Online Time Stamped Text Indexing*

Amihood Amir† Bar-Ilan University and Georgia Tech

Gad M. Landau‡ Haifa University and Polytechnic University

Esko Ukkonen§ University of Helsinki

Abstract

In this paper we present an efficient method to index a text stream on-line in a fashion that allows, at any point in time, to find the longest suffix of the text that has appeared previously, and the closest (farthest) times in which it has appeared. Our algorithms allow efficient answers to these queries.

1 Introduction

Many pattern recognition tasks are solved by ad-hoc heuristics that try to exploit special knowledge of domain properties with varying degrees of success. Recently, there have been attempts to solve a class of pattern recognition problems by analysis of repetitions and periods in a sequence of images taken over time [10]. These methods are more general and not domain-specific.

Johansen [10] used this idea for surveillance, character labeling, and discrimination of handwriting and texture. In the surveillance application, the goal is finding "unexpected" changes in a sequence of photographs, when there is no prior definition of "unexpected". The "surprising" nature of an image is to be detected from the syntax of the sequence of images. Johansen uses exact matching in his applications.

Thus, the assumption is that there are some repetitions in the behavior of the sequence of images. At any point in time, there should be some recent subsequence that is equal to the one ending now. Our task is to quickly find the longest and closest such subsequence.

*This research was conducted while the first two authors were visiting the University of Helsinki as part of an Israel-Finland exchange scientist grant funded by the Israel Ministry of Science and CIMO in Finland.

†Department of Mathematics and Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8770, amir@cs.biu.ac.il. Partially supported by NSF grant CCR-96-10170, BSF grant 96-00509, and an Israel-Finland exchange scientist grant.

‡Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840; email: landau@poly.edu; partially supported by NSF grants CCR-9610238 and CCR-0104307, by NATO Science Programme grant PST.CLG.977017, by an Israel-Finland exchange scientist grant, by the Israel Science Foundation (grants 173/98 and 282/01), by the FIRST Foundation of the Israel Academy of Science and Humanities, and by IBM Faculty Partnership Award.

§Department of Computer Science, P.O. Box 26, FIN-00014 University of Helsinki, Finland; ukkonen@cs.helsinki.fi; supported by the Academy of Finland under grant 22584.

In this paper, we abstract the exact matching version of this problem to its combinatorial essence and give efficient solutions. While finding the longest subsequence that occurred earlier as well as its *earliest* occurrence can be easily done using, say, the suffix tree construction of [14], finding the latest occurrence, or the x first or latest occurrences is trickier. Weiner's suffix tree construction algorithm [15], when applied on the reversal of the string, turns out to be a suitable 'on–line' method that will serve as the basis of our developments.

2 Problem Definition and Preliminaries

2.1 Problem Definition

Definition 1 *The* online time stamped text indexing problem *is defined as follows.*

*INPUT: An incoming stream (*text*) of symbols from alphabet Σ.*

QUERIES: Assume at point i in time, the stream $s_1, s_2, ..., s_i$ has been input. We want to:

1. *Find the longest suffix $s_{i-k}, s_{i-k+1}, ..., s_{i-1}, s_i$ that occurred previously in the text (*indexing*).*
2. *Find the latest occurrence of this longest suffix, i.e. the largest index ℓ, $\ell < i$ for which $s_{i-k}, s_{i-k+1}, ..., s_{i-1}, s_i = s_{\ell-k}, s_{\ell-k+1}, ..., s_{\ell-1}, s_\ell$ (*latest occurrence*).*
3. *Find the earliest occurrence of this longest suffix, i.e. the smallest index ℓ, $\ell < i$ for which $s_{i-k}, s_{i-k+1}, ..., s_{i-1}, s_i = s_{\ell-k}, s_{\ell-k+1}, ..., s_{\ell-1}, s_\ell$ (*earliest occurrence*).*
4. *Given x, find the last x occurrences of this longest suffix, i.e. the largest x indices $\ell_1, ..., \ell_x$, $i > \ell_x > \ell_{x-1} > \cdots > \ell_1 \geq 1$ for which $s_{i-k}, s_{i-k+1}, ..., s_{i-1}, s_i = s_{\ell_r-k}, s_{\ell_r-k+1}, ..., s_{\ell_r-1}, s_{\ell_r}$, $r = 1, ..., x$. If there do not exist x different such indices, then return the largest x_1, $x_1 \leq x$ indices where the suffix equality holds (*latest x occurrences*).*
5. *Given x, find the first x occurrences of this longest suffix, i.e. the smallest x indices $\ell_1, ..., \ell_x$, $i > \ell_x > \ell_{x-1} > \cdots > \ell_1 \geq 1$ for which $s_{i-k}, s_{i-k+1}, ..., s_{i-1}, s_i = s_{\ell_r-k}, s_{\ell_r-k+1}, ..., s_{\ell_r-1}, s_{\ell_r}$, $r = 1, ..., x$. If there do not exist x different such indices, then return the largest x_1, $x_1 \leq x$ indices where the suffix equality holds (*earliest x occurrences*).*

The difficulty with the online time stamped text indexing problem lies with the *time stamping* requirement. The *text indexing* requirement is well studied and can be easily solved.

2.2 The Indexing Problem

The *indexing* problem assumes a (usually very large) text that is to be preprocessed in a fashion that will allow efficient future queries of the following type. A query is a (significantly shorter) string of symbols from Σ which we call a pattern. One wants to find all text locations that match the pattern in time proportional to the pattern length and number of occurrences.

Weiner [15] invented the *suffix tree* data structure whereby the text is preprocessed in linear time, and subsequent queries of length m get answered in time $O(m + tocc)$, where *tocc* is the number of pattern occurrences in the text. The times above are to be multiplied by $\log m$ in case the alphabet size is unbounded.

Weiner's suffix tree in effect solved the indexing problem for exact matching of fixed texts. Succeedingly improved algorithms for the indexing problem in *dynamic* texts were suggested, for example by [7, 12]. For the sake of completeness we review below how Weiner's suffix tree can solve the indexing problem.

Definition 2 *A trie T for a set of strings $\{S_1, \cdots, S_r\}$ is a rooted directed tree satisfying:*

1. *Each edge is labeled with a character, and no two edges emanating from the same node have the same label.*
2. *Each node v is associated with a string, denoted by $L(v)$, obtained by concatenating the labels on the path from the root to v, $L(root)$ is the empty string.*
3. *There is a node v in T if and only if $L(v)$ is a prefix of some string S_j in the set.*

A *compacted trie* T' is obtained from T by collapsing paths of internal nodes with only one child into a single edge and by concatenating the labels of the edges along the path to form the label of the new edge. The label of an edge in T' is a nonempty substring of some S_j, and it can be succinctly encoded by the starting and ending positions of an occurrence of the substring. The number of nodes of a compacted trie is $O(r)$.

Let $S[1, n] = s_1 s_2 \cdots s_{n-1}\$$ be a string, where the special character \$ is not in Σ. The *suffix tree* T_S of S is a compacted trie for all suffixes of S. Since \$ is not in the alphabet, all suffixes of S are distinct and each suffix is associated with a leaf of T_S. There are several papers that describe linear time algorithms for building suffix trees, e.g. [15, 11, 4, 14, 6].

Fact 1 *Let S_T be the suffix tree of text T. Let $P = p_1 \cdots p_m$ be a pattern. Start at the suffix tree root and follow the labels on the tree as long as they match $p_1 \cdots p_m$. If at some point there is no matching label, then P does not appear in T. Otherwise, let v be the closest node (from below) to the label where we stopped. The starting locations of the suffixes that correspond to the leaves in the subtree rooted at v are precisely all the text locations where the pattern appears. These locations can be located and listed, for a fixed bounded alphabet, in time $O(m + tocc)$.*

Implementation Remark 1 *From now on, Weiner's algorithm will be used in a novel way to to build a variant of the suffix tree. Given a string $S[1, n] = s_1 \ldots s_{n-1}\$$, Weiner's algorithm in its standard form computes in an incremental fashion n suffix trees of the substrings $S[i, n]$, $i = n, n-1, \ldots 1$. The suffix tree for $S[i, n]$ is obtained by inserting string $S[i, n]$ to the tree for $S[i+1, n]$. The algorithm moves on the string from right to left. In this paper the input for Weiner's algorithm is instead of $S[1, n]$ the* reversal *of it - the string $S[n, 1] = s_{n-1} \ldots s_1\$$. Now Weiner's algorithm in effect traverses the original string in the natural left-to-right order. The resulting tree represents, at time point i, the reversals of the prefixes of $S[1, i]$. This structure serves as a full text index for $S[1, i]$ like the standard suffix tree; one has only to reverse also the pattern P when making a query.*

Recall that we are interested, at every point i, in the longest suffix $s_{i-k} \dots s_i$ of the stream that has occurred previously. Weiner's construction is suited to our needs since it constructs the suffix tree (compacted trie of the reversed prefixes) online as new symbols are coming in, which is precisely our model.

In addition, by Weiner's construction, the largest suffix we are seeking is no other than the one ending at the parent w of the latest leaf v_i added to the tree. This is because the parent node w is the internal node that is closest to the leaf v_i and therefore, $L(w)$ is the longest prefix of $L(v_i) = s_i \dots s_1\$$ that occurred previously. Hence the (reversal of) string $L(w) = s_i \dots s_{i-k}$ is the longest substring ending at s_i that occurred previously. The locations of $L(w)$ can be found from the leaves of the subtree rooted at w.

Time for computing the longest suffix: Using the $O(n)$ time Weiner's suffix tree algorithm we can compute, at every step i, the answer for query 1 (the longest suffix) for $s_1 \dots s_i$ in $O(1)$ time.

In the remainder of this paper we show how Weiner's suffix tree construction can be extended to enable timestamping which allows us to solve fast the remaining queries as well. We will first concentrate on finding, for every node v in a suffix tree, the latest (earliest) leaf to be added to v's subtree. We then show how our method extends to enable finding the earliest and latest x occurrences.

It should be noted that query type 3 (earliest occurrence) on its own is simple to solve. For every node v, mark the oldest leaf in the subtree of v (see Section 3.3). This leaf would give the earliest occurrence of the substring ending at v. However, this idea would not suffice for finding the earliest x occurrences (type 5 queries). Thus we will actually treat in an equal (and symmetric) way queries of types 2 and 3, latest and earliest occurrences.

Also note that query type 3 is possible to solve using the other well–known suffix tree algorithms [11, 14] without the need to work with reversed string. However, they are not suitable as such for solving the other queries: McCreight's method [11] is not strictly on–line as it has to look beyond the current symbol s_i, and Ukkonen's method [14] uses implicit presentation of the intermediate trees which should be made explicit in the present application.

3 Time Stamped Suffix Trees

As each new leaf is inserted to the suffix tree, label it by a number from $\{1, \dots, n\}$, depicting the order of its insertion in the tree. Call this number the leaf's *timestamp*. Maintain a linked list of all the leaves of the tree from left to right. The order in this list is determined just by the (ordered) branching structure of the tree. At each node v, we maintain in $O(1)$ time the pointers to the leftmost leaf and to the rightmost leaf in the subtree rooted at v, in a growing tree. That these leaves do not change while the tree grows, is a crucial requirement whose implementation will be given in Section 3.3.

At point i of the suffix tree construction, a new leaf v_i, such that $L(v_i) = s_1 s_2 \dots s_i$, is inserted into the tree. (In our implementation $L(v_i)$ is actually the reversed string $s_i s_{i-1} \dots s_1\$$.)

The pointers to the leftmost and rightmost leaves of the subtree rooted at v_i's parent w are precisely the pointers to the linked list of leaves that designate the interval from which we need to extract the smallest (largest) time stamp value to solve the queries of types 2 and 3.

This extraction problem is known in the literature as the *range minimum (maximum) problem.* In this problem a given array is preprocessed in a manner that subsequent queries of the form $[i, j]$ are answered with the first index ℓ, $i \leq \ell \leq j$ where the array's value in location ℓ is the smallest (largest) value in the range $[i, j]$.

Gabow, Bentley and Tarjan [8] showed an algorithm that solves this problem by using Cartesian trees and Lowest Common Ancestor (LCA) queries.

Definition 3 *Let $H = h_1, ..., h_n$ be a list of n numbers.*

A Cartesian Tree *of H is a rooted binary tree defined recursively as follows:*

Let $h_{root} = \min\{h_1, ..., h_n\}$ (for the range maximum problem take $h_{root} = \max\{h_1, ..., h_n\}$). Then,

1. *h_{root} is the* root *of the Cartesian tree*
2. *the* left child *of the root is the Cartesian tree of $h_1, ..., h_{root-1}$*
3. *the* right child *of the root is the Cartesian tree of $h_{root+1}, ..., h_n$.*

In [8] it was shown that, following a linear time preprocessing of H, the Cartesian tree of H can be constructed in time $O(n)$.

Definition 4 *Let S be an n node tree, v, w nodes in S. The* lowest common ancestor *of v and w ($LCA(v, w)$) is the node x such that x is an ancestor of v and w, and every ancestor y of both v and w is either equal to x or an ancestor of x.*

Harel and Tarjan [9], and afterwards other authors (e.g. [13, 3, 2]) showed methods of preprocessing the tree S in time $O(n)$ such that subsequently LCA queries can be answered in constant time.

Observation 1 *Gabow, Bentley and Tarjan [8] observed that in a Cartesian tree, $LCA(x, y)$ is the node with the smallest value between nodes x and y. By [9] the LCA can be found in constant time.*

We need to show how to solve the online range maximum/minimum query problem in a growing list of timestamped leaves.

3.1 Computing the latest occurrence

Using balanced binary search trees one can solve the dynamic range maximum/minimum problem in time $O(\log i)$ per range maximum or minimum query, where there are i entries in the list (see e.g. [16]) . However, it was demonstrated [1, 17] that the range minimum problem on a changing list can not be solved in the cell probe model with amortized constant time per query.

This seems to bode ill for our time stamped indexing problem. Indeed, for the latest occurrence query we do not know how to answer a query in linear time.

Time for computing the latest occurrence at time point i: A range maximum query can be done in $O(\log i)$ time.

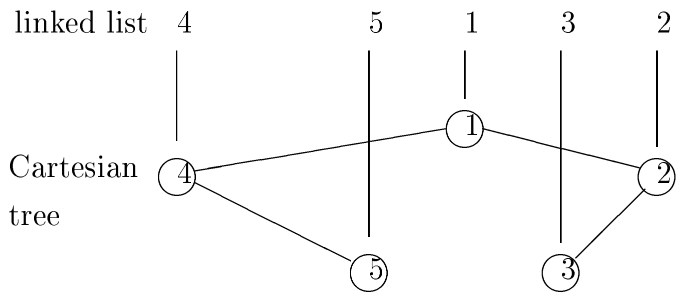

Figure 1: The linked list of leaves 1,...,5 and their Cartesian tree.

3.2 Computing the earliest occurrence

As regards the earliest occurrence queries, we can benefit from the fact that the inputs to our lists are not general. New leaves are inserted in the increasing order of the time stamp values. We will now show that it is possible to perform range *minimum* queries in constant time for a growing list where the data values are *increasing*. This optimally answers the earliest occurrence case.

Our solution involves the following two operations:

1. Construct Cartesian tree online.
2. Enable efficient LCA queries in a dynamically changing tree.

3.2.1 Online Cartesian Tree Construction

Our problem is the following: We are constructing a permutation of the numbers $\{1, ..., n\}$. The numbers are being input in ascending order, from 1 to n. When the $i+1$st number is input, there is a permutation of the numbers $\{1, ..., i\}$, as well as a Cartesian tree of those numbers (appropriately double-linked). The number $i+1$ is inserted at some position in this list, and it is to be inserted correspondingly at the Cartesian tree. For an example see Figure 1 where $i = 5$.

Insertion into Cartesian Tree in Constant Time: Assume the list and Cartesian tree of $\{1, ..., i\}$ have been constructed and $i+1$ is inserted in the list between numbers x and y. Assume that $x < y$ and y is to the right (left) of x in the list. Then

Add $i+1$ as a left (right) son of y.

See Figure 2 for an example.

The following lemma assures us that we can add $i+1$ to the Cartesian tree in the manner proposed above.

Lemma 1 *Let x and y be two adjacent numbers in a linked list, and let $x < y$. Then y does not have a left son.*

Proof: Since $x < y$ it can not be a left son of y, by definition of Cartesian tree. The Cartesian tree definition also assures us that there is no other left son of y, otherwise that son had to be between x and y. ∎

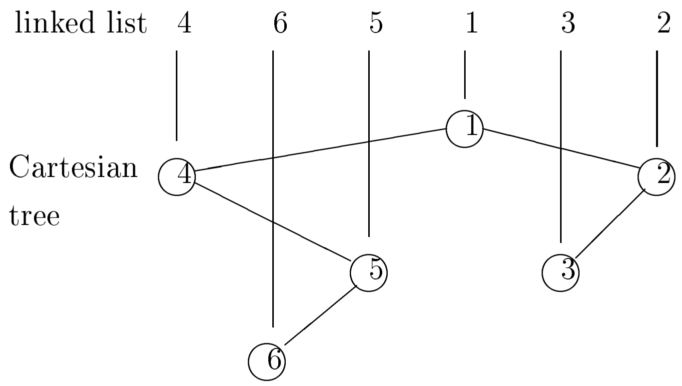

Figure 2: Adding 6 to the Cartesian tree.

3.2.2 Dynamic LCA

Cole and Hariharan [5] show how to compute the LCA in constant time in a tree where leaves and nodes are being added.

3.2.3 Earliest Occurrence in Constant Time

The results of Sections 3.2.1, 3.2.2, and 3.3 together give a constant time solution to query type 3:

Time for computing the earliest occurrence: A range minimum query can be done in $O(1)$ time.

3.3 Leftmost and Rightmost Leaves Pointers

The only remaining problem is how to know in constant time who is the leftmost leaf and who is the rightmost leaf of the subtree rooted at a given node, while the tree is constantly growing (because of the Weiner construction). The problem is caused by the fact that the left-to-right order of the children of a given node is not determined by their timestamp but rather by the lexicographic order of the alphabet symbol on each child.

The solution is surprisingly simple. While the lexicographic order is necessary for constructing the tree efficiently, the time will not deteriorate asymptotically if a constant number of children of every node need to be checked especially. We therefore fix the right and left children of every node, and all additions will be *between these fixed children*. The additions will follow the lexicographic order of the alphabet.

The rules for adding left and right children are as follows:

1. The left and right pointers of a leaf are pointers to the leaf itself.
2. The first son added to a leaf is always its right son.
3. Suppose an edge (w, x) is changed to the two edges (w, v) and (v, x), with a new leaf y added as a left child of v (i.e. add edge (v, y)). See figure 3. The new edge will be to the left *unless*

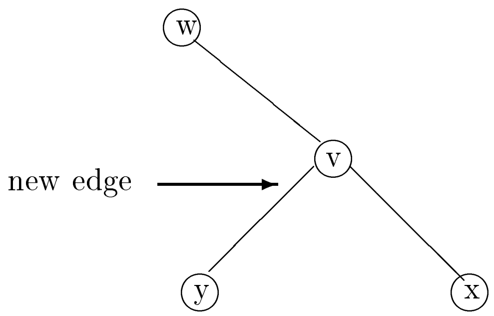

Figure 3: Adding a new edge to the middle of an existing one.

x is the leftmost child of w. Therefore, the left and right pointers of w as well as those of x remain unchanged. v's right pointer is x's right pointer. v's left pointer is y. If x is the leftmost child of w then y is added as the right son of v. All pointers remain unchanged. v's leftmost son pointer is x's leftmost son pointer, and v's rightmost son pointer points to y.

4 Latest and Earliest x Occurrences

For the sake of alleviating awkwardness, we will consider only the latest occurrence throughout most of the ensuing discussion. The earliest occurrence case is entirely symmetric.

Let $S = s_1, ..., s_n$ be a list of numbers. Denote by $M(p,q)$ the first index x $(p < x < q)$ such that $s_x \geq s_\ell, \quad \ell = p+1, ..., q-1$. In other words, $M(p,q)$ is the index of the first occurrence of the maximum value between indices p and q.

Observation 2 *Let $s_{\ell_1}, ..., s_{\ell_x}$ be the x largest numbers in the range $[p,q]$ and assume $\ell_1 < \ell_2 < \cdots < \ell_x$. Then the $x+1$ largest number in range $[p,q]$ is*

$$\max\{M(p-1,\ell_1), M(\ell_1,\ell_2), M(\ell_2,\ell_3), ..., M(\ell_{x-1},\ell_x), M(\ell_x, q+1)\}$$

Note that even if a range maximum query can be done in constant time, computing the x largest numbers naively using the above observation will take time $O(x^2)$. However, this can be avoided by keeping the $x+1$ extra elements $\{M(p-1,\ell_1), M(\ell_1,\ell_2), M(\ell_2,\ell_3), ..., M(\ell_{x-1},\ell_x), M(\ell_x, q+1)\}$ (and their ranges) in a priority queue. Generating $s_{\ell_{x+1}}$ is now simply a matter of choosing the top of the priority queue (maximum). Assume k is the index such that $\ell_{x+1} = M(\ell_k, \ell_{k+1})$, i.e. $M(\ell_k, \ell_{k+1})$ is the largest of $\{M(p-1,\ell_1), M(\ell_1,\ell_2), M(\ell_2,\ell_3), ..., M(\ell_{x-1},\ell_x), M(\ell_x, q+1)\}$. Then delete $s_{\ell_{x+1}}$ from the priority queue and insert two new elements, $M(\ell_k, \ell_{x+1})$ and $M(\ell_{x+1}, \ell_{k+1})$. We have a constant number of range maximum queries and a constant number of priority queue operations for every additional number.

Time: Let t be the time for a range maximum query. Since we do $O(x)$ queries our total query time is $O(xt)$. A priority queue can be implemented as a heap so that insertions and deletions can be done in time $O(\log x)$ each, where x is the heap size. Thus our total priority queue handling time is $O(x \log x)$.

Time for computing x last occurrences of longest suffix at time point i: A range maximum query requires time $O(\log i)$, where i is the current length of the string. Thus the time is $O(x \log i)$.

Time for computing x earliest occurrences of longest suffix at time point i: A range minimum query can be done in constant time. Thus the time is $O(x \log x)$.

5 Open Problems

The dynamic range minimum problem can not be solved with a constant query time. However, we have presented an algorithm that answers such queries in time $O(1)$ for the special dynamic case in which insertions in the increasing order of the values are allowed. We were not able to find such algorithm for the range maximum problem. Such an algorithm would be interesting and useful for our application.

Acknowledgments

The authors wish to thank Mike Fredman for pointing out to us citations [1, 17].

References

[1] M. Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8:235–247, 1988.

[2] O. Berkman, and U. Vishkin. Recursive star-tree parallel data-structure. *SIAM J. Comp.*, 22(2):221–242, 1993.

[3] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *Proc. 6th Annual ACM-SIAM Symposium On Discrete Algorithms*, pages 263–272, 1995.

[4] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, chapter 12, pages 97–107. NATO ASI Series F: Computer and System Sciences, 1985.

[5] R. Cole and R. Hariharan. Dynamic lca queries in trees. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 235–244, 1999.

[6] M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[7] P. Ferragina and R. Grossi. Optimal on-line search and sublinear time update in string matching. *SIAM J. Comp.*, 27(3):713–736, 1998.

[8] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing*, 67:135–143, 1984.

[9] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestor. *Computer and System Science*, 13:338–355, 1984.

[10] P. Johansen. Adaptive pattern recognition. *Journal of Mathematical Imaging and Vision*, 7:325–339, 1997.

[11] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.

[12] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. *Proc. 37th FOCS*, pages 320–328, 1996.

[13] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comp.*, 17:1253–1262, 1988.

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

[15] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[16] D. Willard and G. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32:597–617, 1985.

[17] B. Xiao. *New Bounds in Cell Probe Model.* PhD thesis, U.C. San Diego, La Jolla, California, 1992.