Augmenting Suffix Trees, with Applications

Yossi Matias[1] ⋆, S. Muthukrishnan[2] ⋆⋆, Süleyman Cenk Ṣahinalp[3] ⋆⋆⋆, and Jacob Ziv[4] †

[1] Tel-Aviv University, and Bell Labs, Murray Hill
[2] Bell Labs, Murray Hill
[3] University of Warwick and University of Pennsylvania
[4] Technion

Abstract. Information retrieval and data compression are the two main application areas where the rich theory of string algorithmics plays a fundamental role. In this paper, we consider one algorithmic problem from each of these areas and present highly efficient (linear or near linear time) algorithms for both problems. Our algorithms rely on augmenting the *suffix tree*, a fundamental data structure in string algorithmics. The augmentations are nontrivial and they form the technical crux of this paper. In particular, they consist of adding extra edges to suffix trees, resulting in Directed Acyclic Graphs (DAGs). Our algorithms construct these "suffix DAGs" and manipulate them to solve the two problems efficiently.

1 Introduction

In this paper, we consider two algorithmic problems, one from the area of Data Compression and the other from Information Retrieval. Our main results are highly efficient (linear or near linear time) algorithms for these problems. All our algorithms rely on the *suffix tree* [McC76], a versatile data structure in combinatorial pattern matching. Suffix trees, with suitably simple augmentations, have found numerous applications in string processing [Gus98,CR94]. In our applications too, we augment the suffix tree with extra edges and additional information. In what follows, we describe the two problems and provide some background information, before presenting our results.

⋆ Department of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel; and Bell Labs, Murray Hill, NJ, 07974, USA; `matias@math.tau.ac.il`. Partly supported by Alon Fellowship.

⋆⋆ Bell Labs, Murray Hill, NJ, 07974, USA; `muthu@research.bell-labs.com`.

⋆⋆⋆ Department of Computer Science, University of Warwick, Coventry, CV4-7AL, UK; and Center for BioInformatics, University of Pennsylvania, Philadelphia, PA, 19146, USA; `cenk@dcs.warwick.ac.uk`. Partly supported by ESPRIT LTR Project no. 20244 - ALCOM IT.

† Department of Electrical Engineering, Technion, Haifa 32000, Israel; `jz@ee.technion.ac.il`.

1.1 Problems and Background

We consider the *document listing problem* of interest in Information Retrieval and the HYZ *compression problem* from context-based Data Compression.

The Document Listing Problem. We are given a set of documents $T = \{T_1, \dots, T_k\}$ for preprocessing. Given a query pattern P, the problem is to output a list of all the documents that contain P as a substring.

This is different from the standard query model where we are required to output all the *occurrences* of P. The standard problem can be solved in time proportional to the number of occurrences of P in T using a suffix tree. In contrast, our goal in solving the document listing problem is to generate the output with a running time that depends on the number of documents that contain P. Clearly, the latter may be substantially smaller than the former if P occurs multiple times in the documents.

A related query is where we are required to merely report the *number* of documents that contain P. An algorithm that solves this problem in $O(|P|)$ time is given in [Hui92], which is based on data structures for computing lowest common ancestor (LCA) queries.

The document listing problem is of great interest in information retrieval and has independently been formulated in many scenarios (see pages 124-125 in [Gus98] for a "morbid" application), for example in discovering gene homologies [Gus98].

The HYZ Compression Problem. Formally the (α, β)-$\mathcal{HYZ}$ compression problem is as follows. We are given a binary string T of length t. We are asked to replace disjoint *blocks* (substrings) of size β with desirably shorter codewords. The codewords are selected in a way that it would be possible for a corresponding decompression algorithm to compute the original T out of the string of codewords. This is done as follows. Say the first $i-1$ such blocks have been compressed. To compute the codeword c_j for block j, we first determine its *context*. The context of a block $T[i:l]$ is the longest substring $T[k:i-1], k < i$, of size at most α such that $T[k:l]$ occurs earlier in T. The codeword c_j is the ordered pair $\langle \gamma, \lambda \rangle$ where γ is the length of the context of block j and λ is rank of block j with respect to the context, according to some predetermined ordering. For instance, one can use the lexicographic ordering of all distinct substrings of size exactly β that follow any previous occurrence of the context of block j in the string. The (α, β)-$\mathcal{HYZ}$ compression scheme is based on the intuition that similar symbols in data appear in similar contexts. At the high level, it achieves compression by sorting context-symbol pairs in lexicographic order, and encoding each symbol according to its context and its rank. Thus it is a *context-based* scheme.

The (α, β)-$\mathcal{HYZ}$ compression problem has been proposed recently in [Yok96] and [HZ95,HZ98]. The case considered in [Yok96] is one where $\beta = O(1)$ and α is unbounded. During the execution of this algorithm, the average length of a codeword for representing a β-sized block is shown to approach the *conditional entropy* for the block, $H(C)$, within an additive term of $c_1 \log H(C) + c_2$ for constants c_1 and c_2, provided that the input is generated by a limited order Markovian source.[1] An independent result [HZ95,HZ98] is more general since it applies to all ergodic sources and yields essentially the best possible non-asymptotic compression even for a moderate amount of memory (at least for some ergodic sources). Of relevance to us is the fact that even with limited context of $\alpha = O(\log t)$, this scheme achieves the optimal compression (in terms of the conditional entropy) provided $\beta \geq \log \log t$, and in fact, this is true for all ergodic sources.[2]

Lempel-Ziv compression schemes and their variants [ZL77,ZL78,Wel84] that are popular in practice (and are used in tools such as `UNIX compress, compact, gnuzip, pkzip, winzip`, the gif image compression format and the current modem compression standard V42bis) do not achieve optimality under the refined notion of informational content (namely, conditional entropy) stated above. Hence, the $\mathcal{HYZ}$ compression scheme is more powerful.

Lempel-Ziv schemes and their variants are popular because they have (efficient) online/linear time implementations [RPE81]. However, such efficient algorithms for the HYZ compression scheme have not been known thus far. Our paper addresses the problem of efficiently implementing the (α, β)-$\mathcal{HYZ}$ compression scheme.[3]

1.2 Our Results

We present the following algorithmic results in this paper. Throughout we assume that the strings are drawn from a binary alphabet[4]

1. For the document listing problem, we present an algorithm that preprocesses the k documents in linear time (that is, $\mathrm{O}(\sum_i t_i)$) and space. The time to answer a query with pattern P is $O(|P| \log k + out)$ where *out* is the number of documents that contain P. The algorithm relies on an augmentation of the suffix tree that we call the *Suffix-DAG*.[5]

[1] See [Yok96] for the definition of the conditional entropy, its relevance in context-dependent compression schemes and details of the limited order Markovian source in the claim.

[2] See [HZ95,HZ98] for requirements on the source type and the result. In particular, this results are valid for large n but they are not asymptotic in nature – unlike, for example, [ZL77] – and hence might be of additional practical interest.

[3] There are many other context-based data compression methods in the literature [CW84,BW94,WRF95]. These have not been shown to be optimal under refined information-theoretic notion such as the conditional entropy. Experimental studies have confirmed that these context-based methods compress English text better than Lempel-Ziv variants [BCW90]. However, these context-based methods are either not very fast (they usually require time and sometimes space superlinear with the size of the input), or not online. For a survey and comparison of these methods, see [BCW90].

[4] The analyses in [HZ95,HZ98] are for binary strings.

[5] The suffix-DAG data structure also helps solve the count query in time $O(|P|)$, which matches the best known complexity [Hui92]. We present a new lemma for

Although this problem is natural, no nontrivial algorithmic results were known before our paper. The fastest algorithms for solving this problem run in time proportional to the number of occurrences of the pattern in all documents; clearly, this could be much larger than the number of documents that contain P, if P occurs multiple times in a document.

2. For the (α, β)-$\mathcal{HYZ}$ compression problem, we provide two algorithms. The first algorithm takes time $O(t\beta)$ (for compression and decompression) and works for any maximum context size α. This gives a linear time algorithm for $\beta = O(1)$, the case considered in [Yok96]. The only previously known algorithm for solving this problem is in [Yok96], where for $\beta = O(1)$, the author presents an $O(t\alpha)$ time algorithm. This algorithm is optimal only for small contexts, that is, $\alpha = O(1)$, and for unbounded α, this running time $O(t^2)$.
The second algorithm is tuned for the optimal case identified in [HZ95,HZ98]: we provide an $O(t)$ time algorithm to compress and decompress, provided that $\beta = \log\log t$, and $\alpha = O(\log t)$. It is the first efficient algorithm for this problem. Notice that for both cases, our algorithms are optimal.
The first algorithm is similar to the McCreight's method for building the suffix tree of the string online; however, we need to appropriately augment each node with at most β units of information. The second algorithm maintains a trie on all strings within the specified context size together with "suffix links" on this trie (as in a suffix tree), and additionally, auxiliary edges on each node to some of its descendants. The technical details of the latter algorithm are more involved than the former, but in both cases the crux is to design algorithms for maintaining and utilizing the augmented information and the structural properties of the suffix links are used extensively to achieve this.
The efficient algorithmic results supplement the information-theoretic properties of the HYZ schemes proved in [Yok96,HZ95,HZ98]; hence these schemes may prove to be a promising alternative to the existing ones in practice. We are currently experimenting with our algorithms.

2 The Suffix-DAG Data Structure

Consider a set of document strings $T = \{T_1, T_2, \ldots, T_k\}$, of respective sizes $t_1, t_2, \ldots, t_k$ to be preprocessed. Our goal is to build a data structure which would support the following queries on an on-line pattern P of size p: (1) *list query* – the list of the documents that contain P, and (2) *count query* – the number of documents that contain P.

Theorem 1. *Given T and P, there is a data structure which responds to a count query in $O(p)$ time, and a list query in $O(p \log k + \texttt{out})$ time, where* `out` *is the number of documents in T that contain P.*

---

computing count queries in bottom-up fashion, eliminating dependancy on the use of data structures for handling LCA queries, hence our scheme might work faster in practice.

Proof Sketch. We build a data structure , that we call the suffix-DAG of documents $T_1, \ldots, T_k$, in $O(t) = O(\Sigma t_k)$ time using $O(t)$ space. The suffix-DAG of T, denoted by $SD(T)$, contains the generalized suffix tree, $GST(T)$, of the set T at its core. A *generalized suffix tree* of a set of documents is defined to be the compact trie of all the suffixes of each of the documents in T ([Gus98]). Each leaf node l in $GST(T)$ is labeled with the list of documents which have a suffix represented by the path from the root to l in $GST(T)$. We denote the substring represented by a path from the root to any given node n by $\mathcal{P}(n)$.

The nodes of $SD(T)$ are the nodes of $GST(T)$ themselves. The edges of $SD(T)$ are of two types: (1) the *skeleton edges* of $SD(T)$ are the edges of $GST(T)$; (2) the *supportive edges* of $SD(T)$ are defined as follows: given any two nodes n_1 and n_2 in $SD(T)$, there is a pointer edge from n_1 to n_2, if and only if (i) n_1 is an ancestor of n_2, and (ii) among the suffix trees $ST(T_1), ST(T_2), \ldots, ST(T_k)$ of respective documents $T_1, T_2, \ldots, T_k$, there exists at least one, say $ST(T_i)$, which has two nodes, $n_{1,i}$ and $n_{2,i}$ such that $\mathcal{P}(n_1) = \mathcal{P}(n_{1,i})$, $\mathcal{P}(n_2) = \mathcal{P}(n_{2,i})$, and $n_{1,i}$ is the parent of $n_{2,i}$. We label such an edge with i, for all relevant documents T_i.

In order to respond to the count and list queries, we build one of the standard data structures that support least common ancestor (LCA) queries on $SD(T)$ in $O(1)$ time [SV88]. Also, for each of the internal node n of $SD(T)$, we keep an array that stores its supportive edges in pre-order fashion, and the number of documents which include $\mathcal{P}(n)$ as a substring.

The following lemmas state the complexities of the procedures for responding to list and count queries.

Lemma 1. *The suffix-DAG is sufficient to respond to the count queries in $O(p)$ time, and to list queries in $O(p \log k + out)$ time.*

Proof Sketch. The procedure for responding to count queries is as follows. With P, trace down $GST(T)$ until the highest level node n is reached for which P is a prefix of $\mathcal{P}(n)$. We simply return the number of documents that contain $\mathcal{P}(n)$ as a substring; recall that this information is stored in node n.

The procedure for responding to list queries is as follows. We locate node n (defined above) in $SD(T)$ and traverse $SD(T)$ backwards from n to the root. At each node u on the path, we determine all supportive edges out of u have their endpoints in the subtree rooted at n. The key observation is that all such edges will form a consecutive segment in the array of supportive edges residing with node u. This segment can be identified with two binary searches using an oracle that determines if a given edge has its endpoint in the subtree rooted at node n. Performing an LCA query with the endpoint of that edge and node n provides such an oracle taking $O(1)$ time. We can prove that the maximum size of the array of supportive edges attached to any node is at most $k|\Sigma|$, where $|\Sigma| = O(1)$ is the size of the alphabet of the string. Thus this procedure takes $O(\log k)$ time at each node u on the path from n to the root to identify the segment of supportive edges; the entire segment of such edges is output at each node u. The output of all such segments contains duplicates, however, we can prove that the total size of the output is $O(\texttt{out}|\Sigma|) = O(\texttt{out})$, where $\texttt{out}$ is the number of occurrences of P in t. We provide the proof of the observations above in the full version of this paper. □

Lemma 2. *The suffix-DAG of the document set T can be constructed in $O(t)$ time and $O(t)$ space.*

Proof Sketch. The construction of the generalized suffix tree $GST(T)$ with all suffix links, and a data structure to support constant time LCA queries are standard (see [CR94]). What remains is to describe for each node n of $SD(T)$, (1) how its supportive edges are constructed, (2) how its supportive edge array is built, and (3) how the number of documents that include $\mathcal{P}(n)$ is computed.

The supportive edges with, say, label i, can be built by emulating McCreight's construction for $ST(T_i)$ (the suffix tree of T_i) on $SD(T)$. Notice that for each node in $ST(T_i)$, there is a corresponding node in $SD(T)$ with the appropriate suffix link. This implies that one can iteratively construct the supportive edges by using the suffix links as follows: Suppose that at some given step of the construction, a supportive edge between two nodes n_1 and n_2 with label i has been established. Let the suffix links from n_1 and n_2 reach nodes n'_1 and n'_2 respectively. Then, either there is a supportive edge between n'_1 and n'_2, or there exists one intermediate node to which there is a supportive edge from n'_1, and there exists one intermediate node from which there is a supportive edge to n'_2. The time to compute such intermediate nodes can be charged to the number of characters in the substring represented by the path between n_1 and n_2 in T_i. We leave the details of the non-trivial construction of the supportive edge arrays to the full paper.

Given a node n of $GST(T)$, the number of documents, $\#(n)$, which contain the substring of n, can be computed as follows: The only information we use about each node n is (1) the number of supportive edges from n to its descendants, $\#\uparrow(n)$, and (2) the number of supportive edges to n from its ancestors, $\#\downarrow(n)$. □

Lemma 3. *For any node n, $\#(n) = \Sigma_{n' \in children\ of\ n} \#(n') + \#\uparrow(n) - \#\downarrow(n)$.*

Proof. The proof follows from the following key observations:

- if a document T_i includes the substrings of more than one descendant of n, then there should exist a node in $ST(T_i)$ whose substring is identical to that of n;
- given two supportive edges from n to n_1 and n_2, the path from n to n_1 and the path from n to n_2 do not have any common edges.

□

This concludes the proof of the lemma for suffix-DAG construction and hence the proof of the theorem. □

We note that using the suffix-DAG structure and additional edges between the arrays of supportive edges at endpoints of the edges, we can prove the following. There is an algorithm that preprocesses T in $O(t \log\log t)$ time and $O(t)$ space following which a list query can be answered in time $O(p + \mathtt{out})$. The details will be presented in the final version.

3 The Compression Algorithms

Consider a compression scheme with parameters α and β. We refer to it as $\mathcal{C}_{\alpha,\beta}$, and whenever α and β will be understood we may use C instead. The input to $\mathcal{C}_{\alpha,\beta}$ is a string T of t characters. In this discussion, we assume that the alphabet is binary, however our results trivially generalize to any constant size alphabet. We denote by $T[i]$, the ith character of T $(1 \leq i \leq t)$, and by $T[i:j]$ the substring of T which begins at $T[i]$ and ends at $T[j]$. The parameters α and β are (possibly constant) functions of t.

Recall the compression scheme from Section 1.1. The compression scheme $\mathcal{C}_{\alpha,\beta}$ performs compression by partitioning the input T into contiguous substrings, or blocks of β characters, and replacing each block by a corresponding codeword. To compute the codeword c_j for block j, $\mathcal{C}_{\alpha,\beta}$ first computes the *context* for block j. The context of a block $T[i:l]$ is the longest substring $T[k:i-1]$ for $k < i$, for which $T[k:l]$ occurs earlier in T. If the context size exceeds α, the context is truncated to contain only the α rightmost characters. The codeword c_j is the ordered pair $\langle \gamma, \lambda \rangle$. We denote by γ the size of the context for block j. We denote by λ the lexicographic order of block j amongst all possible substrings of size β immediately following earlier occurrences of the context of block j.[6]

Our results are as follows.

Theorem 2. *There is an algorithm to implement the compression scheme $C_{\alpha,\beta}$ which runs in $O(t\beta)$ time and requires $O(t\beta)$ space, independent of α.*

Proof. We employ McCreight's method for the construction of the suffix tree, during which we augment the suffix tree as follows. For each node v, we store an array of size β in which for each $i = 1, \ldots, \beta$, we store the number of distinct paths rooted at v (ending at a node or within an edge) of precisely i characters minus the number of such distinct paths of precisely $i-1$ characters; note that these numbers may be negative.

We now show how to efficiently construct this data structure.

Lemma 4. *There is an algorithm to construct the augmented suffix tree of T in $O(t\beta)$ time.*

[6] For instance, for $T = 010011010\ldots$ and $\alpha = 2$, $\beta = 1$, the context of block 9 (which consists of $T[9] = 0$ only) is $T[7:8] = 01$, since 010 appears earlier but 1010 does not. The two substrings which follow earlier occurrences of this context are $T[3] = 0$ and $T[6] = 1$. The lexicographic order of block 9 among these substrings is 1.

Proof. Consider the suffix tree construction due to McCreight. While inserting a new node v into the suffix tree, we update the subtree information, if necessary, of the ancestors of v which are at most β characters higher than v. The number of these ancestors whose information will need to be changed is at most β. We can argue that at most one of the β fields of information at any ancestor of v needs to be updated. That completes the proof. $\square$

We now show that the resulting date structure facilitates the implementation of the compression scheme $\mathcal{C}$ in desired time and space bounds.

Lemma 5. *The augmented suffix tree is sufficient to compute the codeword for each block of input T in amortized $O(\beta^2)$ time.*

Proof Sketch. Recall that the codeword for a block j consists of the pair $\langle\gamma, \lambda\rangle$, where γ is the context size and λ is the lexicographic order of block j among substrings following its context. The computation of γ can be performed by locating the node in the suffix tree which represents the longest prefix of the context. This can be achieved in an inductive fashion by using the suffix links of McCreight in amortized $O(\beta)$ time (details omitted).

The computation of λ can be performed as follows. We traverse the path between nodes v and w in the suffix tree. The node v represents the longest prefix of the context of block j. The node w, a descendant of v, represents the longest prefix of the substring formed by concatenating the context of block j to block j itself. During this traversal, we compute the size of the relevant subtrees representing substrings lexicographically smaller and lexicographically greater than the substring represented by this path.

We present the details of the amortization argument used in the computation of γ, and the details of subtree traversal used in the computation of λ in the full version. $\square$

This completes the proof of the theorem. $\square$

For $\beta = O(1)$ as in [Yok96], the algorithm above takes linear time independent of α.

We now turn our focus to our second algorithm for implementing the compression method $\mathcal{C}_{\alpha,\beta}$ for the optimal choice of parameters $\alpha = O(\log t)$ and $\beta = \log\log t$. In what follows, we restrict ourselves to the case of $\alpha = \log t$. Later we will describe how to extend this to larger α.

Theorem 3. *There is an algorithm to implement the compression method $\mathcal{C}_{\alpha,\beta}$ for $\alpha = \log t$ and $\beta = \log\log t$ in $O(t)$ time using $O(t)$ space.*

Proof Sketch. Our algorithm is based on a suffix tree-like data structure which supports the following feature. Consider a node v in the suffix tree and the set of descendants of v which are β characters apart from v. Our data structure enables one to compute the lexicographic order of the path between any such descendant and the node v. Once the lexicographic order of a given node w is known, the codeword of the block represented by the path between v and w of size β is easily computed.

Our algorithm exploits the fact that the context size is bounded, and it seeks similarities between suffixes of the input up to a small size. For this purpose, we build the *trie* of the $\log t$-sized prefixes of all suffixes of the input T. We note that this data structure is similar to a suffix tree except that (i) every edge contains exactly one character, and (ii) for any pair of suffixes, it represents common prefixes of size at most $\log t$. We will refer this data structure as the *limited suffix trie* of input T. In order to support the lexicographic order queries we augment the limited suffix trie by keeping a search data structure for each node v to maintain is descendants which are β characters away from v.

The lemma below states that one can compute the codeword of any input block in optimal $O(\beta)$ time with our data structure.

Lemma 6. *The augmented limited suffix trie of input T is sufficient to compute the codeword for any input block j in $O(\beta)$ time.*

Proof Sketch. Given a block j of the input, suppose that the node v represents its context and that among the descendants of v, the node w represents the substring obtained by concatenating the context of block j and block j itself. Because $\beta = \log\log t$, the maximum number of elements in the search data structure for v is $2^\beta = O(\log t)$. There is a simple data structure that maintains k elements and computes the rank of any given element in $O(\log k)$ time; hence, one can compute the lexicographic order of node w in only $O(\log\log t) = O(\beta)$ time. We leave the details of the proof to the full paper. □

Lemma 7. *The augmented limited suffix trie of input T can be built and maintained in $O(t)$ time and space.*

Proof Sketch. The depth of our augmented limited suffix trie is bounded by $\log t$, hence the total number of nodes in the trie is only $O(t)$. This suggests that one can adapt McCreight's suffix tree construction in $O(t)$ time - without being penalized for building a suffix trie rather than a suffix tree.

To complete the proof what we need to do is to show that it is possible to construct and maintain the search data structures of all nodes in $O(t)$ time. This follows from the fact that each node v in our data structure is inserted to the search data structure of at most one of its ancestors. Therefore, the total number of elements maintained by all search data structures is $O(t)$. The insertion time of an element e to a search data structure, provided that the element e' in the data structure which is closest to e in rank, is $O(1)$. As the total number of nodes to be inserted in the data structure is bounded by $O(t)$, one can show that the total time for insertion of nodes in the search data structures is $O(t)$.

We leave the details of how the limited suffix trie is built and how the search data structures are constructed to the full paper. □

This completes the proof of the theorem. □

We use several new ideas to extend the result above to $\alpha = O(\log t)$, or more generally, to the arbitrary α case. These include encoding all possible β length paths into a constant number of machine words of size $\log t$, and performing bit-wise operations on these words. The final step involves showing how a table indexed by all machine words can be built which will replace bit operations on machine words by mere table lookups. The end result is an $O(t)$ time and space compression and uncompression algorithm for general α and $\beta = \log\log t$.

4 Acknowledgements

We thank an anonymous referee for very fruitful suggestions.

References

[BCW90] T. Bell, T. Cleary, and I. Witten. *Text Compression*. Academic Press, 1990.

[Bro98] G. S. Brodal. Finger search trees with constant insertion time. In *ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[BW94] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, DEC SRC, 1994.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford Press, 1994.

[CW84] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[Gus98] D. M. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Addison Wesley, 1998.

[Hui92] J. Hui. Color set size problem with applications to string matching. In *Combinatorial Pattern Matching*, 1992.

[HZ95] Y. Hershkovits and J. Ziv. On sliding window universal data compression with limited memory. In *Information Theory symposium*, pages 17–22, September 1995.

[HZ98] Y. Hershkovits and J. Ziv. On sliding window universal data compression with limited memory. *IEEE Trans. on Information Theory*, 44:66–78, January 1998.

[McC76] E. M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[RPE81] M. Rodeh, V. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, January 1981.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors:simplification and parallelization. *SIAM Journal of Computing*, 17:1253–1262, 1988.

[Wel84] T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, pages 8–19, January 1984.

[WRF95] M. J. Weinberger, J. J. Rissanen, and M. Feder. A universal finite memory source. *IEEE Transactions on Information Theory*, 41(3):643–652, 1995.

[Yok96] H. Yokoo. An adaptive data compression method based on context sorting. In *IEEE Data Compression Conference*, 1996.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.

A Appendix

In this section we demonstrate the construction of the suffix-DAG of a set of documents by means of figures.

T: { T1: abcb, T2: abca, T3: abab }

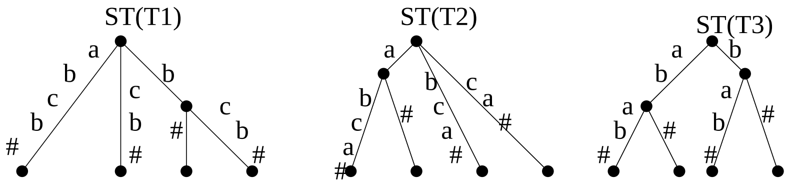

Fig. 1. The independent suffix trees of a given set of documents.

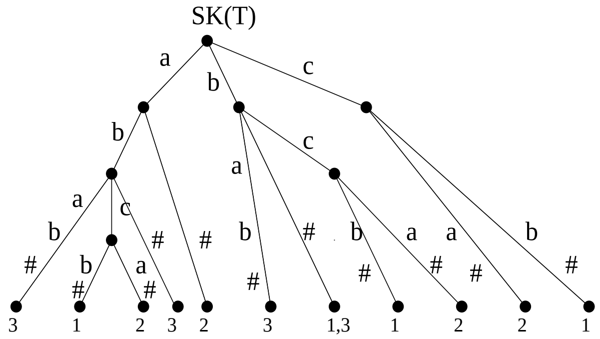

Fig. 2. The generalized suffix tree of the set of documents in Figure 1.

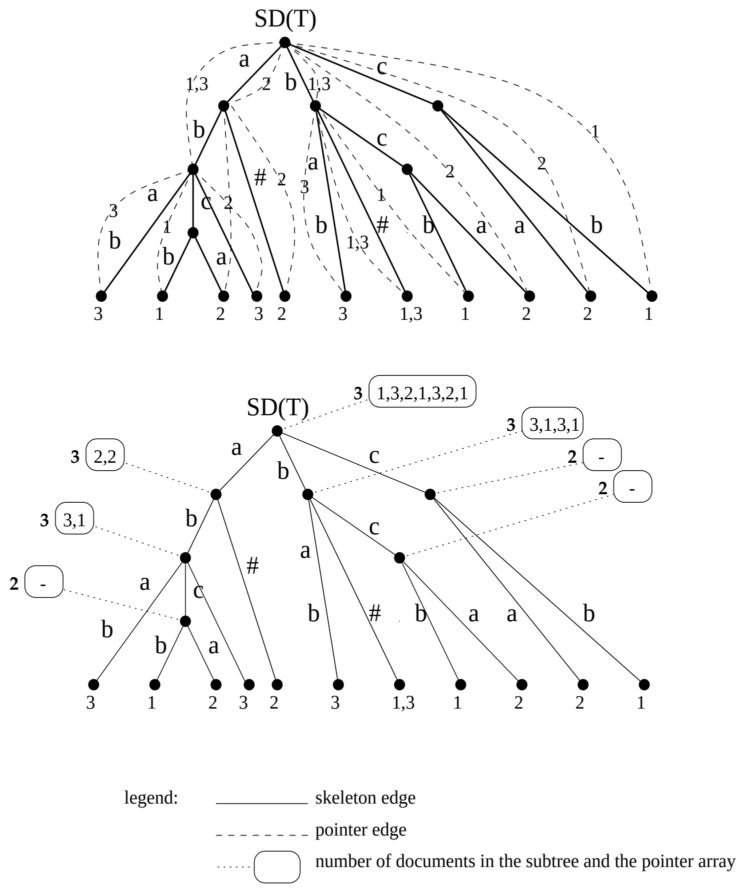

Fig. 3. The suffix-DAG of the set of documents in Figure 1.