SOME THEORY AND PRACTICE OF GREEDY OFF-LINE TEXTUAL SUBSTITUTION

Alberto Apostolico[*] Stefano Lonardi
Purdue University and Università di Padova

1 Introduction

Greedy off-line textual substitution refers to the following *steepest descent* 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. This paper examines computational issues and performances resulting from implementations of this paradigm in preliminary applications and experiments. There is enough motivation for studying this and the many other conceivable variants of greedy off-line methods that fall in the wide and relatively unexplored gap between the classical, linear and polar methods introduced by [15] and [16], and the generally intractable optimal macro schemes [12]. Apart from intrinsic interest, these methods may find use in the compression of massively disseminated data, e.g., of the kind considered in [8], and lend themselves to efficient parallel implementation, perhaps on dedicated architectures such as, e.g., in [7].

Given a *string* x over an alphabet Σ and any substring w of x, we denote by f_w the number of *nonoverlapping* occurrences of w in x. Clearly, f_w may differ from the total number of occurrences of w. For example, $w =$ `aba` occurs 11 times in $x =$ `abaababaabaababaababababaa`, but no more than 7 such occurrences can be chosen so that no two of them overlap. Our scheme is based on repeated identification of a substring w of x such that replacing all occurrences of w except one with a pointer to the unique reference copy yields the highest contraction of x. This process requires knowledge of the f-values and lengths of all substrings of the current version of x.

---

[*] Corresponding Address: Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, IN 47907, USA. `{axa,stelo}@cs.purdue.edu`. Work supported in part by NSF Grants CCR-9201078 and CCR-9700276, by NATO Grant CRG 900293, by British Engineering and Physical Sciences Research Council Grant GR/L19362, and by the National Research Council of Italy.

The computation of the statistics of all substrings of a string x is an easy application of the *suffix tree* T_x of x. As is well known, the latter is a trie (digital search tree) collecting all the suffixes of $x\$$, where \$ is a special symbol not included in Σ. The tree in compact form is built by iterated insertion of consecutive suffixes in $\Theta(n^2)$ worst case time and $O(n \log n)$ expected time (see, e.g., [3]). A number of more clever constructions are available achieving linear time for finite alphabets (see, e.g., [13]). The number of occurrences (with overlap) of a string w of x is trivially given by the number of leaves reachable from the node closest to the locus of w in T_x, irrespective of whether or not w ends in the middle of an arc. Thus, labeling every internal node α of T_x with the number $c(\alpha)$ of the leaves in the subtree rooted at α yields this statistics for all substrings of x.

The problem becomes more involved if we wanted to build a similar index for the statistics *without overlap*. A perusal of Figure 1 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 [2]. For a string x, the resulting structure is denoted $\hat{T}_x$ and called the *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 individual suffix insertions, hence at an expected cost of $O(n \log n)$ [3]. The time required by the construction given in [2] 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.

2 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 first one is to implement the node 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 second one is to implement the node as a linked list (or, better, as a balanced search tree). This keeps space to a minimum, but introduces an overhead on the search. The third one is to realize the adjacency of a node as part of a global hash coding. This yields expected constant time seach within overall $\Theta(n \log n)$ space.

As is well known, 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 x, 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

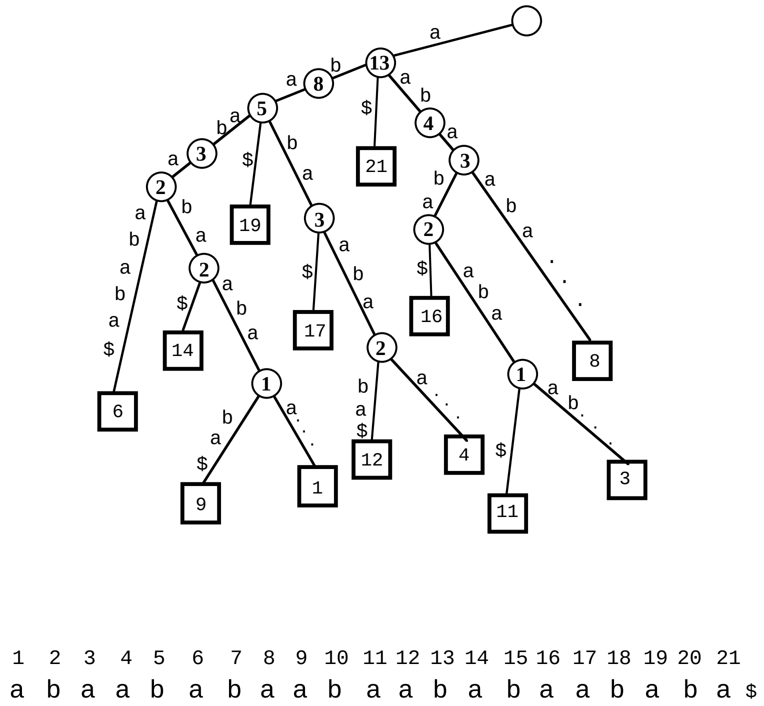

Figure 1: Node and weight changes are required in the index storing statistics without overlaps

structures have been devised with the primary objective of space minimization (cf., e.g., [13], [9], and references therein). 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 $<w>$ to denote the node, if it exists, precisely at the end of the path in $\hat{T}_x$ labeled by the string w. In our realization, $<w>$ contains the following items: (1) two indices $[i, j]$ identifying an occurrence of w in x, i.e., such that $w = x[i, j]$; (2) one pointer to the list of children and one to the list of siblings of $<w>$; (3) 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. 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 subsequently by others. However, we did not find an existing satisfactory solution to the problem of quickly modifying our statistical index so as to reflect the deletion from the corresponding textstring of *all* the occurrences of a given substring. In our experiments, every new version of the suffix tree was built from scratch.

3 Choosing and Computing a Gain Measure

By "gain measure", we refer here to the function that will be evaluated at every node of $\hat{T}_x$ 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 (see. e.g., [1]). 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 $B \cdot f_w \cdot m_w$ 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 $(f_w - 1)(l(n) + l(m_w) + 1)$ bits for the copies, yielding a gain (or loss) given by $G(w) = B \cdot f_w m_w - B \cdot m_w - (f_w - 1)(1 + l(n) + l(m_w)) - 1 = (f_w - 1)B \cdot m_w - (f_w - 1)(1 + l(n) + l(m_w)) - 1 = (f_w - 1)(B \cdot m_w - 1 - l(n) - l(m_w)) - 1 = (f_w - 1)(B \cdot m_w - l(n) - l(m_w)) - f_w$.

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 com-

```
x = text
do {
  mast = create_min_augm_suffix_tree(x);
  (substr,G(substr)) = compute_gain(mast);
  if (G(substr) > 0) {
    write the encoding;
    x = delete all the occurrence of substr from x;
  }
} while (G(substr) > t);
run huffman on the encoding;
```

Figure 2: The top level structure of the encoder.

posed 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: (1) the length m_w, at a cost of $l(m_w)$ bits; (2) the string w, that is $B \cdot m_w$ bits long; (3) the value of f_w, at a cost of $l(f_w)$ bits; (4) 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 $G(w) = B \cdot f_w m_w - l(m_w) - B \cdot m_w - l(f_w) - f_w l(n) = (f_w - 1) B \cdot m_w - l(m_w) - l(f_w) - f_w l(n)$. 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 2.

4 Encoding the Output

The iterated substring substitution process is exemplified in Figure 3. 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_occurr`; the latter term represents a minimum acceptable f-value, which is 0 in the example but 2 in the experiments; `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,j+occur[i]+min_occurr-1]` contains the higher and the lower byte of the

```
1: abaababaabaababaababa$   Substituted substring: "aba"
2: bababa$                  Substituted substring: "ba"
--------------------------- Final encoding:
sublen = [3 2]
substr = [ababa]
abspol = [0 0]              abspoh = [0 0]
relpol = [0 2 0 2 0 0]      relpoh = [0 0 0 0 0 0]
occurr = [5 3]
text   = [$]
```

Figure 3: A run of the code on the string `abaababaabaababaababa$`.

Array	Type of final coding
`text`	Huffman
`sublen`	RLE + Huffman
`substr`	Huffman
`occurr`	RLE + Huffman
`abspol`	plain
`abspoh`	Huffman
`relpol`	plain
`relpoh`	Huffman

Figure 4: Illustrating one of the possible final encodings of the arrays.

consecutive displacements of the other occurrences. 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 4. 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 [14], 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

Coding	`paper2`	`progl`	`mitoDNA`	`chr-I`	`camera`	`hiv.pcb`	`chr-VI`
plain text	82201	71648	78521	230195	66336	108922	270148
Huffman (PACK)	47736	43093	18152	63144	58947	45859	74077
LZ-78 (COMPRESS)	36165	27148	17891	62935	55367	25499	**73873**
OFF-LINE	**32798**	**22427**	**17074**	**62369**	**51034**	**20982**	73903

Figure 5: Comparing Off-line with Huffman and LZ-78

Coding	`paper2`	`progl`	`mitoDNA`	`chr-I`	`camera`	`hiv.pcb`	`chr-VI`
plain text	82201	71648	78521	230195	66336	108922	270148
OFF-LINE	**32798**	**22427**	**17074**	62369	51034	**20982**	**73903**
OFF-LINE-PREF	33240	22928	17117	**62336**	**51024**	21255	73909

Figure 6: Forcing all prefixes of a selected word to be part of the encoding

mitigate this problem. For instance, one could resort to block or fixed codes, 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 textstring given in the above representation is easily done in linear time. The details are left for an exercise.

5 Experimental Results and Conclusion

Our data structures and algorithms were coded in C++ using the Standard Template Library (STL), a clean collection of containers and generic functions endowing C++ with some of the features of higher-order imperative languages. 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 sixt 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. which 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 5 shows, the performance of OFF-LINE

Coding	paper2	progl	mitoDNA	chr-I	camera	hiv.pcb	chr-VI
plain text	82201	71648	78521	230195	66336	108922	270148
Huffman (Pack)	47736	43093	18152	63144	58947	45859	74077
LZ-78 (Compress)	36165	27148	17891	62935	55367	25499	73873
LZ-77 (Gzip)	**29754**	**16273**	19371	66264	**48750**	22443	78925
Off-line	32798	22427	**17074**	62369	51034	**20982**	**73903**
Off-line-pref	33240	22928	17117	**62336**	51024	21255	73909

Figure 7: Comparison table including Gzip

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 6). In other words, the encoding overhead introduced by such a complication seems to counterbalance the possible increase in compression. On the other hand, variants built along these lines were found to be is considerably faster. The advantages of our off-line approach seem to fade in a comparison that would include Gzip (see Figure 7). 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 [5] 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.

Figures 8 and 9 show the results achieved by a preliminary *sloppy* variant of the algorithm, in which more than just one substring selection and substitution is performed between two consecutive updates of the statistical index. Of course such an approach saves time on one hand, but it risks blurring the perception of the best candidates for substitution. In our implementation, a heap is maintained with the statistical index, containing at each step the Q best words in terms of G, for some chosen value of the parameter Q. Between any two consecutive index reconstructions, the Q strings in the heap are retrieved and used in succession in a contraction step for the text. It is possible at some point that a string from the heap will be no longer found in the contracted text. In fact, part of the words in the heap turn out to be useless in general. In any case, as soon as all words in the heap have been considered, a new augmented suffix tree is built on the contracted text.

Q	Subst.	Trees	Col_2/Col_3	Speedup	Compr. size	Percent.
1	787	788	1.0	1.0	32798	100.00%
10	799	83	9.6	6.2	32837	100.11%
100	910	13	70.0	23.7	33113	100.96%
1000	1174	4	293.5	22.5	33688	102.71%

Figure 8: OFF-LINE-SLOPPY on `paper2` (82201 bytes). The columns indicate, from right to left: size of the heap, total substring selections-substitutions, total tree constructions, ratio of Columns 2 to Column 3, normalized time, size of compressed string, and same size as a percentage of the size achieved by the standard method.

Q	Subst.	Trees	Col_2/Col_3	Speedup	Compr. size	Percent.
1	165	165	1.0	1.0	17074	100.0%
10	170	22	7.7	6.4	17141	100.4%
100	303	7	43.3	14.1	17440	102.1%
1000	619	3	206.3	11.3	17861	104.6%

Figure 9: OFF-LINE-SLOPPY on `mito` (78521 bytes).

As the figures display, the number of individual substring substitution passes over the text grows with the maximum allowed size of the heap. On the other hand, we spend less and less time building weighted suffix trees. The overall result is, within a wide interval, a considerable speed up with respect to the eager version of OFF-LINE without substantial penalty in compression performance. When the size of heap becomes too large (approximately Q=1000 in our experiments) only a small subset of the words in the heap is used: most of the computational effort is spent in pattern searching, which results in deterioration of both speed and compression.

As mentioned, the parallel implementation of the method might result in relatively clean and very fast real-time applications. The table in Figure 10 shows the modest number of iterations of the main loop performed by OFF-LINE on our inputs. The experiments that subtend Figures 8 and 9 suggest that such a figure might become negligible in practical cases. This means that in a parallel implementation, the most expensive tasks, represented by the tree constructions, can be limited so that very little time is charged overall by it.

Finally, it is interesting to examine the performance of OFF-LINE when used as

Coding	`paper2`	`progl`	`mitoDNA`	`chr-I`	`camera`	`hiv.pcb`	`chr-VI`
plain text (bytes)	82201	71648	78521	230195	66336	108922	270148
OFF-LINE	788	577	165	71	634	216	27
OFF-LINE-PREF	776	615	168	73	641	218	29

Figure 10: Iterations of the main loop of OFF-LINE under the various inputs

a tool for inferring hierarchical grammatical structures in sequences. The grammar inferred for our example string by the SEQUITUR algorithm by Nevill-Manning *et al.* [11], which is essentially patterned after an LZ parsing scheme, consists of the productions: $S \rightarrow DDC\$; A \rightarrow \mathtt{ba}; B \rightarrow \mathtt{a}A; C \rightarrow BA; D \rightarrow BC$. Except for the 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 3 is: $S \rightarrow AABAABAB\$; A \rightarrow \mathtt{aba}; B \rightarrow \mathtt{ba}$. Re-iteration of the treatment would expose productions of the form $C \rightarrow AAB$ and $D \rightarrow AB$, and finally $S \rightarrow CCD$.

References

[1] A. Apostolico and A. F. Fraenkel. Robust transmission of unbounded strings using fibonacci representations. *IEEE Transactions on Information Theory*, 33(2):238–245, 1987.

[2] A. Apostolico and F. P. Preparata. Data structures and algorithms for the strings stastitics problem. *Algorithmica*, 15(5):481–494, May 1996.

[3] A. Apostolico and W. Szpankowski. Self-alignment in words and their applications. *J. Algorithms*, 13(3):446–467, 1992.

[4] T. C. Bell, J. G. Cleary, and Ian H. Witten. *Text Compression.* Prentice Hall, 1990.

[5] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. *TR Digital Equipments Corporation*, (124), May 1994.

[6] M. Cohn and R. Khazan, Parsing with suffix and prefix dictionaries. In *DCC: Data Compression Conference*, pages 180–189. IEEE Computer Society TCC, 1996.

[7] M. Crochemore and W. Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Inform. Process. Lett.*, 38:57–60, 1991.

[8] S. DeAgostino and J. A. Storer. On-line versus off-line computation in dynamic text compression. *Inform. Process. Lett.*, 59(3):169–174, 1996.

[9] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22, 1993.

[10] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[11] C. Neville-Manning, I. H. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In *DCC: Data Compression Conference*, pages 244–253. IEEE Computer Society TCC, 1994.

[12] J. A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, 1988.

[13] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[14] I. H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[15] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inform. Theory*, IT-23(3):337, May 1977.

[16] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Inform. Theory*, 24(5), September 1978.