1997

Some Theory and Practice of Greedy Off-Line Textual Substitution

Alberto Apostolico

Stefano Lonardi

Report Number:
97-045
SOME THEORY AND PRACTICE OF 
GREEDY OFF-LINE TEXTUAL SUBSTITUTION

Alberto Apostolico  
Stefano Lonardi

Purdue University  
Department of Computer Science  
West Lafayette, IN  47907

CSD-TR #97-045  
October 1997  
(Revised November 1997)
Some Theory and Practice of Greedy Off-line Textual Substitution

Alberto Apostolico* Stefano Lonardi
Purdue University and Università di Padova

Purdue CS TR 97-045
(September 97)

Abstract

Greedy off-line textual substitution refers to the following approach to compression or structural inference. Given a long textstring x, a substring w is identified such that replacing all instances of w in x except one by a suitable pair of pointers yields the highest possible contraction of x; the process is then repeated on the contracted textstring, until substrings capable of producing contractions can no longer be found. The paper examines computational issues arising in the implementation of this paradigm and describes some applications and experiments.

Keywords: off-line textual substitution, dynamic text compression, grammatical inference, suffix tree, substring statistics, augmented suffix tree.

1 Introduction

In data compression by textual substitution (see, e.g., [35, 34, 7]), substrings with multiple occurrences in a textstring are replaced by a suitable set of pointers to a unique common copy (for instance, by giving (1) a textstring position starting from which the substring can be recopied, and (2) the length of that substring). Disparate conventions, regarding issues such as the location of the common copy, and the mechanics of the encoding-decoding process, give rise to various macro schemes of compression. In general, the
relative performance of such schemes depends on many factors, including the often subtle interplay between pointer sizes and dictionary parameters (say, number of entries, and average length). Partly in response to this fact, techniques were devised for the compact encoding of integers in an unbounded domain (see, e.g., [11, 12, 2]). Unfortunately, however, the optimal implementation of the majority of macro schemes translates into NP-complete problems [35], even before the problem of encoding of pointers is taken into account. One noteworthy exception to this rule is represented by the well-known Lempel-Ziv schemes [40, 41, 42], which attain asymptotic optimality both in terms of compression achieved and algorithmic complexity. In Lempel-Ziv data compression (LZ), data can be processed on-line as it is read, a feature that nicely fits the standard paradigm of sequential transmission. The uni-directional or "polar" nature of pointers is crucial in determining the computational efficiency inherent to this scheme.

In some applications, like for instance in the production of a CD-ROM or magnetic disk for massive data dissemination, one could afford to perform the compression off-line, in particular, to issue pointers in either direction if this brings an increase in compression. Off-line heuristics may be expected to introduce extra time by whatever sequential implementation, but their possible implementation on parallel, perhaps dedicated architectures (see, e.g., [33, 9]), may be expected to achieve sufficient speed to process streams of large consecutive textfile windows consecutively in real-time for any practical purpose. Within the realm of sequential computation, investing more time in the compression may be desirable and feasible for information destined to be massively distributed, as long as the decompression can be still carried out fast and on-line [10]. In other situations, such as e.g., in backup archiving, the odds of having to restore the data might be feeble enough that even the requirement that this phase be on-line could be forfeited. Finally, as we briefly illustrate at the end of this paper, the study and implementation of macro schemes of the kind considered here may be of some interest in the germane field of inference of hierarchical structures or grammars for sequences (see, e.g., [16, 17, 29]).

The idea that some of the polarity or greedyness inherent to LZ schemes could be traded in for increased compression is intuitively appealing and not new. In [15, 3, 20], for instance, the authors discuss variations such as, e.g., relaxing the longest-match criterion in determining the next phrase within an LZ parse. The underlying goal is to try and converge faster to the entropy of the source. Studies on the redundancy of the Lempel Ziv code have been also performed along these lines, most recently in [39, 32]. However, none of these works addresses the problem of greedy off-line coding that we treat in the present paper. In view of the intractability of optimal off-line macro schemes, we concentrate here on the implementation of approximate methods such as one of the simplest possible steepest descent paradigm. This will consist of performing repeated stages in each one of which we identify a substring of the current version of the text yielding the maximum compression, and then replace all those occurrences except one with a pair of pointers to the untouched occurrence. As we shall see, this simple scheme already poses some interesting algorithmic problems, some of which we discuss in detail. However, the main issue that we try to address here is that of whether and to what extent a greedy use of
Figure 1: Overlapping and non-overlapping occurrences

Bi-directional pointers can yield good compression.

The structure of the paper is as follows. In section 2 we give some background and notation, and describe a data structure used to gather the statistics of the text. The overall design is presented in sections 3, 4 and 5. Finally, in section 6, we present experimental results and make some final remarks.

2 Computing Substring Statistics

We use \( \Sigma \) to denote an alphabet of symbols. For a string \( x \) over \( \Sigma \), the number of consecutive symbols in \( x \) is the length \( |x| \) of \( x \), and we write \( x[i] \), \( 1 \leq i \leq |x| \) to indicate the \( i \)-th symbol in \( x \). In the following, we assume \( |x| = n \). We use \( x[i,j] \) shorthand for the substring \( w \) of \( x \) composed by \( x[i] \cdot x[i + 1] \cdot \ldots \cdot x[j] \) where \( 1 \leq i \leq j \leq |x| \), and \( x[i,j] = x[i] \). Finally, substrings in the form \( x[1,j] \) are called prefixes of \( x \), and substrings in the form \( x[i,|x|] \) are called suffixes of \( x \). For any substring \( w \) of \( x \), we denote by \( f_w \) the number of nonoverlapping occurrences of \( w \) in \( x \). Clearly, \( f_w \) may be different from the total number of occurrences of \( w \). For example, \( w = \text{aba} \) occurs 11 times in \( x = \text{abaababaabaababaababaababa} \), with starting positions in the set \( \{1, 4, 6, 9, 12, 14, 17, 19, 21, 23, 25\} \) (cf. Fig. 1). However, occurrences starting at positions 4 and 6, or 12 and 14, etc., overlap with each other. We can have no more than 7 occurrences of \( w \) in \( x \) so that no two of them overlap. For instance, we could take those with starting positions in \( \{1, 4, 9, 12, 17, 21, 25\} \). Thus, \( f_{\text{aba}} = 7 \). To understand our interest in the count of nonoverlapping occurrences, assume that a substring \( w \) appears repeatedly in \( x \). Then, replacing all occurrences of \( w \) except one with a pointer to the unique reference copy might yield a more compact description of \( x \). If \( f_w \) is known, then it is also possible to assess beforehand the contraction in length that \( x \) would undergo following such an encoding. If, now, we were asked to identify the one substring \( w \) inducing the highest contraction on \( x \), we could clearly do so based on the \( f \)-value and length of the individual substrings. Choosing instead on the basis of the total number of occurrences would neither guarantee nor allow us to pre-compute the best contraction.

The computation of the statistics of all substrings of a string is an easy application of suffix trees (see Figure 3). Essentially, the suffix tree \( T(x) \) of a string \( x \) is a trie (digital search tree) collecting all the suffixes of \( x$ \), where $ is a special symbol not included in \( \Sigma \). More detailedly, such a tree is a \((|\Sigma| + 1)\)-ary rooted tree in which each leaf corresponds to a string position, and arcs are labeled with substrings of \( x \) in such a way that the concatenation of the labels on the path to leaf \( i \) \( (i = 1, 2, \ldots, n + 1) \) yields the suffix \( x[i, n + 1] \). Storing the tree requires linear space except for the labels on arcs. However,
procedure buildtree( x, T )
  begin
    T₀ ← Ø;
    for i = 1 to n + 1 do Ti ← insert(sufᵢ, Ti₋₁);
    Tx ← Tn₊₁;
  end

Figure 2: Direct construction of a suffix tree

the latter can be encoded each by a pair of pointers to suitable positions of x, thereby achieving an overall linear space bound. Any substring w of x is associated either with a node or with an arc of the tree (called the locus of w). The suffix tree for string x can be built in $O(|x|^²)$ time and space, as outlined in Figure 2: We start with an empty tree and add to it the suffixes of x one at a time. Conceptually, the insertion of suffix $sufᵢ$ ($i = 1, 2, ..., n + 1$) consists of two phases. In the first phase, we search for $sufᵢ$ in $Tᵢ₋₁$. 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 $sufᵢ$ that has a locus in $Tᵢ₋₁$. Let headᵢ be this prefix and α the locus of headᵢ. We can write $sufᵢ = headᵢ ∙ tailᵢ$ with tailᵢ nonempty. In the second phase, we need to add to $Tᵢ₋₁$ a path leaving node α and labeled tailᵢ. This achieves the transformation of $Tᵢ₋₁$ into $Tᵢ$.

We can assume that the first phase of insert is performed by a procedure findhead, which takes $sufᵢ$ as input and returns a pointer to the node α. The second phase is performed then by some procedure addpath, that receives such a pointer and directs a path from node α to leaf i. The details of these procedures are left for an exercise. As is easy to check, the procedure buildtree takes time $Θ(n²)$ in the worst case. It is possible to prove (see, e.g., [5]) that the expected length of headᵢ is $O(\log i)$, whence building $Tₓ$ by brute force requires $O(n \log n)$ expected time. A number of more clever constructions are available for the tree and some of its close variations [26, 36, 25, 37]. They avoid tracking down each suffix starting at the root, resulting in overall linear time up to a possible multiplicative factor of $\log |\Sigma|$ in the case of an unbounded alphabet. The very notion of headᵢ shows that suffix trees represent a natural habitat for the original LZ schemes, as reflected already in some of their earliest implementations [31].

When seeking all occurrences (with overlap), the number of occurrences of a substring $w$ is trivially given by the number of leaves reachable from the node closest to the locus of $w$, hence irrespective of whether or not $w$ ends in the middle of an arc: thus, to obtain this statistics, it is sufficient to label each internal node α with the number $c(α)$ of the leaves in the subtree rooted at α, as shown in Fig. 3.

The problem becomes more involved if we wanted to build a similar index for the statistics without overlap. A perusal of Figures 3 and 4 shows that this transition induces a twofold change in our structure: on the one hand, the weight in each node does no
longer necessarily coincide with the number of leaves; on the other, extra nodes must be now introduced to account for changes in the statistics that occur in the middle of arcs. The efficient construction of this augmented index in minimal form (i.e., with the minimum possible number of unary nodes) is quite elaborate [4]. For a string \( x \), the resulting structure is denoted \( \hat{T}(x) \) and called the \textit{Minimal Augmented Suffix Tree} of \( x \). It is not difficult to build \( \hat{T}_x \) in \( O(n^2) \) time and space by embedding the necessary weighting as part of the procedure \texttt{findhead}, hence at an expected cost of \( O(n \log n) \) [5]. The time required by the construction given in [4] is instead \( O(n \log^2 n) \) in the worst case. The number of auxiliary nodes can be bounded by \( O(n \log n) \), but it is not clear that such a bound is tight.

3 Implementing the Data Structures

When it comes to the actual allocation in memory of a suffix tree, one faces a number of design choices, prominent among which those pertaining to the implementation of nodes. There are three main possibilities in this regard:

- the node is implemented as an array of size \( |\Sigma| \). This yields fast searches, but is likely to introduce an unbearable amount of waste even for small alphabets;

- the node is implemented as a linked list (or, better, as a balanced search tree). This keeps space to a minimum, but introduces an overhead on the search.
Figure 4: Node and weight changes are required in the index storing statistics without overlaps.

- The adjacency of a node is realized as part of a global hash coding. This yields expected constant time search within overall \( \Theta(n \log n) \) space.

Figure 5 displays a linked implementation of the suffix tree of our example textstring. As already noted, the substrings representing edge labels are not stored explicitly in the nodes but rather encoded each by an ordered pair of integers to a unique common copy of \( \Sigma \), so as to achieve overall linear space. However, even linear space can be problematic: at 20 bytes per node and with a number of nodes 1.5 times the number of symbols in the input string, as typically featured in our experiments, a text of size \( n \) needs approximately \( 30n \) bytes of storage space. In general, although the size of the suffix tree depends on the particular implementation, one might expect it to be never lower than 15–20 bytes per input symbol, or bps. Various related or alternative structures have been devised with the primary objective of space minimization, among which the Patricia tree (12 bps) [27], the suffix-array (6 bps) [25], the suffix-cactus (9 bps) [21,22] and the level compressed trie (11 bps) [1]. In general, these space savings are achieved at the expense of higher complexity in either construction, or searching, or both: thus, for instance, the suffix array and the PAT tree need \( O(n \log n) \) time for the construction (\( O(n) \) on average for the array) and \( O(|w| + \log n) \) when searching for a string \( w \).

We use \( u \rightarrow w \) to denote the node, if it exists, precisely at the end of the path in \( T_x \) labeled by the string \( w \). In our realization, \( u \rightarrow w \) contains the following items:
Figure 5: The augmented suffix tree for abaababaabaababaababa$

- two indices $[i, j]$ identifying an occurrence of $w$ in $x$, i.e., such that $w = x[i, j]$;
- one pointer to the list of children and one to the list of siblings of $<w>$;
- one counter to store the number of nonoverlapping occurrences of $w$ in $x$.

The data structure allocating the textstring $x$ should support somewhat contrasting primitives such as, for instance, efficient string searching and repeated substring deletions. To accommodate the repeated contractions of $x$, the latter is maintained in a linked list of dynamic arrays, as follows. At the beginning, the text is read from the source into a single array of length $n$. Subsequently, the removal of the occurrences of a substring $w = aba$ will partition the array into linked fragments, as shown in figure 6. These arrangements are complemented by refresh cycles that will recombine the text in a single array, from time to time, to counteract excessive fragmentations.

Repeatedly building the suffix tree at each stage exacts a considerable toll irrespective of the method adopted. Ideally, one would like to build the tree once and then maintain it, together with updated statistics, following every substring selection and removal. Linear time algorithms for dynamically maintaining the tree under deletion of a string were originally proposed by McCreight together with his construction. Similar problems have been studied by Fiala and Green [15] in the context of sliding window compression. Recently, Gu et al. [19] introduced a new data structure for dynamic text indexing that supports insertion and deletion of a single character in $O(\log n)$ time and $i$ searches for all the occurrences $occ_w$ of a string $w$ in $O(|w| + occ_w \log i + i \log |w|)$. Larsson [23] shows that the algorithm by Ukkonen can be easily extended to accommodate the sliding window update of the suffix tree in linear time. Recently Ferragina [13, 14] studied the problem.
to delete a string from a suffix tree on words. Baker [6] introduced the P-suffix tree to solve the parametrized text indexing problem, i.e., the problem to finding occurrences of a pattern in a text even when global substitutions have modified those occurrences. However, the more general problem of modifying a suffix tree or statistical index so as to reflect the deletion from the corresponding textstring of all the occurrences of a given substring does not seem to have found a satisfactory solution. In our experiments, we built the suffix tree from scratch at every step.

4 Choosing and Computing a Gain Measure

By “gain measure”, we refer here to the function that will be evaluated at every node of $T'$ in order to select the best substring substitution. In practice, it is not easy to define precisely such a measure, as we explain below.

The main difficulty is due to the fact that at the time when we need to compute the contraction that would be induced by a particular substring, we lack some important costs such as those associated with the optimal encodings of pointers or integers, which can be computed precisely only at the outset. Letting $l(i)$ represent the number of bits needed to encode integer $i$, we assume for simplicity $l(i) = \lceil \log i \rceil$ at the time the gain is computed. Note that this choice does not affect the appraisal of final compression, the latter being based on purely empirical measures. Along the same lines, one could choose an expression for $l$ that reflects more accurately the efficient encoding of integers in an unknown range [11, 12, 2]. However, as long as the ultimate encoding of the compressed string is not based on those representations, but rather on some statistical treatment (e.g., Huffman encoding), there is hardly any sense in resorting to them and hardly any way to compute $l(i)$ accurately at this stage.

With this choice made, we describe now in succession two possible measures of gain. For a string $w$ of length $|w| = m_w$ the $f_w$ copies of $w$ require $B f_w m_w$ bits in the plain text. In practice, the value of $B$ is appraised based on the zero-order entropy of the source: the plain text is Huffman encoded, and then $B$ is set to the average length of a symbol.

In our first measure, we assume that one of the $f_w$ copies of $w$ is left in the original
text, marked by a "literal identification" bit, while the remaining \( f_w - 1 \) copies are encoded by pointers, each pointer being preceded by a suitable identification bit. This results in \( B \cdot m_w + 1 \) bits for the untouched copy and \( (f_w - 1)(I(n) + l(m_w) + 1) \) bits for the copies, yielding a gain (or loss):

\[
G(w) = B \cdot f_w m_w - B \cdot m_w - (f_w - 1)(1 + l(n) + l(m_w)) - 1
\]

\[
= (f_w - 1)B \cdot m_w - (f_w - 1)(1 + l(n) + l(m_w)) - 1
\]

\[
= (f_w - 1)(B \cdot m_w - 1 - l(n) - l(m_w)) - 1
\]

\[
= (f_w - 1)(B \cdot m_w - l(n) - l(m_w)) - f_w
\]

If now \( w \) is the string maximizing \( G \) throughout the nodes of \( \hat{T}_x \) (trivially, it is safe to neglect the \( f \)-values attainable in the middle of arcs), then the above substitution is performed, and the process is repeated: the suffix tree is updated and searched again for the next best substitution. These iterations terminate as soon as the optimum \( G \) becomes zero or goes below some other convenient and predetermined threshold \( t \).

There are some complications, though: from the second step on, the text is composed by literals interspersed with pointers, and the contribution to \( G \) of pointers and literals differ. One possibility is to consider the text partitioned into a number of segments separated by pointers, and treat these segments individually. A related, albeit less critical issue, would then be to decide which one of the \( f_w \) occurrences to preserve as the reference copy of \( w \). These complications lead to formulate an alternative scheme, in which all the \( f_w \) occurrences of the best string \( w \) are removed from the text, while \( w \) itself is saved in an auxiliary data structure that contains:

- the length \( m_w \), at a cost of \( l(m_w) \) bits;
- the string \( w \), that is \( B \cdot m_w \) bits long;
- the value of \( f_w \), at a cost of \( l(f_w) \) bits;
- the \( f_w \) positions of \( w \) in \( x \), at a global cost bounded by \( f_w l(n) \) bits.

The corresponding gain is now computed as:

\[
G(w) = B \cdot f_w m_w - l(m_w) - B m_w - l(f_w) - f_w l(n)
\]

\[
= (f_w - 1)B m_w - l(m_w) - l(f_w) - f_w l(n)
\]

This second framework reflects more accurately the "off-line" nature of the method, in particular, there is no difference in treatment between the first selection and the rest. The outer structure of the encoder built along these lines is displayed in the pseudocode of Figure 7.
text = x
do {
    mast = create_min_augm_suffix_tree(text);
    (substr,G(substr)) = compute_gain(mast);
    if (G(substr) > 0) {
        write the encoding;
        x = delete all the occurrence of substr from text;
    }
} while (G(substr) > t);
run huffman on the encoding;

Figure 7: The top level structure of the encoder.

1: abaabaabaabaabaaba$  Substituted substring: "aba"
2: bababa$  Substituted substring: "ba"
--------------------------- Final encoding:
sublen = [3 2]
substr = [ababa]
abspol = [0 0]
repol = [0 2 0 2 0 0]
occurr = [5 3]
text = [$]

Figure 8: A run of the code on the string abaabaabaabaabaabaababa$.

5 Encoding the Output

The iterated substring substitution process is exemplified in Figure 8. The first iteration results in the choice of aba; the second, of ba. The collection of data representing the output encoding appears at the bottom of the figure.

As seen in the figure, the final encoding requires a few dynamic arrays. At the end of a generic iteration i, resulting in the choice of substring w, such arrays are as follows.

- sublen[i] contains |w| − min_length; the latter term represents a minimum acceptable length and is 0 in the example but 2 in our experiment;
- substr[k,k+sublen[i]+min_length-1] contains w, starting from the end k − 1 of the substring identified in iteration i − 1;
- occurr[i] contains f_{w} − min_occurrence; the latter term represents a minimum acceptable f-value, which is 0 in the example but 2 in the experiments;
<table>
<thead>
<tr>
<th>Array</th>
<th>Type of final coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Huffman</td>
</tr>
<tr>
<td>sublen</td>
<td>RLE + Huffman</td>
</tr>
<tr>
<td>substr</td>
<td>Huffman</td>
</tr>
<tr>
<td>occur</td>
<td>RLE + Huffman</td>
</tr>
<tr>
<td>abspol</td>
<td>plain</td>
</tr>
<tr>
<td>abspoh</td>
<td>Huffman</td>
</tr>
<tr>
<td>relpol</td>
<td>plain</td>
</tr>
<tr>
<td>relpoh</td>
<td>Huffman</td>
</tr>
</tbody>
</table>

Figure 9: Illustrating one of the possible final encodings of the arrays.

- **abspoh[i]** and **abspol[i]** contains the higher and the lower byte of the absolute position of the first occurrence;

- **relpoh[j]** and **relpol[j, max(occur[i] + min_occur - 1)]** contains the higher and the lower byte of the consecutive displacements of the other occurrences, in sorted order.

Finally, **array text** stores whatever may be left of the original textstring at the end of the process. In general, the number 255 is reserved to indicate that a current datum overflows standard space so that an additional byte is devoted to its storage.

As mentioned, the overall compression depends not only on the structure of \( G \) but also on the particular encoding chosen for the arrays in the output. Possible choices suggested by our experiments are summarized in Figure 9. At the end of the iterated substitutions some arrays exhibit a high entropy (e.g., those containing the lower byte of absolute and relative positions), so that their entries could be block-encoded as plain numbers. Others tend to show long runs of identical values (e.g., those storing substring lengths and numbers of occurrences), and can be significantly compressed by a cascade of run-length and Huffman encoding. The remaining arrays are Huffman encoded. Better results might be expected using arithmetic coding, rather than Huffman coding.

As one would expect, the bulk of the output is represented by the (lower byte of the) relative positions **relpol**, and by the array **text**. Our experiments showed that, although the former is practically uncompressible, in principle **substr** could be compressed again.

These considerations make it clearer why a fully reliable computation of \( l \)-values during any substring selection stage is hard. A number of ways exist in principle to mitigate this problem. For instance, one could resort to block or fixed codes like the ones described in [12, 31], or to dynamic Huffman encoding of \( l \) based on past symbols, or even keep a statistics of the code generated so far and use this history to estimate the final value of \( l \). However, we collected no evidence that any of these variations would import enough benefits to warrant their induced overhead.
The values assigned to parameters such as the minimum match length, minimum number of occurrences and the threshold \( t \), also have some impact on the compression achieved. These, too, are difficult to fine-tune, because of their subtle relation to the structure of \( G \).

Before closing this Section, we point out that decoding a compressed text string given in the above representation is easily done in linear time. The details are left for an exercise.

### 6 Experimental Results and Conclusion

Our data structures and algorithms were coded in C++ using the Standard Template Library (STL). The latter is a very clean collection of containers and generic functions endowing C++ with some of the features of higher-order imperative languages [28]. Overall, the program consists of circa 6,000 lines of code. Below we use OFF-LINE to refer to it.

The tables report results from experiments carried out on a small set of test files: paper2 and progl are ASCII files from the Calgary Corpus, mitoDNA, chr-I and chr-VI are, respectively the mitochondrial genome and the first and sixth chromosome of the yeast (Saccharomyces cerevisiae strain S288), hiv.pcb is the collection of the three-dimensional coordinates of the spatial configuration of the hiv, and camera is a 256-level gray scale image. The relative abundance of genetic sequences in our experiments is due on one hand to the circumstance that those sequences are commonly held to be hardly compressible, on the other to some recent revival of interest in the compressibility of genetic sequences as a possible measure of biological significance (see, e.g., [18, 30, 24]), which
Figure 12: Comparison table including GZIP

<table>
<thead>
<tr>
<th>Coding</th>
<th>plain text</th>
<th>paper2</th>
<th>progl</th>
<th>mitoDNA</th>
<th>chr-I</th>
<th>camera</th>
<th>hiv.pcb</th>
<th>chr-VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huffman (PACK)</td>
<td>82201</td>
<td>71648</td>
<td>78521</td>
<td>230195</td>
<td>66336</td>
<td>108922</td>
<td>270148</td>
<td></td>
</tr>
<tr>
<td>LZ-78 (COMPRESS)</td>
<td>47736</td>
<td>43093</td>
<td>18152</td>
<td>63144</td>
<td>58947</td>
<td>45859</td>
<td>74077</td>
<td></td>
</tr>
<tr>
<td>LZ-77 (GZIP)</td>
<td>36165</td>
<td>27148</td>
<td>17891</td>
<td>62935</td>
<td>55367</td>
<td>25499</td>
<td>73873</td>
<td></td>
</tr>
<tr>
<td>OFF-LINE</td>
<td>29754</td>
<td>16273</td>
<td>19371</td>
<td>66264</td>
<td>48750</td>
<td>22443</td>
<td>78925</td>
<td></td>
</tr>
<tr>
<td>OFF-LINE-PREF</td>
<td>32798</td>
<td>22427</td>
<td>17074</td>
<td>62369</td>
<td>51034</td>
<td>20982</td>
<td>73903</td>
<td></td>
</tr>
</tbody>
</table>

excited our curiosity. In terms of the parameters defined earlier, all experiments use a threshold of value 1, and a min-length and min-occur of 2. As the Table of Figure 10 shows, the performance of OFF-LINE is better in all cases except one. Some, but not all, of the scores achieved could be marginally improved upon by incorporating in OFF-LINE the rule that, following the selection of string \( w \), all prefixes of \( w \) capable of producing further compression are immediately used in the encoding (Figure 11). In other words, the encoding overhead introduced by such a complication seems to counterbalance the increased compression. The advantages of our off-line approach seem to fade in a comparison that would include GZIP (see Figure 12). This may surprise, since the latter purports to incarnate a scheme, LZ-77, which in terms of vocabulary build-up would appear to be closer to OFF-LINE than LZ-78. However, a thoroughly faithful comparison to GZIP is made difficult by the many heuristics employed in the latter, among which the critical role played by the window size. Crossing the boundary of textual substitution methods, the block-sorting-method BZIP based on [8] outperformed GZIP on all inputs and OFF-LINE on all inputs except one.

A number of interesting questions were brought up by these experiments which would warrant additional effort. These include possible provisions for variable window sizes, better ways to approximate the gain function \( G \), the feasibility and usefulness of reiteration of treatment following the first application of OFF-LINE, and several issues pertaining to the computational efficiency achievable by sequential and parallel implementations. Among the latter, a prominent concern would be to devise efficient algorithms that avoid building the statistical index from scratch at each iteration, and better storage and matching algorithms for our data structure.

As mentioned, the parallel implementation of the method might result in relatively clean and very fast real-time applications. The table in Figure 13 shows the modest number of iterations of the main loop performed on our inputs.

Finally, it is interesting to examine the performance of OFF-LINE when used as a tool for inferring hierarchical grammatical structures in sequences. Figure 14 displays the grammar inferred for our example string by the SEQUITUR algorithm by Nevill-Manning et al. [29], which is essentially patterned after an LZ parsing scheme. Except for the
Figure 13: Iterations of the main loop of OFF-LINE under the various inputs

\[
\begin{align*}
S & \to \text{DDC}$
A & \to \text{ba} \\
B & \to \text{aA} \\
C & \to \text{BA} \\
D & \to \text{BC}
\end{align*}
\]

Figure 14: Hierarchical grammar produced by SEQUITUR for \text{abaababaabaababaababa}$

one involving the start symbol $S$, productions are constrained to have right-hand sides consisting of digrams. A grammar subtended by the strings of Figure 8 is shown in Figure 15. Re-iteration of the treatment would expose productions of the form $C \to AAB$ and $D \to AB$, and finally $S \to CCD$.

\[
\begin{align*}
S & \to \text{AABAABAB} \$
A & \to \text{aba} \\
B & \to \text{ba}
\end{align*}
\]

Figure 15: First layer of grammar produced by OFF-LINE.
References


