message with excerpts from a main string in minimum time or space. It became soon apparent that the structure of such indexes is ideally suited to several other, almost straightforward, applications.

- By treating T_x as the state transition diagram of a finite automaton it is possible to decide whether or not $w \in V_x$, for an arbitrary w, in $O(|w|)$ time. This is of use in multiple searches for different patterns in a fixed set. The particular role played by \$ makes it possible to tell also whether w is a suffix of x, for the same cost.
- Assume that each vertex of T_x bears the label of the smallest leaf label in its subtree (this is not difficult to maintain during the construction of T_x or it can be achieved in one appropriate walk of T_x). Then it is possible to find in $O(|w|)$ steps and for arbitrary w what is the first occurrence of w in x (whence also whether w is a prefix of x). Notice that to find the last occurrence of w in $O(|w|)$ time for any w requires a walk through T_x, after its construction: similar

asymmetries are inherent to other variations of the tree as well.

- Let $w \in V_x$ and α the locus of w in T_x. By inspecting the leaves in the subtree of T_x rooted at α we can pinpoint all the occurrences of w in x in $O(|w|+\text{output})$ time.
- Consider the *weighted vocabulary* (V_x,C), where the weighting functions C associates, with each $w \in V_x$, the number of occurrences of w in x. To allocate (V_x,C) it is sufficient to traverse T_x bottom up weighting each vertex with the sum of the weights of its offsprings (leaves have weight 1). Then for each $w \in V_x$, $C(w)$ is retrieved in $O(|w|)$ time by accessing the (not necessarily proper) locus of w in T_x.
- Let head_i^* be the longest prefix of suf_i which has a non-leaf locus in T_x; let $\text{suf}_i = \text{head}_i^* \text{tail}_i^*$ and assume that a is the first symbol of tail_i^*. The string $\text{head}_i^* \cdot a$ is the shortest subword of x that occurs only at position i. This is the *substring identifier* for i [AH]: it tells how much of a pattern is necessary to identify a position in the text x completely, which can spare time during searches.
- The head_i of maximum length is the longest repeated subword of x. The tree associated with the string $x\#y\$$ ($\# \notin$ I) makes it possible to find the longest common substring of x and y in $O(n+m)$ time, where $m = |y|$. It is remarkable that this problem has such a straightforward solution once T_x is given. A previous algorithm [KMR] could solve it only in $O((n+m)\log(n+m))$, and, as is reported in [KMP], Knuth had conjectured in 1970 that linear time performance was impossible to achieve.
- The longest subword common to k out of m strings of total length n can be also found in $O(n)$ time, although by more elaborate constructions [PR]. This is not trivial, since the straightforward extension of the case $m=2$ produces an algorithm taking $O(n \cdot m)$ time.

2. A FRAMEWORK FOR LINEAR TIME SEQUENTIAL DATA COMPRESSION

Subword trees T_Q for the set of suffixes suf_j where $j \in Q = \{i_1,i_2,...i_n\}$ and Q is an ordered subset of P are the natural habitat for a class of sequential data compression techniques based on textual substitution. As pointed out elsewhere in this book [ST], this class embodies the few optimization problems in the realm of textual substitution that can be solved in polynomial (actually linear) time. In fact the techniques in this class also feature asymptotic optimality in the information theoretic sense [ZI, ZL, ZL1, LZ, SZ1, SZ2].

The idea is in general that of interleaving the construction of a (possibly partial) subword tree with a *parse* of the textstring into *phrases*. Compression is achieved whenever phrases are susceptible of a more compact representation.

The set Q is retrieved from P by means of a *generative process*, which is actuated by following a set of rules to identify, for each suffix of x with starting position $i_j \in Q$, the associated jth *reproduction* rep_j of x and its strictly related *production* prod_j. The exact nature of rep_j depends on the particular generative process chosen. In all cases, however, rep_j will coincide with a suitable prefix of a suffix suf_{i_f}, with $i_f \in Q$ and $f < j$; prod_j is always:

$$\text{prod}_j = \text{rep}_j \cdot x[i_j + |\text{rep}_j|]$$

Thus prod_j is fully individuated by setting some suitable pointer(s) to the previous suffix and by providing the (possibly new) terminal symbol. This information is the *identifier* for prod_j, denoted by $id(j)$.

For each type $(A,B,C,\ldots)$ of reproduction defined, the *<type>−parse* of x, denoted $type-P(x)$ is the (unique) decomposition of x in terms of those productions that are pinpointed through the greedy left to right scanning of the symbols of x. The production of x that is selected by actuating the hth step in this process represents the hth *phrase* in the parse. Since each phrase is also a production, we can associate with the parse of x its *translation* $\sigma(x)$, defined as the concatenation of the identifiers for the productions that are also phrases in the parse.

The paradigm of the procedure *parse* below encompasses most instantiations of the generative processes in [ZI, LZ, SZ1, SZ2, AGU]. We assume that the operation of *insert* is accompanied with the identification of the current (re)production via the auxiliary function *Lprefix*, and by the insertion of an auxiliary endmarker node whenever needed for possible later reference.

```
    procedure parse (x,q)
    ## produces σ(x) from inputs x and characteristic function q ##
1.  begin i:=1; j:=1; h:=1; T_1:={suf_1};
              phrase_1 := prod_1 := x[1]; σ := id(1) := <x[1]>;
2.  while i<n do ## produce next phrase ##
3.      begin i:=i+1; j:=j+1; h:=h+1;
4.      T_j := insert(T_{j-1}, suf_i)
5.      phrase_h := prod_j := Lprefix(suf_i);
6.      σ=σ·id(j);
7.      if i+|rep_j|<n then
              ## generate intermediate (re)productions ##
              begin
8.              m:=j
9.              with k∈q(i,|rep_j|) do
10.             begin m:=m+1; T_m:=insert(T_{m-1}, suf_k) end
              end
11.         i:=i+|rep_j|
        end
12.     σ(x):=σ
    end.
```

The loop of lines (9,10) enriches the vocabulary between 'active' parsing steps by inserting extra suffixes according to some given characteristic function q. The two extreme cases are when q exhausts all intermediate positions (i.e., $k=i+1, i+2$, etc.), and when it neglects them all. In this latter case it results in $j=h$ at all times during *parse*. One expects the number of phrases in the parse to decrease as the number of intermediate insertions increases. However, there is a subtle interplay between the number of intermediate insertions and the sizes resulting for identifiers, which might offset this benefit. For example, let:

$$x = 111111111111111110111011101110101010101010001000100$$

The A-parse is characterized as follows:

Lprefix(suf_i) - coincides with the longest prefix of suf_i that matches some past production (=phrase), extended by concatenation of the next symbol of suf_i.

Example: $A-P(x)$ = 1-11-111-1111-11111-11110-1110-11101-110-111010-10-101-0-00-100-01-00$

phrases = 17

$|id(j)|$ = $\lceil \log j \rceil$ = $\lceil \log h \rceil$ bits [SZ 1] (i.e., roughly the bits needed to identify one among $h-1$ previous phrases plus the empty phrase λ.)

The B-parse is as follows:

Lprefix(suf_i) - is given by the longest prefix of suf_i that matches the concatenation of two past phrases followed by the terminal symbol as above, or else it is as per scheme A if no such pair of phrases exists.

Example: $B-P(x)$ = 1-11-1111-1111111-11111-0-1110-11101-110- 111010-10-101-010-00-010-001-00$

phrases = 16

$|id(j)|$ = $\lceil \log(3j) \rceil$ = $\lceil \log(3h) \rceil$ [SZ 2] (roughly, the current phrase is identified by selecting one of the h possible simple phrases, plus the $h-2$ pairs followed by an incoming 1, plus as many pairs followed by a 0).

The C-parse and the D-parse are closely related. For the first one we have:

Lprefix(suf_i) - is chosen as the longest concatenation of past phrases, ending perhaps in a prefix of a past phrase, followed by the new symbol as above.

Example: $C-P(x)$ = 1-11-1111-11111111-1110-11101-110-1110111010- 10-101-0-00-10001-00$

phrases = 14

$|id(j)|$ = $\lceil \log h \rceil + \lceil \log i \rceil + 1$ ($\lceil \log h \rceil$ bits are needed to identify the first past phrase, $\lceil \log i \rceil$ bits contain the length of the current phrase and the last bit is needed for the terminal symbol).

In the D-parse, we waive the requirement that the copying process be terminated during some *past* phrase, i.e., we have now:

$rep_j = head_j$

Example: $D-P(x)$ = 1-1111111111111111110-1110-111011101110-10- 1010100-0-100-0100$

phrases = 9

$|id(j)| = [\log k] + [\log(n-i+1)] + 1$ (this has an interpretation similar to that of scheme C, except that the length of the current phrase then exceed the i bits).

The suffix in the E-parse is exactly the same as for the D-parse except that it is now $r > n$. It follows that it now $rep_j = rep_i = head_i$.

Example: $E-P(x)$ = 1-1111111111111111110-1110111011101111010-1010100-01000-100$

phrases = 6

$|id(j)| = [\log i] + [\log(n-i+1)] + 1$ (the copying process may now start at any past position).

It is readily seen that the instantiations $A-D$ of *parse* can be set up to run in linear time.

Other variations and applications are discussed elsewhere in this book [MW,LZ1], along with a broader survey of data compression [ST], and novel compression methods [FK] for sparse bit strings. Intermediate characterizations for the set Q were introduced in [AGU]. Efficient ways of dealing with buffers of limited sizes [ZL] are presented in [RPE].

3. SQUARES IN A WORD

A *square* of x is a word on the form ww, where w is a *primitive* word, i.e., a word that cannot be expressed in any way as v^k with $k > 1$. Square free words, i.e., words that do not contain any square subwords have attracted attention since the early works by A. Thue in 1912 [TH]. A copious literature, impossible to report here, has been devoted to the subject ever since.

By keeping special marks to all nodes leading to suf_1 it is possible to spot all square prefixes of x as a byproduct of the construction of T_x. The same straightforward strategy can be used for square suffixes. On the other hand, devising efficient algorithms for the detection of (all) squares has required more efforts [ML,CR,AP]. The number of distinct occurrences of squares in a word can be $\Theta(n \log n)$, which sets a lower bound for all algorithms that find all squares [CR]. For instance, infinitely many *Fibonacci words*, defined by:

$$w_0 = b ; w_1 = a$$

$$w_{m+1} = w_m w_{m-1} \ for \ m > 1$$

have $O(n \log n)$ distinct occurrences of square subwords. Interestingly enough, by following the proof in [CR] as a guideline and making use of the fact that cyclic permutations of a primitive word are also primitive, it is not difficult to show that, for $m \geq 4$, the number S_m of different square subwords in w_m is such that $S_m \geq 1/12$

$(|w_m| \log |w_m|)$. This fact is of some consequence in trying to assess the space needed for the allocation of the statistics without overlap of all subwords of a text-string [AP1]. We show now that the E-parse $\mathrm{prod}_1 \mathrm{prod}_2 \cdots \mathrm{prod}_k$ of a string x, can be used nicely as a filter to spot the leftmost occurring nontrivial square of x. Our approach is similar to the one in [CR1]. In this context, a square is *trivial* if it is a suffix of prod_j for some $j \in \{1,2,...,k\}$ (which takes, trivially, overall linear time to spot), or if it is detected following the situation described below.

For $j \in \{1,2,...,k\}$, let $\mathrm{prod}_j = \mathrm{rep}_j \cdot a$ with $a \in I \cup \{\$\}$. Now prod_1 is obviously squarefree. Assume $\mathrm{prod}_1 \mathrm{prod}_2 \cdots \mathrm{prod}_{j-1}$ square free and let l be its length. Then if $|\mathrm{rep}_j| \geq l$, there is a square in $\mathrm{prod}_1 \cdots \mathrm{prod}_{j-1} \mathrm{rep}_j$, due to two occurrences of rep_j that either overlap or are contiguous. This circumstance can be easily detected on line with carrying out the E-parse of x, hence in linear time, and we shall say that such square is trivial too.

A few more definitions are needed in order to illustrate the full criterion. We say that two subwords w and $w\cdot$ of x satisfy the *left* (*right*) *property*, denoted $l(w,w\cdot)$ ($r(w,w\cdot)$), if $ww\cdot$ are squarefree but $ww\cdot$ embeds a square vv centered to the left (right) of $w\cdot$. Let x be a string with no trivial square. Then:

> *x is not squarefree iff there is* $i \in (1,2,...,k-1)$ *such that:* $l(\mathrm{prod}_i\, \mathrm{prod}_{i+1})$ *or* $r(\mathrm{prod}_i\, \mathrm{prod}_{i+1})$ *or* $r(\mathrm{prod}_1 \mathrm{prod}_2 ... \mathrm{prod}_{i-1},\ \mathrm{prod}_i\, \mathrm{prod}_{i+1})$.

To prove this claim, let yvv be the shortest non squarefree prefix of x and let j be the smallest index for which yvv is a prefix of $\mathrm{prod}_1 \cdots \mathrm{prod}_{j+1}$. Under our assumptions, it suffices to show that the second occurrence of v must fall entirely within $\mathrm{prod}_j\, \mathrm{prod}_{j+1}$. But this follows at once from the definition of rep_j. Indeed, if the second occurrence of v does not fall within $\mathrm{prod}_j\, \mathrm{prod}_{j+1}$ then rep_j would be contained in the second occurrence of v without being a suffix of v, a contradiction.

The left and right properties can be checked in overall linear time with the aid of auxiliary 'local' subword trees, or simply by resorting to the 'failure function' [AH]. We leave this as an exercise for the reader. An alternative procedure for testing squarefreeness [ML1] and a simple and elegant probabilistic algorithm for this problem [RA] are both discussed elsewhere in this book.

We turn now to the problem of finding all squares in a word. The use of subword trees in this task is brought up by the following fact [AP].

> *x contains a square occurrence at position i iff there is a primitive word $w \in V_x$ and a vertex α in T_x such that i and $j = i + |w|$ are consecutive leaves in the subtree of T_x rooted at α and furthermore* $|w(\alpha)| \geq (i-j)$.

The algorithmic criterion provided by the above condition is implemented straightforwardly in a bottom up computation. Starting from the leaves of T_x, for each interior vertex visited we construct the sorted list of the labels of its leaves. The sorted list of any such vertex is obtained by merging the sorted lists of its offspring vertices. The strategy runs in $O(n \log n)$ time if T_x is nearly balanced or completely unbalanced. Optimal handling of intermediate cases involves pebbling of T_x with an *ad hoc* data structure suited to the efficient repeated merging of integers in a known range [AP].

We devote the remainder of this section to highlight that the structure of T_x may help disclosing general properties about power subwords in a string [AA]. For instance, unlike the number of squares, the number of distinct cube subwords of any string x is bounded by n. To show this, we introduce the notion of cube constrained word (CCW) as follows: we say that $ww \in V_x$ is *cube constrained* if $w^3 \notin V_x$. It is seen [AA] that the number of distinct CCW's in any string x is bounded by n. In order to prove this fact, one first uses the definition of T_x to show that if $w^{k+1} (k \geq 1)$ is a subword of x, then w^k and w^{k+1} have distinct loci in T_x. Next one uses this in conjunction with the *periodicity lemma* [LS] to show that if w^2 and v^2 are distinct CCW's of x, then they must have distinct loci in T_x. The assertion follows then from the fact that the number of interior vertices of T_x is bounded by n.

4. STATISTICS WITHOUT OVERLAPS

The (primitive rooted) squares in V_x have consequences on the amount of storage needed to allocate the statistics without overlap of all substrings of x [AA,AP1], which leads us to another application of T_x. Consider the *weighted vocabulary* (V_x, C') where C' associates, with each $w \in V_x$, the maximum number k of distinct occurrences of w such that it is possible to write $k = w_1 w w_2 w w_3 \cdots w w_{k+1}$ with w_d possibly empty ($d = 1,2,...,k+1$).

The construction of (V_x, C') requires in general augmenting T_x [AP1] by inserting auxiliary nodes of degree 1. The role of such nodes in the augmented tree is to function as proper loci for subwords whose loci in the original tree T_x would not report the actual number of their nonoverlapping occurrences. To be more precise, assume that all nodes in the tree of Fig. 1 are weighted with their associated C' values. Now ab occurs 8 times in w_7, the word of Fig. 1; but the locus α of ab has $w(\alpha) = aba$ with a $C' = 5$. In order for the tree to report the appropriate C' value for ab we have to split an edge and create the proper locus for this subword. Let $\bar{T}_x$ be the minimal (i.e., with the least auxiliary nodes) augmented subword tree. The following fact gives a handle in establishing where the auxiliary nodes should be inserted in T_x in order to produce $\bar{T}_x$ [AP1].

> *If α is an auxiliary node of $\bar{T}_x$, then there are subwords u, v in x and an integer $k \geq 1$ such that $w(\alpha) = u = v^k$ and there is an $w \in V_x$ such that $w = v^m v\cdot$ with $v\cdot$ a prefix of v and $m \geq 2k$.*

An $O(n\log n)$ upper bound on the number of auxiliary nodes needed in $\bar{T}_x$ can be readily set, based on the above fact and on the upper bound on the number of positioned squares in a word. However, it seems to be an interesting open question whether there are words whose minimal augmented suffix trees do in fact attain that bound. The insertion of candidate auxiliary nodes can be carried out during the brute force construction of T_x, after which redundant nodes can be removed through one visit of the structure. Hence $\bar{T}_x$ can be obtained in $O(n^2)$ time, almost straightforwardly. A more efficient construction is also more elaborate [AP2], and we shall not attempt at reporting it here.

5. CONCLUDING REMARKS

Since subword trees embody remarkably structured information about the word(s) they are built out of, it is not surprising that they can be used in a variety of tasks that either aim at retrieving some such information or make crucial use of it in answering disparate queries. Sometimes there are better methods than those based on such trees, however, no digital index seems to outperform subword trees in versatility and elegance.

For instance, the subword tree associated with $y = x\#x^r\$$ can be used to detect all palindrome subwords of x, in $O(n\log n)$, by repeated bottom up merging of leaves (as with the detection of squares) and by making use of the fact that any palindrome in V_x must have a proper locus in T_y, as the reader may check for himself. As is well known, there are linear time solutions for this problem (see for instance [MA]).

Similarly, the subword tree associated with a set of m words of total length l can be adapted to test the *unique decipherability* of the code consisting of those words in $O(m \cdot l)$ time [RO]. However, the same performance can be achieved by a simpler construction, based on pattern matching machines [AC], as shown in [AG]. The subject of unique decipherability testing is also addressed elsewhere in this book [CH]. The relation between subword trees and pattern matching machines is investigated in [CR2].

Acknowledgements

Joel Seiferas kindly supplied some of the reference material. Zvi Galil and Sam Wagstaff made many very helpful comments on a preliminary version of this paper.

References

AC Aho, A., and Corasick, M.J., Efficient String Matching: An Aid to Bibliographic Search, *CACM* 18, 335-340 (1975).

AH Aho, A., Hopcroft, J.E., Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading (1974).

AA Apostolico, A., On Context Constrained Squares and Repetitions in a String, *R.A.I.R.O. Journal Theoretical Informatics* 18, 2, 147-159 (1984).

AG Apostolico, A., Giancarlo, R., Pattern Matching Machine Implementation of a Fast Test for Unique Decipherability, Inf. Proc. Letters 18, 155-158 (1984).

AGU Apostolico, A., Guerrieri, E., Linear Time Universal Compression Techniques for Dithered Images Based on Pattern Matching (extended abstract), *Proceedings of the 21st Allerton Conference on Communication, Control and Computing*, 70-79 (1983).

AP Apostolico, A., Preparata, F.P., Optimal Off-line Detection of Repetitions in a String, *Theoretical Computer Science*, 22, 297-315 (1983).

AP1 Apostolico, A., Preparata, F.P., The String Statistics Problem, Tech. Report, Purdue Univ. CS Dept. (1984). A preliminary version: A Structure for the Statistics of all Substrings of a Textstring With and Without Overlap, *Proceedings of the 2nd World Conference on Math. at the Service of Man*, 104-109 (1982).

BL Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., McConnell, R., Building a Complete Inverted File for a Set of Text Files in Linear Time, *Proceedings of the 16th ACM STOC*, 349-358 (1984).

BE Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., McConnel, R., Building the Minimal DFA for the Set of All Subwords of a Word On-line in Linear Time, *Springer-Verlag Lecture Notes in Computer Science* 172, 109-118 (1984).

CH Capocelli, R.M., Hoffmann, C.H., Algorithms For Factorizing and Testing Subsemigroups, *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.) Springer-Verlag (1985).

CR Crochemore, M., An Optimal Algorithm for Computing the Repetitions in a Word, *Inf. Proc. Letters* 12, 5, 244-250 (1981).

CR1 Crochemore, M., Recherche Lineaire d'un Carre dans un Mot, *C.R. Acad. Sc. Paris*, t.296, Serie I, 781-784 (1983).

CR2 Crochemore, M., Optimal Factor Transducers, *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.) Springer-Verlag (1985).

CS1 Chen, M.T., Seiferas, J., Additional Notes on Subword Trees, unpublished lecture notes (1982).

CS2 Chen, M.T., Seiferas, J., Efficient and Elegant Subword Tree Construction, *Combinatorial Algorithms on Words* (A. Apostolico and Zvi Galil, eds.), Springer-Verlag (1985).

FK Fraenkel, A. S., Klein, S. T., Novel Compression of Sparse Bit Strings, *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.) Springer-Verlag (1985)

KMP Knuth, D.E., Morris, J.H., Pratt, V.R., Fast Pattern Matching in Strings, *SIAM Journal on Computing* 6, 2, 323-350 (1977).

KMR Karp, R.M., Miller, R.E., Rosenberg, A.L., Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays, *Proceedings of the 4th ACM STOC*, 125-136 (1972).

KN Knuth, D.E., *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*, Addison-Wesley, MA (1973).

LS Lyndon, R.C., Schützenberger, M.P., The Equation $a^M = b^N c^P$ in a Free Group, *Michigan Math. Journal* 9, 289-298 (1962).

LZ Lempel, A., Ziv, J., On the Complexity of Finite Sequences, *IEEE TIT* 22, 1, 75-81 (1976).

LZ1 Lempel, A., Ziv, J., Compression of Two-dimensional Images, *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.) Springer-Verlag (1985).

MA Manacher, G., A New Linear-time On-line Algorithm for Finding the Smallest Initial Palindrome of a String, *JACM* 22, 346-351 (1975).

MR Majster, M.E., Reisner, A., Efficient On-line Construction and Correction of Position Trees, *SIAM Journal on Computing* 9, 4, 785-807 (1980).

MC McCreight, E.M., A Space Economical Suffix Tree Construction Algorithm, *JACM* 23, 2, 262-272 (1976).

ML Main, M.G., Lorentz, R.J., An O(nlogn) Algorithm for Finding all Repetitions in a String, *Journal of Algorithms*, 422-432 (1984).

ML1 Main, M.G., Lorentz, R.J., Linear Time Recognition of Square-Free Strings, *Combinatorial Algorithms on Words*, (A. Apostolico and Z. Galil, eds.) Springer-Verlag (1984).

MO Morrison, D.R., PATRICIA – Practical Algorithm to Retrieve Information Coded in Alphanumeric, *JACM* 15, 4, 514-534 (1968).

MW Miller, V.S., Wegman, M.N., Variations on a Theme by Ziv and Lempel, *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.) Springer-Verlag (1985).

PR Pratt, V.R., Improvements and Applications for the Weiner Repetition Finder, unpublished manuscript (1975).

RA Rabin, M.O., Discovering Repetitions in Strings, *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.), Springer-Verlag (1985).

RO Rodeh, M., A Fast Test for Unique Decipherability Based on Suffix Trees, *IEEE TIT* 28, 648-651 (1982).

RPE Rodeh, M., Pratt, V.R., and Even, S., Linear Algorithms for Data Compression via String Matching, *JACM* 28, 1, 16-24 (1981).

SE1 Seiferas, J., Subword Trees, unpublished lecture notes (1977).

SL Slisenko, A.O., Detection of Periodicities and String Matching in Real Time, *Journal of Soviet Mathematics* 22, 3, 1316-1387 (1983).

ST Storer, J.A., Textual Substitution Techniques for Data Compression, *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.) Springer-Verlag (1985).

SZ1 Seery, J.B., Ziv, J., A Universal Data Compression Algorithm: Description and Preliminary Results, Bell Labs TM77-1212-6/77-1217-6 (1977).

SZ2 Seery, J.B., Ziv, J., Further Results on Universal Data Compression, Bell Labs, TM78-1212-8/78-1217-11 (1978).

TH Thue, A., Uber Die Gegenseitige Lage Gleicher Teile Gewisser Zeichenreichen, Skr. Vid. Kristiana I. Math. Naturv. Klasse 1, 7-67 (1912).

WE Weiner, P., Linear Pattern Matching Algorithms, *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, 1-11 (1973).

ZI Ziv, J., Coding Theorems for Individual Sequences, *IEEE TIT* 24, 4, 405-413 (1978).

ZL Ziv, J., Lempel, A., A Universal Algorithm for Sequential Data Compression, *IEEE TIT* 23, 3, 337-343 (1977).

ZL1 Ziv, J., Lempel, A., Compression of Individual Sequences On Variable Length Encoding, *IEEE TIT* 24, 5, 530-536 (1978).

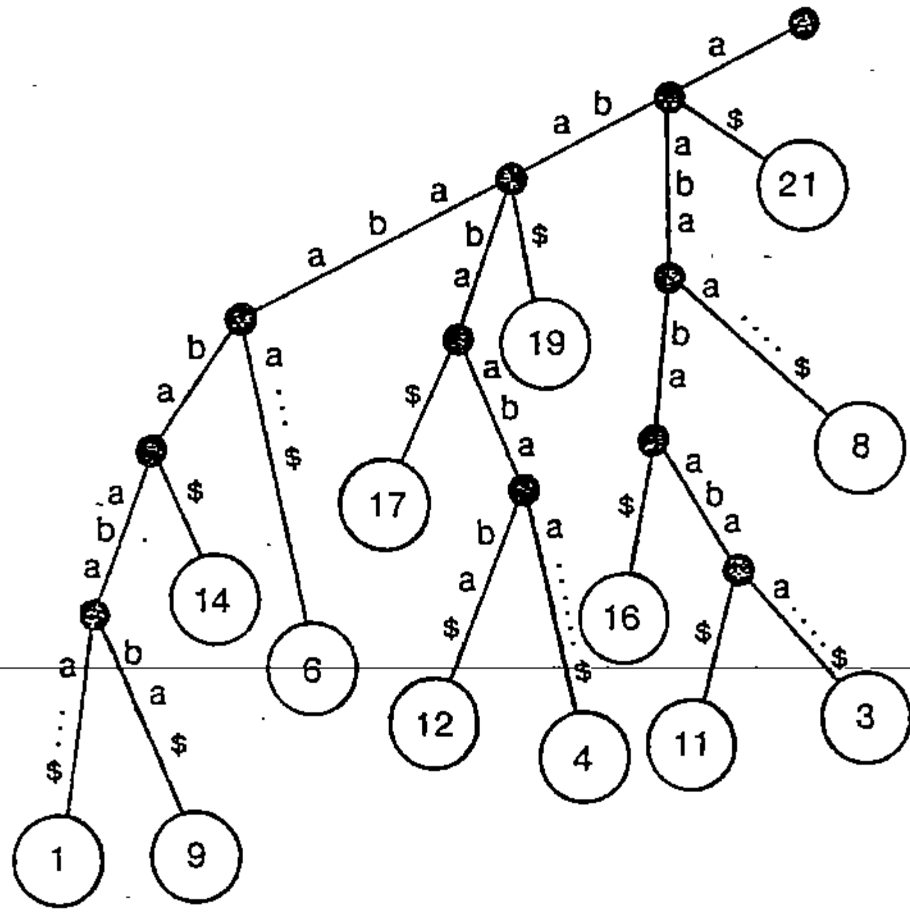

Figure 1

A partial view (all suffixes starting with *a*) of the subword tree of the string *abaababaabaaba-baababa.*