Department of Computer Science Technical Reports

Department of Computer Science

1985

Data Structures and Algorithms for the String Statistics Problem

A. Apostolico

F. P. Preparata

Report Number:
85-547

DATA STRUCTURES AND ALGORITHMS FOR THE STRING STATISTICS PROBLEM

A. Apostolico

F. P. Preparata
University of Illinois

CSD-TR-547
October 1985

DATA STRUCTURES AND ALGORITHMS FOR THE STRING STATISTICS PROBLEM

(Preliminary Report)

A. APOSTOLICO
Department of Computer Science
Purdue University
West Lafayette, IN 47907

F.P. PREPARATA
Coordinated Sciences Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

October 1985

ABSTRACT

Given a textstring x of length n, the MAST (*minimal augmented suffix tree*) of x is a digital index from which the number of nonoverlapping occurrences in x of any substring w of x can be retrieved in a number of comparisons proportional to the length of w. It is shown here that the MAST of x can be constructed in time $O(n \log^2 n)$ and space $O(n \log n)$, off-line on a RAM.

Key words and Phrases: statistics in a textstring, pattern matching, suffix tree, augmented suffix tree, MAST, period of a string, repetition in a string, abacus, design and analysis of algorithms.

1. INTRODUCTION

Let I be a finite alphabet and x a *string* (*word*) of *lenght* $|x| = n$ on I. A suitably weighted *subword tree* [AA] of x, such as for instance the *suffix tree* [MC] of x, can be easily adapted to serve as an index which enables to retrieve, for any given *substring* w of x and in $O(|w|)$ comparisons, the number of distinct *occurrences* of w in x. The preparation of such an index is accomplished easily in time and space linear in n, off-line on a RAM. Indeed, the suffix tree or any of its variations can be constructed in linear time by clever methods (see, for instance, [MC]; the complete set of references is given in [AA]). Having built the tree, it is sufficient to traverse it in postorder weighting each of its nodes with the number of leaves which are found in the subtree rooted at that node (see Fig. $1a$ for an example).

In this paper we approach the construction of an index which is organized and used along the same lines as the above one, except that the novel structure must report, for any substring w of x, the value $C(w)$ which represents the maximum number of distinct *nonoverlapping* occurrences of w in x. For instance, *aba* occurs 8 times in *abaababaabaababaababa*, but $C(aba)$ is only 5. As pointed out in [AR], the discrepancy between these two types of *statistics* for a substring w is always attributable to some *periodicity* of w (in the above example a is both a prefix and suffix of *aba*). In these cases, moreover, there may be more than one maximal (i.e., of cardinality $C(w)$) set of nonoverlapping occurrences of w in x. In correspondence with any word w, we call *compact* w*-tagging* (sometimes, shortly w*-tagging*) of x the unique maximal set G of nonoverlapping occurrences of w in x produced by a greedy left-to-right scansion of x and defined as follows. The leftmost occurrence of w in x is a member of G; if the occurrence of w starting at i is in G, then the earliest occurrence of w which starts at some $j \geq i + |w|$ is also in G. Thus, in order to compute $C(w)$, it suffices to construct the compact w-tagging of x and then to report the cardinality of this set. Along these lines, it can be seen that the entire weighted index can be obtained from scratch in $O(n^2)$ time and $O(nlogn)$ space [AR].

It might intrigue the reader that linear space seems now no longer sufficient. As the following example illustrates, however, it is in fact fortunate that a better bound than $O(n^2)$ can be drawn for the space. In the suffix tree of Fig. $1a$, node μ, i.e., the *locus* [AR] of *ab*, correctly reports the number of occurrences of *ab* in the textstring, even though μ is not the *proper locus* of *ab* (i.e., the path from the root labeled *ab* terminates in the middle of an arc). This is no longer true if each node α of that suffix tree is weighted with $C(W(\alpha))$, where $W(\alpha)$ is the substring of x whose proper locus is α. Indeed, μ would be given the weight $C(aba) = 5$ in this case, while the statistics without overlaps for ab is still 8. To overcome this, *auxiliary* unary nodes have to be inserted in the suffix tree to function as proper loci for substrings whose locus in the original tree could not consistently report their C−value.

As Fig. 1*b* suggests, however, it is not necessary to change all arcs into chains (which would result in quadratic worst case space requirement) in order to accommodate all necessary additional weights. In fact, the space required by the *minimal augmented suffix tree* (MAST), i.e., the tree the auxiliary nodes of which are all and only the necessary ones, can be bounded by $O(nlogn)$ [AR]. However, it is still a noteworthy open problem whether there are strings which attain this bound. Some special cases of strings with an $O(n)$ nodes MAST are discussed in [AO]. Fig. 1*b* displays portion of the MAST associated with our example string.

Most of the criteria underlying an efficient construction of the MAST of a string were previously presented in [AR]. To render the present paper self-contained, we recall here and in the following section some basic notions and the main facts from that reference.

An integer p is a *period* of w if $w[i]=w[i+p]$ $(i=1,2,...|w|-p)$. A string w is *periodic* if it has a period of size not larger than $|w|/2$. A string w is *primitive* if setting $w=u^k$ implies $u=w$ and $k=1$. A *repetition* in x is a positioned substring $x[i,m]$ for which there are indices j,d $(i<d\leq j\leq m)$ such that: $x[i,j]$ and $x[d,m]$ are occurrences of the same word, $x[i,d-1]$ corresponds to a primitive word (the *root*), and $x[j+1]\neq x[m+1]$. Thus, a repetition is the occurrence of a periodic substring in the form $(st)^k s$ where $k>1$, $s\in I^*$ and $t\in I^+$; as such, it is completely identified by the triple of its starting position i, its period $p=d-1$, and its length $L=m-i+1$, respectively, and is denoted by the symbol $R(i,p,L)$. $R(i,p,L)$ is a *maximal repetition* if $i-p$ is not the starting position of the repetition $R(i-p,p,L+p)$.

Repetitions are responsible for the additional nodes which need to be inserted in T_x, the suffix tree of x, in order to convert it into the MAST $\hat{T}_x$.

FACT 1 [AR]: If α is an auxiliary node of $\hat{T}_x$, then there are substrings u, v of x and an integer $k \geq 1$ such that $W(\alpha) = u = v^k$ and there is a repetition in x in the form $v^m v'$ with v' a prefix of v and $m \geq 2k$.

It is shown in [AP] that the suffix tree T_x itself can be used to detect in optimal $O(n \log n)$ time and space $O(n)$ - all distinct repetitions in x. This is done by a rather sophisticated bottom-up merge of leaves in T_x and by exploiting the following simple property of T_x:

FACT 2 [AP]: $R(i,p,L)$ is a repetition of x if and only if there is a vertex α in T_x such that $|W(\alpha)| \geq p$, where i and $j = i+p$ are consecutive leaves in the subtree of T_x rooted at α.

2. RUNS, CHUNKS AND NECKLACES

Facts 1 and 2 give a handle in spotting all candidate auxiliary nodes. We now recall from [AR] other facts pertaining to the structure and evolution of compact taggings, which set forth our algorithmic criteria.

Let $S(\alpha)$ be the ordered sequence of leaves in the subtree of T_x rooted at vertex α, and let i and j be two elements of $S(\alpha)$, with $i < j$. *Segment* i is the occurrence of $W(\alpha)$ starting at i. Segments i and j are said to *overlap* if $j - i < |W(\alpha)|$. Moreover, if segments i and j overlap and are consecutive, then i is called the *origin* for j and j is called the *detector* for i. Notice that two overlapping segments need not be consecutive in $S(\alpha)$, and that a segment can be an origin and a detector at the same time.

Definition 1: If $i_0 i_1 \cdots i_{s+1}$ is a substring of $S(\alpha)$ and $i_1 - i_0 \geq |W(\alpha)|$, $i_{j+1} - i_j < |W(\alpha)|$ for $j = 1,2, \ldots, s-1$, and $i_{s+1} - i_s \geq |W(\alpha)|$ the sequence of segments $i_1, i_2, \ldots, i_s$ is a *run*.

Segments i_1 and i_s are called, respectively, the *head* and the *tail* of the run. The *span* of the run is the interval which is the union of the segments in the run. Within a run we single out the following important subsequence of segments:

Definition 2: A *necklace* is a maximal subsequence of segments in a run such that only consecutive segments overlap.

Each necklace is considered as an alternating sequence of *master* and *slave* segments, the first term being a master. Also, a necklace is *odd* or *even* depending upon the parity of the number of its segments. Note that, by the definition of necklace, two consecutive segments of a necklace cannot be disjoint occurrences of $W(\alpha)$. If $W(\alpha)$ is periodic, then there may be leaves of $S(\alpha)$ falling within the span of a necklace that do not belong to it. This suggests consideration of a new type of subassembly in a run which is quite relevant to our objectives.

Definition 3 [GA]: A *chunk* is a maximal substring of a run such that $i_j - i_{j-1}$ $(j = 2,3,\ldots,s)$ is not larger than $p < |W(\alpha)|/2$.

The following are useful properties of chunks.

FACT 3 [GA]: If $W(\alpha)$ has minimum period $p < |W(\alpha)|/2$, then the difference between the starting positions of two consecutive segments in a chunk is exactly p.

FACT 4 [AR]: Although a given run may contain more than one chunk, two consecutive chunks at α may overlap at most for $q < p$ positions, where $p < |W(\alpha)|/2$ is the minimum period of $W(\alpha)$.

Facts 3 and 4 are a consequence of the well known *Periodicity Lemma* [LS], which states that if a word w has periods p and q and $|w| > p+q$, then w has period *g.c.d.* (p,q).

FACT 5 [AR]: Let i_j $(j = 1,2)$ be the leftmost master segment in a chunk, and let $\bar{C}$ be the number of segments contributed by the chunk to the $W(\alpha)$-tagging of x.

$$\bar{C} = \begin{cases} \left[\dfrac{L_j}{|W(\alpha)} \right] & \text{if } W(\alpha) = u^k \text{ with } u \in I^+ \\ \left[\dfrac{L_j + |t|}{|W(\alpha)| + |t|} \right] & \text{if } W(\alpha) = (st)^k s \text{ with } s,t \in I^+ \end{cases}$$

with:

$$L_j = \begin{cases} L & \text{if } j = 1 \\ L - p & \text{if } j = 2 \end{cases}$$

The notions of *head*, *tail* and *span* extend quite naturally to necklaces and chunks. Thus $\bar{C}$ can be retrieved from the knowledge of the *status* (=master, slave) of the chunkhead and of the span of the chunk.

FACT 6 [AR]: The compact $W(\alpha)$-tagging of x consists of all and only the master segments in all necklaces at α. In particular, $C(W(\alpha))$ can be computed by counting, in $S(\alpha)$, the number of master segments that belong to a necklace, but not to a chunk, and then by adding to the value thus obtained the contribution of each individual chunk.*

* We remark that, by the definition of necklace, if $i_1, i_2 \cdots i_k$ are consecutive segments w with $|w| > (i_j - i_{j-1})$, $j = 1,2,\ldots,k-1$, then the construction of necklaces for all segments w of x partitions the set $I \equiv \{i_1, i_2 \cdots i_k\}$ into two subsets I_1 and I_2 (with I_2 possibly empty) so that the segments of I_1 are all in the same necklace and those in I_2 are

Very succinctly, we construct $\dot{T}_x$ from T_x as follows. We visit T_x in postorder. At each original node α, we obtain the related necklace-structured set of segments by an appropriate merge of the structured sets associated with the offsprings of α. Fact 6 gives a handle in computing $C(W(\alpha))$ as a trivial byproduct of such merge. Facts 1,2 are used instead to schedule the examination of perspective auxiliary nodes at appropriate later stages of the traversal. We need a data structure tailored to the efficient management of dynamic runs, chunks and necklaces. This will be discussed in next section.

3. SECTS AND ABACUSES

Let w be a substring of x, and let $\bar{S} \equiv \{i_1, i_2, ..., i_k\}$ be the ordered set of the segments relative to all the occurrences of w in x. A subset $S \subseteq \bar{S}$ is a *sect* if, for any segment i in a chunk of $\bar{S}$ such that s is not the tail of a chunk of $\bar{S}$, the fact that i is in S implies that all the segments preceding i in the chunk of $\bar{S}$ are also in S. In particular, the set $\bar{S}$ is clearly a sect. We call this sect the w*−maxsect* of x.

Example: Let $x = ababa\ babab\ aabab\ ab$ and $w = abab$. Then $S = \{1,5,12\}$ is not a sect, since segment 5 is in S but segment 3 is not. The set $\{1.3,5,12\}$ is a sect, as are the sets $\{1,12\}$, $\{1,3,7\}$, $\{7\}$, $\{1, 7, 12\}$, etc.

Let S be a sect of w-segments and i a w-segment not in S. Segment i is S*−compatible* iff $S \cup \{i\}$ is still a sect and if, moreover, the following condition holds: if i is the tail of a chunk of $\bar{S}$, then either all the other segments of that chunk are already in S, or none of such segments is. Let now S be the w−maxsect of x. The (*legal*) *contraction* u of w is the (possibly empty) prefix of w identified by the following rule. Let v be the longest one among the prefixes of w which satisfy, with integers L,p,k, the following properties: $|p| = pk$, p is a period of w and there is a repetition in x of prefix w and length $L \geq (k+1)p$. If the v−maxsect of x is described by the same set of labels as the w−maxsect of x, then $u = v$. Otherwise $u = y$, where y is the longest prefix of w such that the y−maxsect is larger than the w*−maxsect*.

Example: Let $x = abaab\ aabaa\ baaba\ abaaa\ baaba\ ab$ and $w = abaabaab$. Consider the sect $S = \{1,20\}$. Segment 7 is not S−compatible, since $\{1,7,20\}$ is not a sect (due to the occurrence of w starting at position 4). On the other hand, segment 4 is clearly S−compatible. The maxsect of x relative to our w is $\bar{S} \equiv \{1,4,7.10.20\}$. This set of labels completely describes also the exhaustive sect of x relative to $u = abaabaa$. However, 7 is not the period of a repetition in x, and thus u is not the legal contraction of w.

deleted. In particular, the segments in a chunk cannot be partitioned between two distinct necklaces.

Indeed, the contraction of w is $abaaba$, since the maxsect of this latter string is in fact $\bar{S} \cup \{13\}$.
In the example string of Fig.1, on the other hand, $u=ab$ is the contraction of $w=aba$, since the two associated maxsects are described by the same set of labels and there is a repetition in that string of prefix aba and length $L \geq 4$.

Informally, the *abacus* $A(S)$ associated with a sect S of w-segments is a logically structured collection of subsequences of the integers in $[1,n]$, which provides all the information relative to the w-tagging of $x(S)$, and is dynamically updatable to reflect: (1) the addition to S of any S–compatible w-segment, and (2) if S is the maxsect, the deletion of any suffix of w which changes w into its legal contraction u.

Each one of the subsequences of $[1,n]$ which concur in the formation of the abacus is logically structured, in its own turn, in the way best fit to the repertoire of primitive operations which are to be performed on it. We describe the richest such structure, which is used to allocate a necklace of segments in the sect S, and which we shall call *Concatenable Queue with Parity* (PCQ). In practice, a PCQ can be implemented as the slight upgrade, say, of a 2-3 tree [AH]. The leaves at the bottom of such a tree are assigned as follows. Form left to right, each segment in the necklace which does not simultaneously belong in a chunk is stored in a leaf, and a leaf is also assigned to the chunkhead of each chunk falling within the span of the necklace.

As is well known, each of the instructions *SEARCH*, *INSERT*, *DELETE*, *MIN*, *CONCATENATE* and *SPLIT* (we adopt the formalism in [AH] throughout) can be executed on a 2-3 tree with m leaves in $O(\log m)$ time. By doubly linking the leaves of the tree, the above repertoire is trivially extended to contain the constant-time primitives *PRED* (predecessor) and *SUCC* (successor). Maintaining a pointer from each node to its father enables an $O(\log m)$-time *FIND* operation, which returns the root of the tree (necklace) containing a given segment. In addition, each segment in a PCQ is given a *rank* of 0 or 1 according to the following. Ordinary segments have rank '1'; the rank of a chunkhead equals the number of chunk segments currently in the necklace. Correspondingly, the left (middle, right) *rank* of an internal node is equal to the sum of the ranks of the leaves in the left (middle, right) subtree of that node. It is an easy exercise to show that the ranks of nodes and leaves can be maintained without penalty during dynamic updates of the tree, within the logarithmic time bound. In particular, a PCQ Q with m leaves can be updated to reflect an unit increase in the rank of chunkhead i through a suitable primitive $RANK(i,Q)$, in $O(\log m)$ time. Again, we omit the details. The primitive $STATUS(i)$ is devised to return the *status* (master or slave) of a leaf i in the

PCQ.

Clearly, $status(i)=master$ if the sum of the ranks of all leaves of Q not larger than i is odd, $status(i)=slave$ otherwise. $STATUS(i)$ is easily engineered along the lines of a search: each time we leave a node through the $l-th$ branch (l =1,2,3), we add (mod. 2) the contents of the rank fields of this node up to the $(l-1)-st$ to a suitable counter k. At the end, $status(i)=master$ if $k=1$. Notice that the rank of the rightmost segment in the tree yields the parity of the necklace.

We summarize our discussion of PCQs by recording the following.

FACT 7: There is an implementation of PCQs on the integers in $[1, n]$ which supports each of the PCQ primitives in $O(\log n)$ time.

The collection of PCQs relative to the various necklaces in S form the TAG section of $A(S)$. The abacus contains the following additional sections.

2. The INDEX section: this is simply a 2-3 tree the leaves of which represent all segments of S. The index supports only $INSERT, PRED, SUCC$. It is also maintained in such a way that the head of each chunk stores the value of the span of the chunk, and that such segment is directly accessible from each other segment in the chunk.

3. The ROSARY section is formed by 2-3 tree implementations of dictionaries. The main such dictionary (the ROSARY INDEX) collects the w-rep periods of w: p is a $w-rep$ period of w if p is a period of w, no fraction of p is a period of w, and there are two w-segments in S whose starting positions differ by p. (Hence there is a repetition of period p and prefix w in x.) In addition, each w-rep period p such that $p \geq |w|/2$ is assigned a dictionary, the $p-weave$, whose leaves point in succession to all segments that are origins of period p in S. Finally, the (unique) period $q < |w|/2$, if it exists, is assigned a special dictionary, the $chunk-weave$, that collects pointers to the chunkhead of each chunk.

4. The COUNTER C which stores the cardinality of the compact w-tagging of x which uses only segments of S.

The storage required by the index and tag portions of the abacus associated with S is obviously $O(|S|)$. Finally, by Fact 2, a segment cannot simultaneously belong to two different p-weaves. Thus we have, in conclusion:

FACT 8: The total space required by the abacus $A(S)$ associated with a sect S is $O(|S|)$.

4. THE PROCEDURE "UPDATE"

Let $\bar{S}$ be the w-maxsect of x, S a sect of w-segments of x, and i an S-compatible w-segment. With $S' = S \cup \{i\}$, we address the problem of constructing $A(S')$ from inputs $A(S)$ and i.

Lemma 1: Let, in S', $l = PRED(i)$ and $r = SUCC(i)$. If l overlaps with i and r, then r is the tail of a chunk of $\bar{S}$ and $(i-l)$ is the minimum period of the (periodic) string w.

Proof: Under the assumptions of the Lemma, either $(i-l)$ or $(r-i)$ (or both) must be smaller than $|w|/2$. It is then a consequence of the periodicity lemma that l, i and r are in a chunk of $\bar{S}$. Since S is a sect, r must be the tail of that chunk. Since i is S-compatible, then S' is also a sect. Since i is not the tail of the chunk of $\bar{S}$, then the predecessor of i in that chunk must be in S'. Such predecessor is l, by hypothesis, and $(i-l) < |w|/2$, by Fact 3. □

The conditions of Lemma 1 set forth a special case which shall be discussed later. We analyze now the cases where segment i does not fall entirely within the span of a run (whence, of a necklace) of segments of S.

Towards this end, it is convenient to define Λ, the empty necklace, whose parity is trivially even (zero). Thus the insertion in S of segment i always bridges two necklaces of S, η_1 and η_2, both of which may be empty. Since S is a sect, then, by Fact 4, i cannot overlap with two segments of η_2 at the same time. However, nothing prevents in principle i from overlapping with two segments in η_1. This case arises in fact if i joins in a chunk of segments of S. Assume that i overlaps with at most one necklace segment to its left. Then the two necklaces η_1 and η_2 are concatenated by segment i into a new necklace η, which contains all the segments of η_1 and η_2, in addition to segment i. We also have: parity(η) = [parity(η_1) + parity(η_2) + 1] (mod. 2). Notice that all segments retain their original parity, except when parity(η_1)=0, in which case the masters of η_2 become slaves and *vice versa*. We will use the following fact from [AR].

FACT 9: Assume that l and r do not overlap. Then Segment i belongs to the compact w-tagging of $x(S \cup \{i\})$ if and only if i does not simultaneously overlap with two necklace segments to its left and, moreover, its insertion bridges two even necklaces of S.

We are now ready to establish the following theorem.

Theorem 1: Let S be a sect and i an S-compatible w-segment. Then there is an algorithm which correctly transforms $A(S)$ into $A(S \cup \{i\})$.

Proof: We prove the claim by exhibiting the procedure *update* which produces $A(S \cup \{i\})$ from inputs $A(S)$ and an S-compatible segment i. We describe the action provoked by the call *update*$(A(S),i)$ through a case analysis. To simplify our notation, we use henceforth the word 'necklace' also to refer to the PCQ allocation of a necklace.

The first operation performed by *update* is *INSERT*$[INDEX,i]$, which inserts i in the index portion of $A(S)$. Let $l = PRED[INDEX,i]$ and $r = SUCC[INDEX,i]$, and assume that l and r do not overlap. The following cases may occur.

A. *i does not overlap with l nor r.* The segment i is a single-element-run of $S' = S \cup \{i\}$ and, trivially, the unique (master) segment in a new necklace η of S'. In $A(S)$, *update* initializes a PCQ for η and increments the COUNTER C by one. Since i does not overlap with any segment previously in S, then the repetitions in $x(S)$ are the same as those in $x(S')$. Thus the ROSARY of $A(S)$ is the same as that of $A(S')$. The procedure terminates having in fact produced this latter abacus.

B. *i is a detector for l but not an origin for r.* That is i joins the same run (and perhaps the same necklace η) of l. This case splits into two subcases, B1 and B2, depending on whether or not $(i-l) < |w|/2$.

B1. $(i-l) \geq |w|/2$. (cfr. Fig. 2a). Since S' is a sect, then i cannot overlap with l and $PRED(l)$. Indeed, this would imply that $PRED(l)$, l and i are in a chunk of $\bar{S}$, which also contains segments falling between l and i. But then i is not S-compatible. Thus l is either a *master* or a *slave* in a necklace into which i needs to be inserted. In addition, a pointer must reach l from the p-weave of the rosary with $p=(i-l)$. To obtain this, *update* simply performs the following.

```
B10:  $\eta \leftarrow$FIND[$l$]
B11:  INSERT[$\eta$,$i$]
B12:  if $status(i)=master$ then $C=C+1$; (i.e., use Fact 9);
B13:  WEAVE $\leftarrow$ SEARCH[$ROSARY-INDEX$,($i-l$)] (which returns the root of the
      ($i-l$)-weave, or prompts its creation if it did not already exit);
B14:  INSERT[$WEAVE$,$l$]; (that initializes the ($i-l$)-weave, if needed, and in any
      case issues a pointer to $l$ from the sorted list of leaves at the bottom of this
      weave.
```

The procedure can now return control, since all the sections of $\bar{A}(S)$ which were affected by the insertion of i have been correctly updated to reflect such insertion.

B2. $(i-l)<|w|/2$. Then l and i belong in the same chunk of S', although l is not necessarily also in a necklace of S'.

If l was not in a chunk of S prior to the insertion of i (see Fig. 2b), then i causes such a chunk to be detected and initialized. By Fact 4, either l or i is a master segment in the necklace within whose span both segments fall, and l is the head of the chunk. The details of chunk initialization are straightforward, and we shall not spend time on them. The COUNTER is given a unit increment if i is a master segment. Finally, a pointer to the chunkhead l is inserted in the chunk-weave.

If l was formerly in a chunk of S (Fig. 2c), then, by hypothesis, the chunkhead h is reachable through a pointer from l. The procedure copies the value of this pointer in the appropriate field attached to i. Clearly, the insertion of i in the chunk cannot change the chunkhead. However, such insertion might alter the rank of h (recall here that chunkheads have a special rank which accounts at once for the chunk contribution in the necklace). Now it is easy to retrieve $status(i)$ based on $|w|$, $(i-l)$ and the information stored in h. If i is either a *master* or a *slave* segment in its chunk, an appropriate call to *RANK* will provide for an unit increment of $rank(h)$. If i is a master in S' then the procedure also sets: $C \leftarrow C+1$.

This concludes the discussion of this case.

C. *i is an origin for r but not a detector for l.* This case splits into subcases C1: $((r-i) \geq |w|/2)$ (Fig. 2*d*), and C2: $((r-i) < |w|/2)$ (Fig. 2*e*), and it is similar to case B. It is handled in much the same way, with minor modifications. A distinguishing feature of this case is in that now r is always in a necklace of S as well as of S'. (I.e., if i is in a chunk of S', then it it also the chunkhead and a master segment in that chunk, whence r, which was a master in S, becomes a slave in S').

Case $C1$ is dealt with along the same lines as $B1$. For later reference, we only have to record the observation that the reverse task, namely, the task of producing $A(S)$ from $A(S \cup \{i\})$ under the conditions of case $C1$ is straightforwardly accomplished by means of searches and deletions. If case $C2$ applies, then combining the definition of a chunk with the the hypotheses that S is a sect and i is an S-compatible segment yields that r could not be the head of a chunk in S. Thus *update* does not face the problem of resetting the pointers to the chunkhead in all segments of a former chunk of S. However, it is easy to see that i and r are always, respectively, the head and the tail of a chunk of $\bar{S}$. If $(r-i)$ is the smallest period of w, then i and r are also the only two segments in that chunk of $\bar{S}$, which becomes now a chunk of S' as well. In any case, since r is the tail of a chunk, then $SUCC(r)$ cannot overlap with i. The procedure can exploit the observation which was made in connection with case C1 above. Namely, *update* creates a necklace for i (note: this can be regarded as the insertion of i in the empty necklace). Next, it extracts r from its necklace. To reinsert r in the abacus, the procedure now faces an instance of case C1.

D. *i is a detector for l and an origin for r.* Under our current hypothesis, l and r do not overlap, that is, r is always a master segment in S. Thus this case is partitioned into the three subcases below.

D1. $(i-l) \geq |w|/2$ and $(r-i) \geq |w|/2$,

D2. $(i-l) < |w|/2$ and $(r-i) > |w|/2$,

D3. $(i-l) > |w|/2$ and $(r-i) < |w|/2$.

In view of the discussion of cases B and C, it is immediately seen that *update* can produce the updated abacus in each case through the following.

D1: D11 - Follow the management of case B1 for segments i and l except the update of the COUNTER.

D12 - Let η_1 be the necklace into which i has been inserted and η_2 that containing r. Execute: $\eta \leftarrow$ CONCATENATE(η_1,η_2).

D13 - Use Fact 9 to update the COUNTER.

D2: - D21 is similar to D11, the main difference being that B2 is now used in place of B1. The stages D22 and D23, which correspond to D12 and D13, respectively, take place only if D21 returns that i is in a necklace of S'.

D3: - By the discussion of case C2, i and r must be, respectively, the head and the tail of a chunk of $\bar{S}$. Thus the action involved in this case is similar to the one taken in connection with case C2, except that the necklace into which i is initially inserted is not the empty necklace.

We observe at this junction that not only the reverse of the task of case C1, but the reverse of any of the tasks under A-D discussed above can be achieved by a finite number of primitive PCQ manipulations. Thus i can be extracted from $A(S \cup \{i\})$) to obtain $A(S)$, whenever the segments $l=PRED(i)$ and $r=SUCC(i)$ do not overlap. We can now deal with the special case where l and r overlap. By Lemma 1, r must be the tail of a chunk of $\bar{S}$, and $(i-l)$ is the minimum period of w. If r is not in a necklace of S, then neither is i. In this case the insertion of i in the INDEX is all is needed to update $A(S)$. If r is a necklace segment, then, by Fact 4, $PRED(r)$ and $SUCC(r)$ in S cannot overlap, neither can i and $SUCC(r)$ in S'. Our procedure extracts r from $A(S)$, and then inserts i and r in this abacus, in succession.

This concludes the proof of Theorem 1. □

5. THE PROCEDURE "SHRINK"

Let $\bar{S}$ be the maxsect of w-segments of x, and let u be the contraction of w. We will now use S to denote the set that is obtained by contracting all the segments in $\bar{S}$ to the new length $|u|$. Note that S is not necessarily a maxsect. However, the reader is urged to check that S is always a sect. We want to transform $A(\bar{S})$ into $A(S)$, i.e., we want to examine the effect on the structure of $A(\bar{S})$ caused by the contraction of all segments of $\bar{S}$ to achieve their new length $|u|$. By Fact 1, u is not shorter than the longest prefix of w

for which $A(S)$ and $A(\bar{S})$ may differ.

We examine, informally at first, the left-to-right sequence of contractions of the segments of $\bar{S}$.

Let then i be the generic such segment and assume that the necklace structure to the left of i has reached its final status. Assume first that i is not in a chunk. Then i is certainly in a (possibly singleton) necklace η of $\bar{S}$. It is clear that the contraction of i has no effect on the structure of η if i has no detector in η, or if i has a detector j such that $(j-i)<|u|$. If, on the other hand, i has a detector j in η such that $(j-i)=|u|$, then the contraction of i splits η in two nonempty necklaces η_1 and η_2, i becoming the last segment in η_1 and j the first segment of η_2. In addition, if j was a slave in η, then all slave segments in $\eta\cap\eta_2$ become master and *vice versa* (See Fig. 3).

Assume now that i is a *master* or *slave* segment in a chunk. It is easy to see that the contraction of i cannot split the necklace η in this case. However, such contraction may play havoc with the previous master-slave organization of the chunk, by freeing a chunk segment formerly not in the necklace η. This perturbation ripples through the segments of the chunk which fall to the right of i and might affect the status of the first segment of η which was not in the chunk. In addition, if $u=v^2$ with v primitive, then the contraction of all the segments which precede i in the chunk has destroyed a prefix of the chunk. All those segments are now in the necklace which contained the chunkhead prior to the contraction. With its contraction, also segment i detaches itself from the chunk.

Finally, it is easily seen that the contraction of a segment which is in a chunk but not in a necklace does not affect the structure of the necklace.

Let (m,k) be the span of η and let C' and C'' be, respectively, the sizes of the compact $u-$taggings of $x(1,k)$ before and after the contraction of i. We summarize our discussion in the following fact from [AR].

FACT 10:

$C''=C'+1$ if and only if j is a slave in η and η_2 is an odd necklace.

The above discussion suggests that, in order to transform $A(\bar{S})$ into $A(S)$ we do not necessarily have to reconsider all the segments of $\bar{S}$: in fact we will achieve our objective if we show how, under each of the various cases above, we can scan the sequence of segments whose contraction infringes the consistency of the abacus and restore such consistency.

In the abacus transformation which we propose, a crucial role is played by the ROSARY section of the abacus, as it was to be expected. Indeed, by the definition of an abacus, either the segments that have to be reconsidered are reachable through the chunkweave of $A(\bar{S})$, or such segments can be scanned through the p-weave, where $p = |u|$. We now describe the procedure *shrink*. We assume a *legal* call to *shrink*, i.e, a call *shrink* $(A(\bar{S}), |u|)$ such that $\bar{S}$ is a maxsect and u is the legal contraction of w.

```
procedure shrink(A(S̄), |u|)
begin q = |u|;
Case A: no integer fraction of q is the period of the chunkweave
        begin scan the q-weave;
              for each segment i do restore(i);
        end;
Case B: q/k, with k>2 is the period of the chunkweave
        begin scan the chunkweave;
              for each chunkhead i do reset(i);
        end;
Case C: q/2 is the period of the chunkweave
        begin scan the chunkweave;
              for each chunkhead i do dissolve(i);
        end;
end.
```

We shall use the structure of *shrink* along with the description of the auxiliary procedures *dissolve*, *reset* and *restore*, in order to establish the following:

Theorem 2: If u is the legal contraction of w, then *shrink* correctly transforms $A(\bar{S})$ into $A(S)$.

Proof. The assertion is a consequence of Lemmas 2-4 below. □

Assume that the conditions of Case A are met. Let i be the generic segment in the q*-weave*, and let A_i be the *hybrid* abacus such that the portion of A_i relative to all segments which precede i has been correctly updated to reflect the contraction of such segments, whereas the portion of A_i relative to all other segments (inclusive of i) coincides with the corresponding portion of $A(\tilde{S})$. Consider the following:

```
procedure restore (i)
begin
      DELETE [q−WEAVE ,i];
      $\eta \leftarrow FIND[i]$;
      $\eta_1, \eta_2 \leftarrow SPLIT[\eta, i]$;
      (**this produces two necklaces, $\eta_1$ containing $i$, $\eta_2$ not empty**)
      Use Fact 7 to update C;
end.
```

Lemma 2: With $j = SUCC[q-WEAVE, i]$ and under the conditions of Case A, *restore*(i) correctly transforms A_i into A_j.

Proof. It is easy to check that the procedure performs all and only the manipulations needed to reflect the contraction of segment i. (Note, in particular, that the rank of j does not change even if j is a chunkhead.) Moreover, u being the legal contraction of w implies that the difference between the starting positions of any two consecutive segments of $\tilde{S}$ cannot be larger than $|u| = q$. Thus *restore*(i) has produced a hybrid abacus which is identical to A_j. □

We now turn to Case B. It is an immediate consequence of Fact 3 that, under the present conditions, whenever the contraction of segment i removes a former overlap of i with another segment j, then j is in the same chunk as i. Let then A_i and A_j be defined as above, except that now i and j are chunkheads. The effect which we purport for *reset*(i) is that of accomplishing at once the contraction of all the segments in the chunk headed by i. Fact 5 will be used to achieve this with a constant number of primitive operations on PCQs. To be more precise, let η be the necklace within which i falls. One way of specifying the operation of *reset*(i) is as follows. Assume first that the tail r of the chunk is not in η. Then the last segment of η is a segment in the chunk headed by i. In order to account for the contraction of all segments in the chunk it is sufficient to recompute the rank of i using Fact 5. Assume now that r is in η. Then, by Fact 4, the removal of r from η splits η into two necklaces, η_1 and η_2. Indeed, if η_2 is not empty, then the last segment of η_1 (both before and after its contraction) does not overlap with the first segment of η_2. To deal with the most general case, assume η_2 not empty. Segment r can be extracted from the abacus along the lines of the discussion of *update*. Once r is extracted from A_i, the rank of i can be updated through Fact 5 as above, with the only proviso of not counting r into the chunk. Since $S-\{r\}$ is clearly a sect, and r is a compatible segment, the reinsertion of the contracted segment r can be accomplished through resort to the procedure *update*. Hence:

Lemma 3: With $j=SUCC[CHUNKWEAVE,i]$ and under the conditions of Case B, *reset*(i) correctly produces A_j from A_i.

Proof. An immediate consequence of the operation of *reset* and of an argument similar to that used in the derivation of Lemma 2. □

Finally, we turn to Case C. The iterated action of the procedure *dissolve* has the effect of of downgrading a formerly chunkweave to the rank of an ordinary p-weave. Indeed, each call to *dissolve* results in the destruction of the structure of a chunk. The segments formerly in the chunk are now suitably aggregated in a necklace. The operation of *dissolve* is similar to that of *reset*, except that now all the segments formerly in the chunk have to be individually inserted, through *update*, in the necklace which used to contain only the chunkhead. We leave the details of such manipulations to the reader. It is also easy to prove the last one of the lemmas supporting Theorem 1.

Lemma 4: With $j=SUCC[CHUNKWEAVE,i]$ and under the conditions of Case C, *dissolve* correctly transforms A_i into A_j.

Notice that *dissolve*(i) is the only auxiliary procedure in *shrink* which does not require a constant number of primitive PCQ operations.

6. THE TOP LEVEL AND ITS ANALYSIS

We are now in the position to discuss the top level procedure *weight* that, starting from the leaves of T_x: produces the abacuses associated with each internal node by 'merging' those associated with the offsprings, inserts candidate auxiliary nodes and tests their necessity. Thus *weight* will produce the MAST of x from T_x.

Assume (without loss of generality [AP]) that T_x is a a binary tree and denote by $\bar{T}_x$ the partially updated structure that is obtained from T_x by the time our algorithm handles node α. Node α can be a node originally in T_x or an auxiliary node recently inserted. Let $S(\alpha)$, $S_L(\alpha)$ and $S_R(\alpha)$ (with one of the two latter possibly empty) be the sets of segments of length $|W(\alpha)|$ and pertaining, respectively, to the subtrees of $\bar{T}_x$ rooted at α, LSON(α), and RSON(α) (again, LSON or RSON may be empty). $S(\alpha)$ is clearly a maxsect.

Lemma 5: $S_R(\alpha)$ and $S_L(\alpha)$ are sects.

Proof. The assertion is true if $S(\alpha)$ contains no chunks. Let now $\{i_1,i_2,...,i_k\}$ be a chunk of segments of $S(\alpha)$. Then $W(\alpha)$ has a (unique) period p such that $p \leq |W(\alpha)|/2$, and by definition of a chunk, there is a maximal repetition $R(i_1,p=(i_2-i_1),L)$ in x which has the form $(s\pi)^t s$ and is such that L is equal to the span of the chunk. By Fact 2, and by the definition of maximal repetition, there must be a vertex ν in T_x such that $W(\nu)=(s\pi)^{t-1}s \geq |W(\alpha)|$ and such that i and $i+p$ are consecutive leaves in the subtree of T_x rooted at ν, but i and $i+p$ are not simultaneously in the set of leaves pertaining to LSON(ν) or RSON(ν). If ν coincides with α, then it must be $k=2$, and the assertion holds for this case. Assume now that ν is a descendant of α, and let i_m $(1<m<k)$, be the rightmost segment in, say, $\mu = S_L(\alpha)$ such that i_{m-1} is not in $S_L(\alpha)$. Since there are occurrences of $W(\mu)$ in x with starting positions $i_m, i_{m+1}, \cdots i_k$, and there is an occurrence of $W(\alpha)$ with starting position $i_{m-1}=i_m-p$, with $|W(\alpha)| > p$, then, by the definition of period, there is also an occurrence of $W(\mu)$ starting at position $m-1$, a contradiction. □

Consider the following procedure.

```
procedure weigth $(\alpha)$
begin if $\alpha$ is a leaf then initialize$(A(\alpha))$;
    else begin $A_L(\alpha) \leftarrow$ weight$(LSON(\alpha))$;
        $A_R(\alpha) \leftarrow$ weight$(RSON(\alpha))$;
        if $|S_L(\alpha)| < |S_R(\alpha)|$ then swap the roles of LSON and RSON of $\alpha$ ;
        (**Thus now $|S_R(\alpha)| \leq |S_L(\alpha)|$

        (MERGE) - if both $S_R(\alpha)$ and $S_L(\alpha)$ are nonempty
            then (**construct $A(\alpha)$ **)
            Apply update in succession to the segments of $S_R(\alpha)$ and to the
            successively updated versions of $A_L(\alpha)$;
            (**The value of $C(W(\alpha))$ is a trivial byproduct of this operation**).

        (REMOVE) - else (** $\alpha$ has only one son**)
            if $C(W(\alpha)) = C(W(\mathrm{SON}(\alpha)))$ then remove $\alpha$;
            else assign to $\alpha$ the weight $C(W(\alpha))$.

        (CLIMB) - Let $kq$ be the maximum length of a prefix $u$ of $W(\alpha)$ which can be
            obtained with $q$ in the rosary index and $k$ a nonzero integer;
            Let $\nu$=FATHER$(\alpha)$;
            If $|u| > |W(\nu)|$ , then
                Use shrink to transform $A(\alpha)$ into $A_L(\nu)$ or $A_R(\nu)$,
                depending on whether $\alpha$ = LSON($\nu$) or $\alpha$ = RSON($\nu$).
            else begin
                Create a node $\mu$ such that $W(\mu)=w$ ;
                (**this is the mechanism that inserts auxiliary nodes**).
                Use shrink to transform $A(\alpha)$ into $A(\nu)$;
                weight ( $\nu$ );
            end;
        end;
end.
```

Theorem 3: With ρ denoting the root of T_x, the execution of *weight* (ρ) correctly transforms T_x into the MAST of x.

Proof. The correctness of *REMOVE* is trivial. The correctness of *MERGE* follows from Theorem 1 and Lemma 5, and from the simple observations that the set of segments resulting from the union of S_L with the first $k-1$ ($k \geq 1$) segments of S_R is a sect S, and the $k-th$ segment of S_R is S-compatible. The correctness of *CLIMB* follows from Theorem 2 in view of the following two observations. First, for any node α in the current version $\bar{T}_x$ of T_x, $S(\alpha)$ is maxsect. Second, the node which *weight* (α) sets to be the father of α in $\bar{T}_x$ is precisely the proper locus of the legal contraction of $W(\alpha)$, as is easy to check. Thus each call to *shrink* complies with our definition of a legal call. □

We now analyze the performance of *weight*. We have the following:

Theorem 4: The procedure WEIGHT produces the MAST $\bar{T}_x$ associated with any string x in $O(n \log^2 n)$ time and $O(n \log n)$ space.

Proof. At any given time, the space taken by all working abacuses is $O(n)$, since such space is proportional to the sum of the cardinalities of the disjoint sets being merged. Thus the dominant factor is the space occupied by T_x, the current version of $\hat{T}_x$. But this cannot exceed the space required by $\bar{T}_x$, which is bounded by $O(n \log n)$, in force of a result in [AR]. Thus this bound applies to the space needed by the procedure.

The discussions of the procedures *update* and *shrink* show that each call to one such procedure that does not involve the procedure *dissolve* requires a finite number of constant time manipulations and primitive operations on PCQs. Thus each such call takes $O(\log n)$ at most to be executed. Since each time we merge a leaf i from a set $S^{(1)}$ into $S^{(2)}$ it is $|S^{(1)}| \leq |S^{(2)}|$, the set $S = S^{(1)} \cup S^{(2)}$ has cardinality at least double than that of $S^{(1)}$, then the same leaf cannot be involved in an *update* for more than $\log n$ times. Hence the total work charged by the calls to *update* is by $O(n \log^2 n)$.

It is easy to see that the total number of calls to *restore* is $O(n \log n)$. Indeed, each such call *restore*(i) can be charged to a distinct positioned square substring uu of x, namely, the square of starting position i and period equal to the difference $j-i$, where j is the (starting position) of the current detector of i. Since the number of distinct positioned squares (or, equivalently, repetitions) in a string of length n is bounded by $O(n \log n)$ [CR], then so is the total number of such calls. Thus also the calls to *restore* charge a total $O(n \log^2 n)$ time. A similar argument shows that this bound also applies to the overall executions of *dissolve*. Indeed, each step (i.e., the contraction of each segment) performed within the loop of *dissolve* can be charged to a positioned square, namely, the square which is spanned exactly by the contracted segment, and each square can be charged exactly once.

Finally, Fact 2 and Lemma 5 show that, between any two consecutive calls to *reset* bearing the same parameter i, a new repetition of x is detected. Thus also the total number of executions of *reset* is $O(n \log n)$. By exploiting the structure of abacuses, it is easy to accomplish the manipulations involved at the inception of *CLIMB*, inclusive of the creation of a candidate auxiliary node where appropriate, in constant time. We leave it as an exercise for the reader to show that the upper bound on the number of distinct repetitions in x yields an overall $O(n \log n)$ time bound for these manipulations as well. □

REFERENCES

[AA] A. Apostolico, *The Myriad Virtues of Subword Trees*, Combinatorial Algorithms on Words (A. Apostolico and Z. Galil, eds.), Springer-Verlag ASI F-12, (1985), 85-95.

[AH] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addision Wesley, Ma. (1974).

[AP] A. Apostolico and F. P. Preparata, *Optimal Off-line Detection of Repetitions in a String*, Theoretical Comp. Sci., 22, 297-315 (1985).

[AO] A. Apostolico. *On Context Constrained Squares and Repetitions in a String*, The RAIRO Journal on TCS 18, 2, 147-159 (1984).

[AR] A. Apostolico and F.P. Preparata. *Structural Properties of the String Statistics Problem*, Tech. Rep., Purdue Univ. CS Dept. (1985). To appear in: Journal of Computer and Systems Science.

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

[GA] Z. Galil, *On Improving the Worst Case Running Time of the Boyer Moore String Matching Algorithm*, Proceedings of the ICALP (G. Ausiello and C. Bohm eds.), Springer-Verlag Lecture Notes in Computer Sciences 62, (1978), 241-250.

[LS] R. C. Lyndon and M. P. Schutzemberger, *The Equation* $a^M = b_N c^P$ *in a Free Group*, Mich. Math. Journal 9, (1962), 289-298.

[MC] E. M. McCreight, *A Space Economical Suffix Tree Construction Algorithm*, Jour. of the ACM 25 (1976), 262-272.

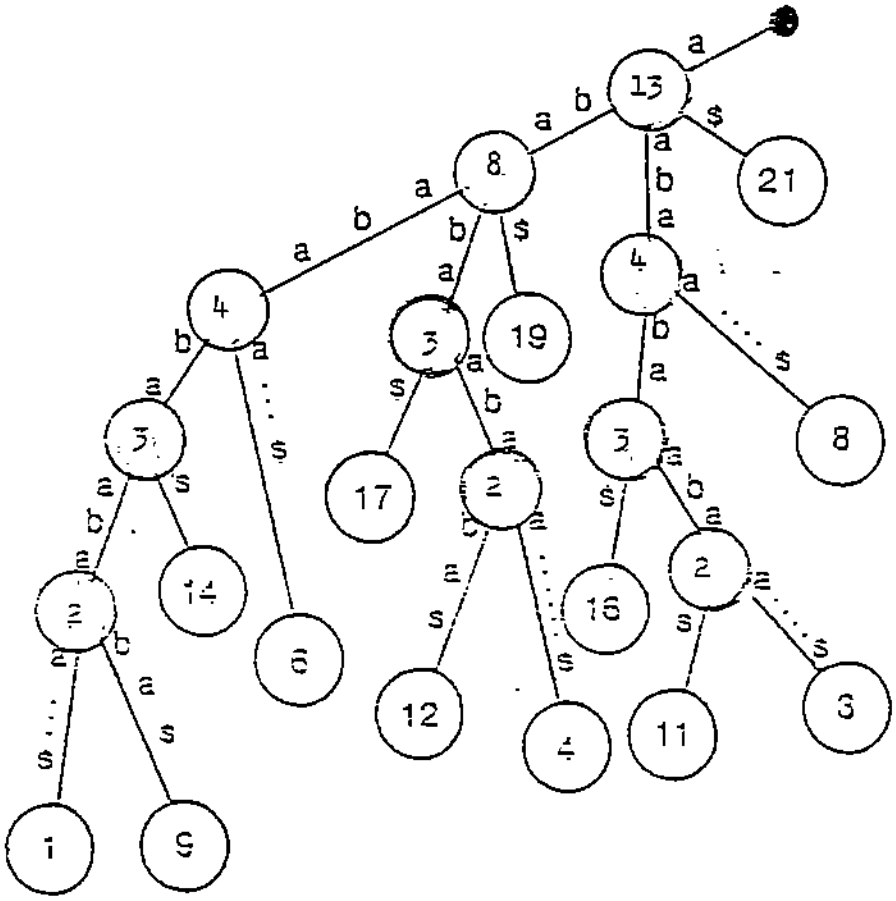


Figure 1.a

A partial view (all suffixes starting with a) of the weighted suffix tree of the string x=*abaababaabaababaababa*. The weight of each internal node reports the statistics with possible overlaps of any substring of x of which that node is the locus. $\$$ is a symbol which never occurs in x. Each leaf is labeled with the starting position of the suffix of $x\$$ which is described by the concatenation of the labels on the path from the root to that leaf. The symbol $\$$ appended at the end of each suffix ensures that the paths relative to any two suffixes eventually diverge in the tree (in fact, no suffix of $x\$$ is the prefix of another suffix of $x\$$). In practice, each arc is labeled with a pair of pointers to the two positions of x which delimit the substring of x associated with that arc (whence the linear space bound).

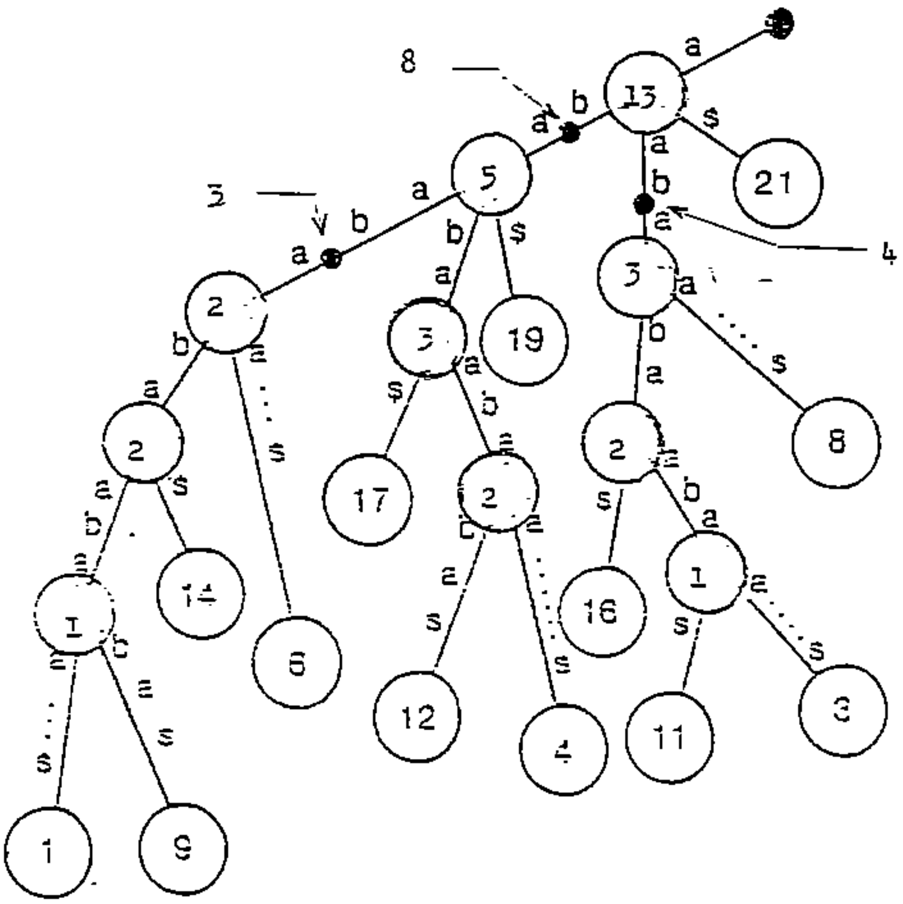
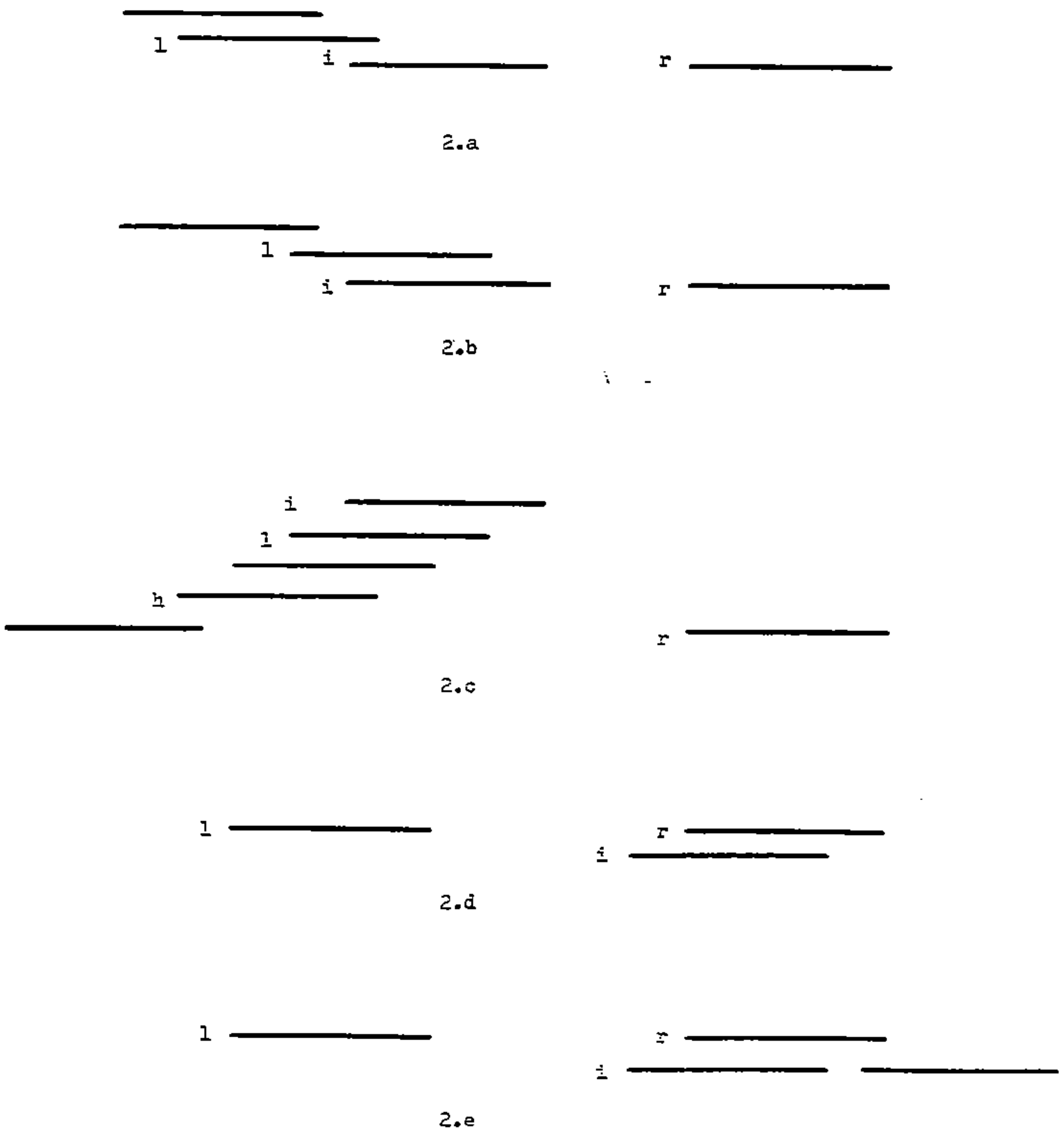


Figure 1.b

A modified version of the (partial) index of Fig. 1.a which reports the statistics without overlaps of all substrings of x. Notice that not only the weights of most internal nodes are changed, but new nodes have been inserted to account for changes in the nonoverlapping statistics occurring in the middle of an arc.



2.a

2.b

2.c

2.d

2.e

Figure 2

Some of the possibilities which may arise when *update* has to handle segment *i*.

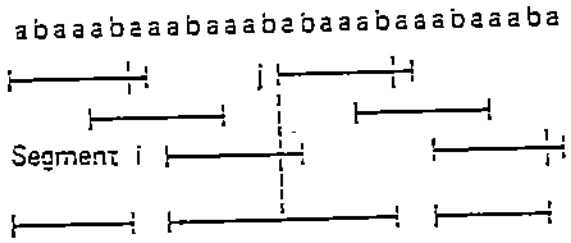


Figure 3

The effect of the contraction of segment i on a previously slave segment j. The necklace which originally contained both segments splits in two necklaces. The statistics without overalps undergoes an unit increment since the conditions of Fact 10 are met.