1996

Annotated Statistical Indices for Sequence Analysis

Alberto Apostolico

Mary Ellen Bock

Xuyan Xu

Report Number:
96-072
ANNOTATED STATISTICAL INDICES
FOR SEQUENCE ANALYSIS

Alberto Apostolico
Mary Ellen Bock
Xuyan Xu

CSD-TR 96-072
November 1996
Annotated Statistical Indices for Sequence Analysis

Alberto Apostolico*   Mary Ellen Bock†   Xuyan Xu‡

November 20, 1996

Abstract

A statistical index for string \( x \) is a digital-search tree or trie that returns, for any query string \( w \) and in a number of comparisons bounded by the length of \( w \), the number of occurrences of \( w \) in \( x \). Clever algorithms are available that support the construction and weighting of such indices in time and space linear in the length of \( x \). This paper addresses the problem of annotating a statistical index with such parameters as the expected value and variance of the number of occurrence of each substring.

Key Words and Phrases: Design and analysis of algorithms, combinatorics on strings, pattern matching, substring statistics, suffix tree, annotated suffix tree, period of a string, repetition in a string.

AMS subject classification: 68C25

*Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, IN 47907, USA and Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy. axa@cs.purdue.edu. Work supported in part by NSF Grant CCR-9201078, by NATO Grant CRG 900293, by the National Research Council of Italy, and by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM).
†Department of Statistics, Purdue University, Math. Sciences Building, West Lafayette, IN 47907, USA. mbock@stat.purdue.edu.
‡Department of Statistics, Purdue University, Math. Sciences Building, West Lafayette, IN 47907, USA. xuyan@stat.purdue.edu.
1 Introduction

Searching for repeated substrings, periodicities, symmetries, cadences, and other similar regularities or unusual patterns in objects is an increasingly recurrent task not only in the analysis of genomic sequences but also in countless other activities, ranging from data compression to symbolic dynamics and the monitoring and detection of unusual events. In most of these endeavors, substrings are sought that are, by some measure, typical or anomalous in the context of larger sequences. Some of the most conspicuous and widely used measures of typicality for a substring hinge on the frequency of its occurrences: a substring that is either too frequent or too rare in terms of some suitable parameter of expectation is immediately suspected to be anomalous in its context.

Tables for storing the number of occurrences in a string of substrings of (or up to) a given length are routinely computed in applications. Actually, clever methods are available to compute and organize the counts of occurrences of all substrings of a given string. The corresponding tables take up the tree-like structure of a special kind of digital search index or trie (see, e.g., [Mc-76], [Ap-85], [AP-96]). These trees have found use in numerous applications [Ap-85], including of course computational biology [Wa-95]. Once the index itself is built, it makes sense to annotate its entries with the expected values and variances that may be associated with them under one or more probabilistic models. One such process of annotation is addressed in this paper.

The paper is organized as follows. In the next section, we review some basic facts pertaining to the construction and structure of statistical indices. We then summarize in Section 3 some needed combinatorics on words. Section 4 is devoted to the derivation of formulae for expected values and variances for substring occurrences, in the hypothesis of a generative process governed by independent, identically distributed random variables. Our formulae will be written in a form that is conducive to efficient computation, within the paradigm discussed in Section 2. The computation itself will be the object of Section 5, which concludes our presentation.

2 Preliminaries

Given an alphabet \( \Sigma \), we use \( \Sigma^+ \) to denote the free semigroup generated by \( \Sigma \), and set \( \Sigma^* = \Sigma^+ \cup \{ \lambda \} \), where \( \lambda \) is the empty word. An element of \( \Sigma^+ \) is called a string or sequence
or word, and is denoted by one of the letters \( s, u, v, w, x, y \) and \( z \). The same letters, upper case, are used to denote random strings. We write \( x = x_1 x_2 \ldots x_n \) when giving the symbols of \( x \) explicitly. The number of symbols that form \( w \) is called the length of \( w \) and denoted by \( |w| \). If \( x = wu \), then \( w \) is a substring of \( x \) and the integer \( 1 + |v| \) is its (starting) position in \( x \). Let \( I = [i, j] \) be an interval of positions of a string \( x \). We say that a substring \( w \) of \( x \) begins in \( I \) if \( I \) contains the starting position of \( w \), and that it ends in \( I \) if \( I \) contains the position of the last symbol of \( w \).

Clever pattern matching techniques and tools (see, e.g., [Ah-90, AHU-74, AG-85, CR-94]) have been developed in recent years to count (and locate) all distinct occurrences of an assigned substring \( w \) (the pattern) within a longer string \( x \) (the text). As is well known, this problem can be solved in \( O(|x|) \) time, regardless of whether instances of the same pattern \( w \) that overlap - i.e., share positions in \( x \) - have to be distinctly detected, or else the search is limited to one of the streams of consecutive nonoverlapping occurrences of \( w \).

When frequent queries of this kind are in order on a fixed text, each query involving a different pattern, it might be convenient to preprocess \( x \) to construct an auxiliary index tree [AHU-74, Ap-85, We-73, Mc-76, CS-85] storing in \( O(|x|) \) space information about the structure of \( x \). This auxiliary tree is to be exploited during the searches as the state transition diagram of a finite automation, whose input is the pattern being sought, and requires only time linear in the length of the pattern to know whether or not the latter is a substring of \( x \). Here, we shall adopt the version known as suffix tree, introduced in [Mc-76]. Given a string \( x \) of length \( n \) on the alphabet \( \Sigma \), and a symbol \$ not in \( \Sigma \), the suffix tree \( T_x \) associated with \( x \) is the digital search tree that collects the first \( n \) suffixes of \( x \$ \). In the expanded representation of \( T_x \), each arc is labeled with a symbol of \( \Sigma \), except for terminal arcs, that are labeled with a substring of \( x \$ \). The space needed can be \( \Theta(n^2) \) in the worst case [AHU-74]. An example of expanded suffix tree is given in Figure 1.

In the compact representation of \( T_x \) (see Figure 2), chains of unary nodes are collapsed into single arcs, and every arc of \( T_x \) is labeled with a substring of \( x \$ \). A pair of pointers to a common copy of \( x \) can be used for each arc label, whence the overall space taken by this version of \( T_x \) is \( O(n) \). In both representations, suffix \( sufi \) of \( x \$ (i = 1, 2, \ldots, n) \) is described by the concatenation of the labels on the unique path of \( T_x \) that leads from the root to leaf \( i \). Similarly, any vertex \( \alpha \) of \( T_x \) distinct from the root describes a subword \( w(\alpha) \) of \( x \) in a natural way: vertex \( \alpha \) is called the proper locus of \( w(\alpha) \). In the compact \( T_x \),
the \textit{locus} of \( w \) 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 \( w \).

An algorithm for the construction of the expanded \( T_x \) is readily organized as in Figure 3. We start with an empty tree and add to it the suffixes of \( x \$ \) one at a time. Conceptually, the insertion of suffix \( \text{suffix}_i \) \((i = 1, 2, ..., n + 1)\) consists of two phases. In the first phase, we search for \( \text{suffix}_i \) in \( T_{i-1} \). Note that the presence of \( \$ \) guarantees that every suffix will end in a distinct leaf. Therefore, this search will end with failure sooner or later. At that point, though, we will have identified the longest prefix of \( \text{suffix}_i \) that has a locus in \( T_{i-1} \). Let \( \text{head}_i \) be this prefix and \( \alpha \) the locus of \( \text{head}_i \). We can write \( \text{suffix}_i = \text{head}_i \cdot \text{tail}_i \) with \( \text{tail}_i \) nonempty. In the second phase, we need to add to \( T_{i-1} \) a path leaving node \( \alpha \) and labeled \( \text{tail}_i \). This achieves the transformation of \( T_{i-1} \) into \( T_i \).

We can assume that the first phase of \textit{insert} is performed by a procedure \textbf{findhead}, which takes \text{suffix}_i \) as input and returns a pointer to the node \( \alpha \). The second phase is performed then by some procedure \textbf{addpath}, that receives such a pointer and directs a path from node \( \alpha \) to leaf \( i \). The details of these procedures are left for an exercise. As is easy to check, the procedure \textbf{buildtree} takes time \( O(n^2) \) and linear space. It is possible to prove (see, e.g., [AS-92]) that the average length of \( \text{head}_i \) is \( O(\log i) \), whence building \( T_x \) by brute force requires \( O(n \log n) \) time on average. Clever constructions such as in [Mc-76] avoid the necessity of tracking down each suffix starting at the root.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{suffix_tree.png}
\caption{An expanded suffix tree}
\end{figure}
Irrespective of the type of construction used, some simple additional manipulations on the tree make it possible to count the number of distinct (possibly overlapping) instances of any pattern $w$ in $x$ in $O(|w|)$ steps. For this, observe that the problem of finding all occurrences of $w$ can be solved in time proportional to $|w|$ plus the total number of such occurrences: either visit the subtree of $T_x$ rooted at the locus of $w$, or preprocess $T_x$ once for all by attaching to each node the list of the leaves in the subtree rooted at that node. A trivial bottom-up computation on $T_x$ can then weight each node of $T_x$ with the number of leaves in the subtree rooted at that node. This weighted version serves then as a statistical index for $x$ [Ap-85, AP-96], in the sense that, for any $w$, we can find the

```
procedure buildtree ( x,T_x )
begin
  T_0 ← ∅;
  for i = 1 to n + 1 do T_i ← insert(suf_i,T_{i-1});
  T_x ← T_{n+1};
end
```

Figure 3: Building an expanded suffix tree
frequency of \( w \) in \( x \) in \( O(|w|) \) time. We note that this weighting cannot be embedded in the linear time construction of \( T_z \), while it is trivially embedded in the brute force construction: Attach a counter to each node; then, each time a node is traversed during \texttt{insert}, increment its counter by 1; if \texttt{insert} culminates in the creation of a new node \( \beta \) on the arc \((\text{Father}(\alpha), \alpha)\), initialize the counter of \( \beta \) to \( 1 + \text{counter of } \alpha \). A suffix tree with weighted nodes is presented in Figure 4 below. Note that the counter associated with the locus of a string reports its correct frequency even when the string terminates in the middle of an arc.

In conclusion, the full statistics (with possible overlaps) of the substrings of a given string \( x \) can be precomputed in one of these trees, within time and space linear in the textlength.

Figure 4: A partial suffix tree weighted with substring statistics
3 Periodicities in Strings

A string $z$ has a period $w$ if $z$ is a prefix of $w^k$ for some integer $k$. Alternatively, a string $w$ is a period of a string $z$ if $z = w^jv$ and $v$ is a possibly empty prefix of $w$. Often when this causes no confusion, we will use the word “period” also to refer to the length or size $|w|$ of a period $w$ of $z$. A string may have several periods. The shortest period (or period length) of a string $z$ is called the period of $z$. Clearly, a string is always a period of itself. This period is called the trivial period.

A germane notion is that of a border. We say that a non-empty string $w$ is a border of a string $z$ if $z$ starts and ends with an occurrence of $w$. That is, $z = uw$ and $z = uv$ for some possibly empty strings $u$ and $v$. Clearly, a string is always a border of itself. This border is called the trivial border.

**Fact 3.1** A string $x[1..k]$ has period of length $q$, such that $q < k$, if and only if it has a non-trivial border of length $k - q$.

**Proof:** Immediate from the definitions of a border and a period. □

A word $x$ is primitive if setting $x = s^k$ implies $k = 1$. A string is periodic if its period repeats at least twice. The following well known lemma shows, in particular, that a string can be periodic in at most one primitive period.

**Lemma 3.2 (Periodicity Lemma [LS-82])** If $w$ has periods of sizes $d$ and $q$ and $|w| \geq d + q$ then $w$ has period of size $gcd(d, q)$.

A word $x$ is strongly primitive or square-free if every substring of $x$ is a primitive word. A square is any string of the form $ss$ where $s$ is a primitive word. For example, $cabca$ and $cababd$ are primitive words, but $cabca$ is also strongly primitive, while $cababd$ is not, due to the square $abab$. Given a square $ss$, $s$ is the root of that square.

Let now $w$ be a substring of $x$ having at least two distinct occurrences in $x$. Then, there are words $u, y, u', y'$ such that $u \neq u'$, and $x = uyuy'$. Assuming w.l.o.g. $|u| < |u'|$, we say that those two occurrences of $w$ in $x$ are disjoint iff $|u'| > |uw|$, adjacent iff $|u'| = |uw|$ and overlapping if $|u'| < |uw|$. Then, it is not difficult to show (see, e.g., [Lo-83]) that word $x$ contains two overlapping occurrences of a word $w \neq \lambda$ iff $x$ contains a word of the form $awawa$ with $a \in \Sigma$ and $v$ a word.

One more important consequence of the Periodicity Lemma is that if $y$ is a periodic string, $u$ is its period, and $y$ has consecutive occurrences at positions $i_1, i_2, \ldots, i_k$ in $x$ with
\( i_j - i_{j-1} \leq |y|/2, \ (1 < j \leq k), \) then it is precisely \( i_j - i_{j-1} = |u| \ (1 < j \leq k). \) In other words, consecutive overlapping occurrences of a periodic string will be spaced apart exactly by the length of the period.

4 Computing Expectations

Let \( X = X_1X_2 \ldots X_n \) be a textstring randomly produced by a source that emits symbols from an alphabet \( \Sigma \) according to some known probability distribution, and let \( y = y_1y_2 \ldots y_m \) \((m < n)\) be an arbitrary but fixed pattern string on \( \Sigma \). We want to compute the expected number of occurrences of \( y \) in \( X \), and the corresponding variance. For \( i \in \{1, 2, \ldots, n - m + 1\} \), define \( Z_i \) to be 1 if \( y \) occurs in \( X \) starting at position \( i \) and 0 otherwise. Let

\[
Z = \sum_{i=1}^{n-m+1} Z_i,
\]

so that \( Z \) is the total number of occurrences of \( y \). For given \( y \), we assume random \( X_k \)'s in the sense that:

1. the \( X_k \)'s are independent of each other and
2. The \( X_k \)'s are identically distributed, so that, for each value of \( k \), the probability that \( X_k = y_i \) is \( p_i \).

Then

\[
E[Z|y] = \prod_{i=1}^{m} p_i = \hat{p}.
\]

Thus,

\[
E[Z|y] = (n - m + 1)\hat{p}
\]  \hspace{1cm} (1)

and

\[
Var(Z|y) = \sum_{i,j} Cov(Z_i, Z_j) = \sum_{i=1}^{n-m+1} Var(Z_i) + 2 \sum_{i < j \leq n-m+1} Cov(Z_i, Z_j)
\]

\[
= (n - m + 1)Var(Z_1) + 2 \sum_{i < j \leq n-m+1} Cov(Z_i, Z_j)
\]

7
Because $Z_i$ is an indicator function,

$$E[Z_i^2] = E[Z_i] = \hat{p}.$$ 

This also implies that

$$\text{Var}(Z_i) = \hat{p}(1 - \hat{p})$$

and

$$\text{Cov}(Z_i, Z_j) = E[Z_i Z_j] - E[Z_i] E[Z_j].$$

Thus

$$\sum_{i < j \leq n - m + 1} \text{Cov}(Z_i, Z_j) = \sum_{i < j \leq n - m + 1} (E[Z_i Z_j] - \hat{p}^2)$$

If $j - i \geq m$, then $E[Z_i Z_j] = E[Z_i] E[Z_j]$, so $\text{Cov}(Z_i, Z_j) = 0$. Thus, if $j - i < m$,

$$\sum_{i < j \leq n - m + 1} \text{Cov}(Z_i, Z_j) = \sum_{i=1}^{n-m} \sum_{j=i+1}^{\min(i+m-1, n-m+1)} \text{Cov}(Z_i, Z_j)$$

$$= \sum_{i=1}^{n-m} \sum_{d=1}^{\min(m-1, n-m+1-i)} \text{Cov}(Z_i, Z_{i+d})$$

$$= \sum_{d=1}^{\min(m-1, n-m)} \sum_{i=1}^{n-m+1-d} \text{Cov}(Z_i, Z_{i+d})$$

$$= \sum_{d=1}^{\min(m-1, n-m)} (n - m + 1 - d) \text{Cov}(Z_1, Z_{1+d}).$$

Before we compute $\text{Cov}(Z_1, Z_{1+d})$, recall that an integer $d \leq m$ is a period of $y = y_1 y_2 \ldots y_m$ if and only if $y_i = y_{i+d}$ for all $i$ in $\{1, 2, \ldots, m - d\}$. Now, let $\{d_1, d_2, \ldots, d_s\}$ be the periods of $y$ that satisfy the conditions:

$$1 \leq d_1 < d_2 < \ldots < d_s \leq \min(m - 1, n - m).$$

Then, for $d \in \{1, 2, \ldots, m - 1\}$, we have that the expected value

$$E[Z_1 Z_{1+d}]$$
may be nonzero only in correspondence with a value of \( d \) equal to a period of \( y \). Therefore, 
\[ E[Z_1 Z_{1+d}] = 0 \]
for all \( d \)'s less than \( m \) not in the set \( \{d_1, d_2, \ldots, d_s\} \), whereas in correspondence of the generic \( d_i \) in that set we have:

\[ E[Z_1 Z_{1+d_i}] = \hat{p}\Pi_{j=m-d_i+1}^{m}p_j. \]

Resuming our computation of the covariance, we get then:

\[
\sum_{i<j \leq n-m+1} \text{Cov}(Z_i, Z_j) = \sum_{d=1}^{\min(m-1,n-m)} (n-m+1-d)\text{Cov}(Z_1, Z_{1+d}) \\
= \sum_{d=1}^{\min(m-1,n-m)} (n-m+1-d) * (E[Z_1 Z_{1+d}] - \hat{p}^2) \\
= \sum_{l=1}^{s} (n - m + 1 - d_l)\hat{p}\Pi_{j=m-d_l+1}^{m}p_j - \sum_{d=1}^{\min(m-1,n-m)} (n-m+1-d)\hat{p}^2 \\
- \hat{p}^2(2(n - m + 1) - 1 - \min(m - 1, n - m)) * \min(m - 1, n - m)/2.
\]

Thus,

\[ \text{Var}(Z) = (n - m + 1)\hat{p}(1 - \hat{p}) - \hat{p}^2(2(n - m + 1) - 1 - \min(m - 1, n - m)) * \min(m - 1, n - m) \]

\[ + 2\hat{p}\sum_{l=1}^{s}(n - m + 1 - d_l)\Pi_{j=m-d_l+1}^{m}p_j, \]

which depends on the values of \( m - 1 \) and \( n - m \). We distinguish the following cases.

Case 1: \( m \leq (n+1)/2 \)

\[ \text{Var}(Z) = (n - m + 1)\hat{p}(1 - \hat{p}) - \hat{p}^2(2n - 3m + 2)(n - 1) \]

\[ + 2\hat{p}\sum_{l=1}^{s}(n - m + 1 - d_l)\Pi_{j=m-d_l+1}^{m}p_j \]

(2)
5 Index annotation

As stated in the introduction, our goal is to augment a statistical index such as \( T_x \) so that its generic node \( \alpha \) shall not only reflect the count of occurrences of the corresponding substring \( y(\alpha) \) of \( x \), but also display the expected values and variances that apply to \( y(\alpha) \) under our probabilistic assumptions. Clearly, this can be achieved by performing the appropriate computations starting from scratch for each string. Even neglecting for a moment the computations needed to expose the underlying period structures, however, this would cost \( O(|y|) \) time for each substring \( y \) of \( x \) and thus result in overall time \( O(n^3) \) for a string \( x \) of \( n \) symbols. Fortunately, expressions 1, 2 and 3 can be embedded in the “brute-force” construction of Section 2 (cf. Figure 3) in a way that yields an \( O(n^2) \) overall time bound for the annotation process. We note that as long as we insist on having our values on each one of the substrings of \( x \), then such a performance is optimal as \( x \) may have as many as \( \Theta(n^2) \) distinct substrings. (However, a corollary of probabilistic constructions such as in [AS-92] shows that if attention is restricted to substrings that occur at least twice in \( x \) then the expected number of such strings is only \( O(n \log n) \).

Our claimed performance rests on the ability to compute the values associated with all prefixes of a string in overall linear time. These values will be produced in succession, each from the preceding one (e.g., as part of insert) and at an average cost of constant time per update. Observe that this is trivially achieved for the expected values in the form \( E[Z|y] \). In fact, even more can be stated: if we computed once and for all on \( x \) the \( n \) consecutive prefix products of the form

\[
\hat{p}_f = \Pi_{i=1}^{f} p_i \quad (i = 1, 2, ..., f),
\]

then this would be enough to produce later the homologous product as well as the expected value \( E[Z|y] \) itself for any substring \( y \) of \( x \), in constant time. To see this, consider the product \( \hat{p}_f \), associated with a prefix of \( x \) that has \( y \) as a suffix, and divide \( \hat{p}_f \) by \( \hat{p}_{f-|y|} \).
This yields the probability \( \hat{p} \) for \( y \) that appears in 1. Multiplying this value by \( (n - |y| + 1) \) gives then \( (n - m + 1)\hat{p} = E[Z|y)] \). From now on, we assume that the above prefix products have been computed for \( x \) in overall linear time and are available in some suitable array.

The situation is more complicated with the variance. However, expressions 2 and 3 still provide a handle for fast incremental updates of the type that was just discussed. Observe that each expression consists of three terms. In view of our discussion of prefix products, we can conclude immediately that the \( \hat{p} \)-values appearing in the first two terms of either 2 or 3 take constant time to compute. Hence, those terms are evaluated in constant time themselves, and we only need to concern ourselves with the third term, which happens to be the same in both expressions. In conclusion, we can concentrate henceforth on the evaluation of the sum:

\[
B = \sum_{i=1}^{s} (n - m + 1 - d_i) \prod_{j=m-d_i+1}^{m} p_j.
\]

Note that the computation of \( B \) depends on the structure of all \( d_i \) periods of \( y \) that are less than or equal to \( \min(m - 1, n - m) \). What seems worse, Expression \( B \) involves a summation on this set of periods, and the cardinality of this set is in general not bounded by a constant. Still, we can show that the value of \( B \) can be updated efficiently following a unit-symbol extensions of the string itself. We will not be able in general to carry out every such update in constant time. However, we will manage to carry out all the updates relative to the set of prefixes of a same string in overall linear time, thus in amortized constant time per update. This possibility rests on a simple adaptation of a classical implement of fast string searching, that computes the longest borders (and corresponding periods) of all prefixes of a string in overall linear time and space. We report such a construction in Figure 5 below, for the convenience of the reader, but refer for details and proofs of linearity to discussions of “failure functions” and related constructs such as found in, e.g., [AHU-74, Ah-90, CR-94].

To adapt Procedure maxborder to our needs, it suffices to show that the computation of \( B(m) \), i.e., the value of \( B \) relative to prefix \( y_1 y_2 ... y_m \) of \( y \), follows immediately from knowledge of \( \text{bord}(m) \) and of the values \( B(1), B(2), ... B(m - 1) \), which can be assumed to have been already computed. Noting that a same period \( d_i \) may last for several prefixes of a string, it is convenient to define the border associated with \( d_i \) at position \( m \) to be

\[
b_{i,m} = m - d_i.
\]
procedure maxborder ( y )
begin
  bord[1] ← 0;
  for m = 2 to h do
    r ← bord[m - 1];
    while r > 0 and yr+1 ≠ ym do
      r ← bord[r];
    endwhile
    if yr+1 ≠ ym and r = 0 then bord[m] ← r + 1;
  endfor
end

Figure 5: Computing the longest borders for all prefixes of y

Note that 
\[ d_t \leq \min(m - 1, n - m) \]
implies that
\[ b_t, m = m - d_t \geq m - \min(m - 1, n - m) \]
\[ = \max(m - m + 1, m - n + m) = \max(1, 2m - n). \]
However, this correction is not serious unless \( m > (n + 1)/2 \) as in case 2. We will assume we are in Case 1, where \( m \leq (n + 1)/2 \).

Let \( S(m) = \{ b_{t, m} \}_{t=1}^m \) be the set of borders at \( m \) associated with the periods of \( y_1 y_2 ... y_m \). The crucial fact subtending the correctness of our algorithm rests on the following simple observation.

\[ S(m) \equiv \{ \text{bord}(m) \} \cup S(\text{bord}(m)). \]

Going back to Expression B, we can write now using \( b_{t, m} = m - d_t \):
\[ B(m) = \sum_{i=1}^{m} (n - 2m + 1 + b_{i, m}) \prod_{j=b_{i, m}+1}^{m} p_j. \]
Separating from the rest the term relative to the largest border, this becomes:
\[ B(m) = (n - 2m + 1 + \text{bord}(m)) \prod_{j=\text{bord}(m)+1}^{m} p_j \]
Using Relation 4 and the definition of a border to re-write indices, we get:

\[ B(m) = (n - 2m + 1 + \text{bord}(m)) \prod_{j=\text{bord}(m)+1}^{m} p_j \]

\[ + \sum_{l=1}^{s_{\text{bord}(m)}} (n - 2m + 1 + b_i, \text{bord}(m)) \prod_{j=b_i, \text{bord}(m)+1}^{m} p_j \]

which becomes, adding the substitution \( m = m - \text{bord}(m) + \text{bord}(m) \) in the sum,

\[ B(m) = (n - 2m + 1 + \text{bord}(m)) \prod_{j=\text{bord}(m)+1}^{m} p_j \]

\[ + 2(\text{bord}(m) - m) \sum_{l=1}^{s_{\text{bord}(m)}} \prod_{j=b_i, \text{bord}(m)+1}^{m} p_j \]

\[ + \sum_{l=1}^{s_{\text{bord}(m)}} (n - 2 \cdot \text{bord}(m) + 1 + b_i, \text{bord}(m)) \prod_{j=b_i, \text{bord}(m)+1}^{m} p_j \]

\[ = (n - 2m + 1 + \text{bord}(m)) \prod_{j=\text{bord}(m)+1}^{m} p_j \]

\[ + 2(\text{bord}(m) - m) \sum_{l=1}^{s_{\text{bord}(m)}} \prod_{j=b_i, \text{bord}(m)+1}^{m} p_j \]

\[ + (\prod_{j=\text{bord}(m)+1}^{m} p_j) \cdot \sum_{l=1}^{s_{\text{bord}(m)}} (n - 2 \cdot \text{bord}(m) + 1 + b_i, \text{bord}(m)) \prod_{j=b_i, \text{bord}(m)+1}^{m} p_j \]

\[ = (n - 2m + 1 + \text{bord}(m)) \prod_{j=\text{bord}(m)+1}^{m} p_j \]

\[ + 2(\text{bord}(m) - m) \sum_{l=1}^{s_{\text{bord}(m)}} \prod_{j=b_i, \text{bord}(m)+1}^{m} p_j \]

\[ + (\prod_{j=\text{bord}(m)+1}^{m} p_j) \cdot B(\text{bord}(m)) \].

From knowledge of \( n, m, \text{bord}(m) \) and the prefix products, we can clearly compute the first term of \( B(m) \) in constant time.

Except for \( (\text{bord}(m) - m) \), the second term is essentially a sum of prefix products taken over all distinct borders of \( y_1 y_2 \ldots y_m \). Assuming that we had such a sum and \( B(\text{bord}(m)) \) at this point, we would clearly be able to compute \( B(m) \) whence also our variance, in constant time. In conclusion, we only need to show how to maintain knowledge of the
value of such sums during \texttt{maxborder}. But this is immediate, since the value $T(m)$ of the
sum at $m$ is clearly:

$$T(m) = (T(\text{bord}(m)) + 1) \cdot \prod_{j=\text{bord}(m)+1}^{m} p_j$$

and the product appearing in this expression is immediately obtained from our prefix
products.

We have thus established that, under the probabilistic assumptions which were made,
the variances of all prefixes of a string may be computed in linear time. This construction
may be applied, in particular, to each suffix $suf_i$ of a string $x$ while that suffix is being
handled by \texttt{insert} as part of procedure \texttt{buildtree}. This would result in an annotated
version of $T_x$ in overall quadratic time and space in the worst case.

6 Acknowledgements

We are indebted to R. Arratia for suggesting the problem of annotating statistical indi­
cices with expected values, and to M. Waterman for enlightening discussions and help­full
comments.

References

AHU-74 Aho, A.V., J.E. Hopcroft and J.D. Ullman, \textit{The Design and Analysis of Computer

Ah-90 Aho, A.V., Algorithms for Finding Patterns in Strings, in \textit{Handbook of Theoretical


AG-85 Apostolico, A. and Z. Galil (eds.), \textit{Combinatorial Algorithms on Words}, Springer-Verlag

AP-96 Apostolico, A. and F.P. Preparata, Data Structures and Algorithms for the String

AS-92 Apostolico, A. and W. Szpankowski, Self-Alignments in Words and Their Applications,


