DIMACS Technical Report 93-45
August 1993

An Efficient Algorithm for Dynamic Text Indexing

by

Ming Gu[1]
Dept. of Computer Science
Yale University
New Haven, CT 06520
and Dept. of Mathematics
UC Berkeley

Martin Farach[2,3]
Dept. of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

Richard Beigel[4]
Dept. of Computer Science
Yale University
New Haven, CT 06520-2158

[1]The work was supported by a Yale University Fellowship and NSF Grants CCR-8958528 and CCR-8808949
[2]Permanent Member
[3]Supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center under NSF contract STC-8809648.
[4]This work was supported by NSF Grants CCR-8958528 and CCR-8808949.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

Text indexing is one of the fundamental problems of string matching. Indeed, the *suffix tree*, the central data structure of string matching, was developed as an efficient static text indexer. The text indexing problem is that of building a data structure on a text which allows the occurrences of patterns to be quickly looked up.

All previous text indexing schemes have been *static* in the sense that if the text is modified, the data structure must be rebuilt from scratch. In this paper, we present a first *dynamic* data structure and algorithms for the *On-line Dynamic Text Indexing* problem. Our algorithms are based on a novel data structure, the *border tree*, which exploits string periodicities.

1 Introduction

Pattern matching is one of the most well-studied fields in computer science. Problems in this field have very broad applications in many areas of computer science. Elegant and efficient algorithms have been developed for exact pattern matching. (e.g. [4, 9]).

One of the central problems of pattern matching is that of *text indexing*. In a static text indexing scheme, a fixed text string is preprocessed so that on-line queries about pattern occurrences can be quickly answered. The classical solution to the text indexing problem is the *suffix tree*. A suffix tree is a compressed trie of all suffixes of a string, and it has a linear time construction, for constant alphabet size [11, 5, 12]. Given the suffix tree, T_s^S, of a text string S, and a pattern P, it is possible to find all occurrences of P in S in $O(|P| + tocc)$, where $tocc$ is the total number occurrences of P in S.

Based on the success of the suffix tree, several other indexing problems have been tackled with similar approaches. The *suffix array*, a space economical alternative to the suffix tree, was proposed by Manber and Myers [10]. In [7], Giancarlo introduced the L-suffix tree which he used to solve the static text indexing problem on two dimensional arrays. In [3], Baker introduced the P-suffix tree to solve, amongst other things, the *parameterized text indexing problem*, that is, the problem of finding occurrences of patterns in text, even when global substitutions have modified occurrences of the pattern (see e.g. the emacs command `query-replace`).

As noted above, these solutions all assume that the text is static. If the text changes, we could attempt to report all matches in the old text and modify our output according to the edit operation that has been performed on the text. Consider, however, the following example. Let $S = a^{2n-2}$ and $P = a^n$. We insert a character b in the nth position of S to get $S' = a^{n-1}ba^{n-1}$. There are n matches of P in S, but there is no match in S'. In this example, our naïve approach takes $O(n)$ time after only one edit operation. We would have done just as well to run any linear time string matching algorithm to find P in S', especially since our naïve algorithm says nothing about occurrences that were introduce by edit operations. Of course, any algorithm for dynamic text indexing must depend on the type of edit operations allowed.

We define the *On-line Dynamic Text Indexing Problem* to be the problem of preprocessing an initial text $T = T^{(1)}$, followed by a sequence of operations of the following type which must be completed on-line, that is, before the next operation is examined.

Insert c at k: Insert character c after text location k of $T^{(i)}$ to give $T^{(i+1)}$.

Delete at k: Delete character at text location k of $T^{(i)}$ to give $T^{(i+1)}$.

Match P: Find all occurrences of P in $T^{(i)}$.

Applying known techniques, we can solve this problem in time:

Preprocessing: $O(|T|)$

Edit operations: $O(|T^{(i)}|)$

Matching: $O(|P| + tocc)$

We present a solution which trades off matching time and edit time. In particular, we present a solution which runs in time:

Preprocessing: $O(|T|)$

Edit operations: $O(\log |T^{(i)}|)$

Matching: $O(|P| + tocc \log i + i \log |P|)$

In other words, while the edit operations take only logarithmic time (if the characters of a text are store in a linked list, we could not hope to do better), the matching time is now dependent on the *fragmentation* of the text. Such a trade-off seems well suited to applications where edit operations must be completed quickly, and where the text can be defragmented occasionally, say during a garbage collection routine. In fact, this is not an uncommon setup in text editors – see e.g. Gnu Emacs.

The main contributions of this paper are:

- We introduce the area of dynamic text indexing, which is an important generalization of static text indexing.
- In order to achieve our complexities, we introduce a new and interesting data structure, the *border tree*. The border tree, like the suffix tree, is a tree on substrings of a string. We will show that the border tree is a generally useful tool, and that it has two advantages over suffix trees: it can be constructed in linear time *independent of the alphabet size*, and its depth is no more than logarithmic in the string size.

In Section 2, we give some introductory material, and present an outline of our algorithm. After the algorithm is outlined, we will break down the steps of the algorithm into a few subproblems, and then give an organizational outline of the paper.

2 Preliminaries

2.1 Notation

We begin with some general notation that we will use throughout the paper. More notation will be introduced as we use it.

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots\}$ be an ordered set, with $\$, \# \notin \Sigma$. Then $S \in \Sigma^*$ is a *string*, and if $S = s_1, s_2, \ldots, s_n$ is a string, then $S^R = s_n, s_{n-1}, \ldots, s_1$ is its *reverse string*.

Let $S = s_1, s_2, \ldots, s_n$ be a string over Σ. Then $S[i:j] = s_i, \ldots, s_j$, for $1 \leq i, j \leq n$. Note that if $i > j$, we set $S[i:j]$ to be the empty string λ. Then $S[i:n]$ is the ith suffix of S and $S[1:i]$, the ith prefix.

For any node n in T, we let $p^T(n)$ be the parent of n in T, $E^T(n)$ be the edge connecting n to $p^T(n)$, with the superscript T in the above notations will being dropped when there is no ambiguity. We set $T(n)$ to be the subtree rooted at n.

2.2 Suffix Tree

Let $S = s_1 s_2 \ldots s_m \in \Sigma^*$. As noted above, a suffix tree, T_s^S, of S is a compressed trie of all the suffixes of $S\$$, $\$ \notin \Sigma$. Similarly, a *prefix tree* T_p^S of S is a compressed trie of all prefixes of $\$S$, $\$ \notin \Sigma$. The rôle of the \$ in the definition of the suffix (prefix) trees is to insure that each suffix (prefix) is unique, and thus corresponds to a unique leaf in the tree.

We restrict the following comments to suffix trees. The following facts and definitions regarding suffix trees can be symmetrically applied to prefix trees.

For any edge $e \in T_s^S$, we define $\mathcal{L}(e)$ to be the edge label on e. We set $\mathcal{L}(r(T_s^S)) = \lambda$. For any node $n \neq r(T_s^S)$, we set $\mathcal{L}(n) = \mathcal{L}(p(n))\mathcal{L}(E(n))$.

One of the most useful properties about suffix trees is that, for node suffixes $S[i:n]$ and $S[j:n]$ with corresponding suffix tree leaves l_i and l_j, $\mathcal{L}(lca(l_i, l_j)) = lcp(S[i:n], S[j:n])$, where the *lcp* of two strings is their longest common prefix, and the *lca* of two nodes is their least common ancestor.

2.3 Main Algorithm

Define a maximal non-empty subrange of the text in which no edit operations have been performed to be a *chunk*. By a *segment*, we will mean either a chunk, or an inserted character. Finally, let the *segment* decomposition of a text $T^{(i)}$ be the sequence $W_1 \ldots W_s$, such that $T^{(i)} = W_1 \ldots W_s$, and such that each W_i is a segment.

Then any occurrence of a pattern P occurs within some segment W_i, or in some concatenation of segments $W_i \ldots W_j$. We call such matches *segment* and *block* matches, respectively. We need the following notation to outline our searching algorithm. The *representing substring*, $Rss(S)$, of S is the maximum substring of P which is also a suffix of S. The *representing prefix*, $Rp(S)$, is the maximum prefix of P which is also a *suffix* of S. The *representing suffix*, $Rs(S)$, is the maximum suffix of P which is also a *prefix* of S.

Our general scheme will be to find all occurrences within a segment, and to find information about partial occurrences overlapping segment borders. The partial information from adjacent segments can then be combined to detect block matches.

We can find all block and segment occurrences as follows:

Algorithm A: Pattern Search — Algorithm to find pattern P in $T^{(i)}$.

A.1. Preprocess P and decompose $T^{(i)}$ into segments $T^{(i)} = W_1 \ldots W_i$;

A.2. Find all segment matches - we will find them all as a batch. See §5;

A.3. $W_0 = \lambda, Rp(W_0) = \lambda$;

A.4. For each $j = 1, \ldots, i$, do

A.4.1. Find block matches ending in W_j:

A.4.1.1. Compute $Rs(W_j)$.

A.4.1.2. Find all chunk matches within $Rp(W_0 \ldots W_{j-1})Rs(W_j)$.

A.4.2. Prepare for next iteration:

A.4.2.1. Find $Rss(W_j)$

A.4.2.2. If $|Rss(W_j)| = W_j$, find $Rp(W_0 \ldots W_j)$ from $Rss(W_j)$
else find $Rp(W_0 \ldots W_j)$ from $Rp(W_0 \ldots W_{j-1})Rss(W_j)$

From the above main algorithm, we can see that the pattern search is reduced to solving the following subproblems:

1. Preprocess $T^{(i)}$ and decompose $T^{(i)}$ into segments;
2. Find all segment matches in W_j;
3. Find $Rs(W_j)$;
4. Find $Rss(W_j)$;
5. Find $Rp(W_0 \ldots W_j)$ from $Rs(W_1 \ldots W_{j-1})Rss(W_j)$ or from $Rss(W_j)$;
6. Find all matches of P in $Rp(W_1 \ldots W_{j-1})Rs(W_j)$.

Several of these sub-tasks are intimately related. We define the following problems and show the relevant reductions.

Prefix-Suffix Matching

Preprecess: A string $S = s_1 \ldots s_n$.

Given: i, j such that $1 \leq j \leq i \leq n$.

Output: All occurrences of S in $S[1:i]S[j:n]$.

The subproblem 6 is trivially an instance of the prefix-suffix matching problem, a solution of which will be given in Section 4.1. The complexities involved will be $O(n)$ to preprocess the string, and $O(\log n + tocc)$ to answer a query, where *tocc* is the total number of occurrences, i.e. the output size.

A related problem is the following:

Prefix-Substring Matching

Preprocess: A string $S = s_1 \ldots s_n$.

Given: i, j, k such that $j \leq k$.

Output: $Rp(S[1:i]S[j:k])$.

In this case, we can immediately see that the first part of problem 5 is an instance of prefix-substring matching. The second part is also such an instance. We simply set i to 0 so that $S[1:i]$ is the empty string. Finally, subtask 3 is also an instance of prefix-substring matching as follows. Since we must solve subtask 4, we may as well also compute $Rss(W_j^R)$ at no extra cost. Then we can compute the $Rp(W_j^R)$ of pattern P^R by prefix-substring matching. This will be the same string as $Rs(W_j)$ of pattern P. A $O(n)$ preprocessing, $O(\log n)$ query time solution to the prefix-substring matching problem will be given in section 4.2.

Finally, we must compute all segment matches as well as $Rss(W_j)$ - and also $Rss(W_j^R)$ within the same bounds. We solve the first problem in time $O(|P| + tocc \log i)$ in section 5.1 and the second in $O(\log n)$ per segment in section 5.2. We give final details of updates and preprocessing, as well as summarizing the complexity of pattern matching in section 6.

3 Border Tree and its Properties

The border tree will be the main data structure which will allow us to solve the prefix-suffix and prefix-substring matching problems efficiently. We first give some preliminary definitions.

Let $S \in \Sigma^*$. We say that $S[1:i] \prec S[1:j]$ if $S[1:i]$ is a suffix of $S[1:j]$, and that $S[1:i] \preceq S[1:j]$ if $S[1:i] \prec S[1:j]$ or $i = j$. We say that $S[1:i]$ is a *border* of $S[1:j]$, denoted $S[1:i] \propto S[1:j]$, if $S[1:i] \prec S[1:j]$ and if there is no k such that $S[1:i] \prec S[1:k] \prec S[1:j]$. We can symmetrically define the above relations in terms of prefixes of suffixes.

In this section we first reveal some relations among prefixes of S. Then we introduce the *border tree*. All of the results apply to a similarly defined *suffix border tree*.

The following lemma is well known.

Lemma 3.1 ([6]) Given string S, suppose $S[1:t] \prec S[1:t+d]$ are two prefixes of a string, with $t > 0$ and $d > 0$. Let $t = \alpha d + s$ with $0 \leq s \leq d-1$, then $S[1:t] = S[1:d]^\alpha S[1:s]$ and $S[1:t+d] = S[1:d]^{\alpha+1} S[1:s]$.

We now introduce one of the main structural properties on borders.

Theorem 3.2 Let $S[1:a_1] \propto \ldots \propto S[1:a_l]$ be a chain of non-empty prefixes of a string S with $S[1:a_j] = S[1:a_{j-1}]D_{j-1}$. Then either $D_j = D_{j-1}$ or $|D_j| > |D_{j-1}|$ with $a_{j+1} > 3/2 a_j$.

Theorem 3.2 guarantees that while the number of prefixes in the prefix chain $S[1:a_1] \propto \ldots \propto S[1:a_l]$ can be as large as $O(|P|)$, the number of *different* D_j's in this chain can be at most $O(\log |P|)$. So we can compactly represent this chain by these different D_j's and their numbers of occurrences in the chain.

In [9], Knuth, Morris and Pratt introduced a pattern matching automaton which finds all occurrences of a pattern in a string in linear time. Their automaton has three components: nodes, one each for each prefix of the pattern string; success links, pointing from a prefix node n_i to the node of the next longest prefix, n_{i+1}; and failure links, which point from n_i to n_j if $j < i$ and $n_j \propto n_i$.

The failure links of the KMP automaton form a tree which we will call the *failure tree* of a string. Further, since the KMP automaton can be built in linear time, even for unbounded alphabet, the failure tree can also be built within the same bounds. This is in marked contrast with the suffix tree, the construction of which take time linear in the string length time log of the effective alphabet size.

We will interchangeably refer to a prefix and the node represented by the prefix in the failure tree. We also extend the definition of $\mathcal{L}(E(v))$ to be the string $S[|p(v)|+1 : |v|]$ for v a node in the failure tree of S.

In light of Theorem 3.2, we derive a new data structure, the *border tree*, from the failure tree. We define a *border tree* $T_b^S = (R, E, \mathcal{L})$ for a string S to be a tree with:

- Nodes set R which is a subset of the prefixes of S such that $v \in R$ iff either v has depth no more than 1 or $\mathcal{L}(E(v)) \neq \mathcal{L}(E(p(v)))$ in the failure tree of S;
- Edge set E derived by setting $p(v) = u$ iff $u \prec v$ and there is no node w in the vertex set such that $u \prec w \prec v$;
- Edge label $\mathcal{L}(E(u)) = (|u|, |D|, \alpha)$, where D is a non-empty string and α is the maximum integer such that PD^α is a prefix of S for $0 \leq \alpha \leq n$ and such that $P \propto u = PD \propto \ldots \propto PD^\alpha$.

Lemma 3.3 T_b^S can be built in $O(|S|)$.

Lemma 3.4 T_b^S has depth $O(\log |S|)$.

Proof: Follows from Theorem 3.2. ∎

4 Substring Problems

4.1 Prefix-Suffix Matching

Very briefly, we will find all prefix-suffix matches by consulting the border trees of string S and S^R. Some pairs of nodes $(u, v) \in T_b^S \times T_b^{S^R}$ will represent matches. Let $p \in T_b^S$ be the node representing our input prefix, and let $s \in T_b^{S^R}$ be the node representing our input suffix. By Lemma 3.4, each of p and s has $O(\log |S|)$ ancestors, and so there are $O(\log^2 |S|)$ pairs of ancestors to check. While this is a significant savings when compared to the $O(|S|^2)$ pairs to check if we where using a failure tree or suffix tree, we can further reduce our work to $O(\log |S|)$ by exploiting the fact that all matches are of length $|S|$ and so only a sparse subset of the possible ancestor pairs will be relevant to the computation.

We first provide more insights about the relations among prefixes of a string. Then we give a somewhat more detailed sketch of a procedure for efficient prefix-suffix matching.

Theorem 4.1 Let $S = a_1 \ldots a_n$ be a string such that $S[1 : l] \propto S[1 : l + d]$ and that $S[r : n] \propto S[r - h : n]$, with $l \geq d, r + h \leq n$ and $h \neq d$, then $(l + d) - (r - h) + 1 < 2\max(h, d)$.

Corollary 4.2 Let $X \propto XD \propto \cdots \propto XD^n$ be a chain of prefixes of S and let $Y \propto CY \propto \cdots \propto C^m Y$ be a chain of suffices of S, where X, D, Y and C are non-empty strings with $m \geq 2$ and $n \geq 2$. If $|D| > |C|$, then $|XD^i| + |C^j Y| < |S|$ for $0 \leq i \leq n-2$ and $0 \leq j \leq m$. And if $|D| < |C|$, then $|XD^i| + |C^j Y| < |S|$ for $0 \leq i \leq n$ and $0 \leq j \leq m-2$.

Proof: Follows from Theorem 4.1 by setting $l = |XD^{n-1}|$, $d = |D|$, $r = |C^{m-1}Y|$ and $h = |C|$. ▮

Lemma 4.3 Let $X \propto XD \propto \cdots \propto XD^n$ be a chain of prefixes of S and let $Y \propto CY \propto \cdots \propto C^m Y$ be a chain of suffices of S, where X, D, Y and C are non-empty strings. If $|D| = |C|$ and if there exist strings α, β and γ such that $XD^n = \alpha\beta$, $C^m Y = \beta\gamma$, $S = \alpha\beta\gamma$ and such that $|\beta| \geq |D|$, then $S = P[|D|]^\kappa P[t]$, where $|S| = \kappa|D| + t$ with $0 \leq t \leq |D| - 1$.

Note that if $\tilde{l} \leq l$ and $\tilde{r} \leq r$ are such that $S[1:\tilde{l}]S[\tilde{r}]$ is an occurrence of S in $S[1:l]S[r:n]$, then $\tilde{l}$ and $\tilde{r}$ must satisfy

$$S[1:\tilde{l}] \prec S[1:l], \quad S[\tilde{r}:n] \prec S[r:n], \quad \text{and} \quad \tilde{r} = \tilde{l} + 1. \tag{1}$$

As noted above, we want to avoid checking condition (1) for every possible $\tilde{l}$ and $\tilde{r}$. Instead, we want to take advantage of the short depth of trees T_b^S and $T_b^{S^R}$. Assume, as before that $p \in T_b^S$ represents $S[1:l]$ and that $s \in T_b^{S^R}$ represents $S[r:n]$. Let the path from the root to p consist of $(r(T_b^S) = p_1), p_2 \ldots, (p_k = p)$ and similarly, let $(r(T_b^{S^R}) = s_1), s_2, \ldots, (s_{k'} = s)$ be the path from the root to s. Recall that a single node in the border tree may represent a whole chain of prefixes of a string, so by p_k, we mean the node that represents the chain containing $S[1:l]$ and by $s_{k'}$ the node that represents the chain containing $S[r:n]$. Note also that $k, k' = O(\log|S|)$. Let $\mathcal{L}(p_k) = (l_p, d_p, n_p)$ with $l = l_p + \alpha \times d_p$, $0 \leq \alpha \leq n_p$, and let $\mathcal{L}(s_{k'}) = (r_s, c_s, m_s)$ with $n - r + 1 = r_s - \beta \times m_s$, $0 \leq \beta \leq c_s$.

By Condition (1), if $S[1:\tilde{l}]S[\tilde{r}:n]$ is a match in $S[1:l]S[r:n]$, then $S[1:\tilde{l}]$ is represented by some p_a, $1 \leq a \leq k$, and $S[\tilde{r}:n]$ is represented by some s_b, $1 \leq b \leq k'$. Let $i = k$ and $j = 1$. We simultaneously walk up on the p-path to reduce i and down on the s-path to increase j. Assume that we are at (p_i, s_j). Also assume that we have checked all possible matches of the form $S[1:\tilde{l}]S[\tilde{r}:n]$, where either $S[1:\tilde{l}]$ is represented by $P_{\tilde{\imath}}$ for some $\tilde{\imath} > i$, or $S[\tilde{r}:n]$ is represented by $S_{\tilde{\jmath}}$ for some $\tilde{\jmath} < j$.

Due to space consideration, we cannot give the details of the loops which implements this scheme. We note, however, that when checking a pair (p_i, s_j), each represents a set of prefixes or suffixes, respectively. We need only check for length constraints to see if we have a match. By Corollary 4.2 and Lemma 4.3, this requires, at most, the solving of one linear equation and therefore takes constant time. We climb down the p-path to shorten the prefixes and up the s-path to lengthen the suffixes.

Theorem 4.4 For string S, the prefix-suffix problem can be solved with linear preprocessing and query time $O(\log|S| + tocc)$, where $tocc$ is the total number of occurrences.

4.2 Prefix-Substring Matching

We give a sketch of the techniques used to solve the prefix-substring matching problem. Recall that we are given a prefix $S[1:i]$ and a substring $S[j:k]$, and we must find the longest prefix $S[1:l]$ such that $S[1:l]$ is a suffix of $S[1:i]S[j:k]$. We can consider two cases: either $l > k-j+1$ or $l \leq k-j+1$, in other words, either $S[1:i]$ contributes to $S[1:l]$ or it does not.

Suppose $l \leq k-j+1$. Then we find l by finding the longest border of $S[1:k]$ with length no more than $k-j+1$. However, we can do this is $O(\log |S|)$ time by consulting T_b^S as follows. Let n_k be the node in T_b^S representing $S[1:k]$. We can check in constant time for the longest prefix represented by n_k that is no longer than $k-j+1$. If no such prefix exists, we proceed to the parent of n_k and perform the same length check. Finally, there are $O(\log |S|)$ nodes on the path from n_k to the root of T_b^S.

Now suppose $l > k-j+1$. This means that $S[1:l] = S[1:l']S[j:k]$, for some $l' \leq l$. But $S[1:l']$ must be a border of $S[1:i]$. Therefore, $S[1:l]$ is a suffix of $S[1:i]S[j:k]$ exactly if $S[1:l-k+j]$ is a border of $S[1:l]$ and $S[j:k]$ occurs starting in the $l-k+j+1$th position of S. We once again resort to the border tree of S to find the appropriate borders of $S[1:i]$, and combine this with information from the suffix tree of S to find occurrences of $S[j:k]$. We refer the reader to the full paper for details.

Theorem 4.5 For string S, the prefix-substring problem can be solved with linear preprocessing and query time $O(\log |S|)$.

5 Work within segments

5.1 Chunk Matches

The goal of this subsection is to find all chunk matches. The example in the introduction showed that if the pattern is highly periodic, a single edit operation can interfere with many pattern occurrences. The following theorem gives the basis for exploiting the periodicity of a string to detect when many matches have been eliminated.

Theorem 5.1 Let P be a string with period d and length $d\alpha + s$, $s < d$, and let X, Y and Z be non-empty strings such that $XY = P$ and $YZ = P$. Then either $XYZ = P[1:d]^{\gamma}P[1:s]$ for some $\gamma > \alpha$ or $|X| > |P|/4$.

Therefore, when $\alpha = 1$, any operation to T can affect at most four matches. In such a case, the normal suffix tree indexing of a text T would allow us to find all chuck matches of P in $T^{(i)}$ in time $O(|P| + tocc \log i)$, where $tocc$ is the total number of occurrences of P in $T^{(i)}$.

As we have noted, many occurrences of a periodic pattern can occur in a small text segment. We formalize this as follows. Let a *pattern cluster* be a maximal substring of T of the form $P[1:d]^{\alpha+\tau}P[1:s]$, where d is the period of P, $s < d$, and with $\tau \geq 0$. There are

$\tau+1$ matches in a pattern cluster. By Theorem 5.1, any operation on $T^{(i)}$ can affect at most four pattern cluster. On the other hand, every chunk match of $T^{(i)}$ is a match in a pattern cluster.

In order to find the pattern clusters of a string, we proceed in two stages. We first find the beginning of the cluster and then determine its length. For pattern P with period d, a pattern cluster begins in T a some location k if P occurs at the kth position of T but $P[1:d]$ does not occur at the $k-d+1$st position of T. All such location can easily be found in the prefix tree of T be observing that if N_1 is the shallowest node such P is a prefix of $\mathcal{L}(N_1)$ and N_2 is the shallowest node such that $P[1:d]P$ is a prefix of $\mathcal{L}(N_2)$, then the locations we seek are the leaves which are descendants of N_1 but not N_2. If there are C clusters, then we can find their beginning locations in time $O(|P|+C)$ by finding N_1 and N_2 in $O(|P|)$ time (by tracing down from the root of T_p^T), and then in time $O(C)$ performing a depth first search to find all the appropriate leaves.

Finally, we need to determine the length of each cluster. Suppose there is a cluster beginning at location k. Then, by Lemma 3.1, we need only check the longest common prefix of $T[k:n]$ and $T[k+d:n]$ to find the end of the pattern cluster. But, as noted in subsection 2.2, this can be done in constant time. We conclude with:

Lemma 5.2 We can find all chunk matches of P in $T^{(i)}$ in time $O(|P|+tocc\log i)$, where *tocc* is the total number of occurrences.

5.2 Finding the representing substring of each chunk

Recall that we wish to find $Rss(W_j)$. Consider the tree T_s^T. It was shown in [1] that T_s^T can be converted into $T_s^{P\#T}$, $\# \notin \Sigma$ in $O(|P|)$ time. In $T_s^{P\#T}$, we will say that a node is *new* if it was not in T_s^T. We will say a node is *touched* if it has a new descendant. All other nodes will be *old*. Finally, let those new and touched nodes which are leaves or the least common ancestors of new leaves be *pattern* nodes.

Suppose that $W_j=T[a:b]$ and let l_a be the leaf in $T_s^{P\#T}$ representing $T[a:n]$. Then we can find $Rss(W_j)$ by finding the nearest non-old ancestor of l_a. This node will give the longest common prefix between $T[a:n]$ and a suffix of P. However, this algorithm will take time $O(|T|)$ to answer queries, or if we use more clever techniques for finding nearest marked ancestors in trees [2], $O(\log|T|/\log\log|T|)$. We present a method that will compute $Rss(W_j)$ in $O(\log|P|)$ time.

We give a brief outline. Let $\mathcal{E}(T)$ be the euler tour of tree T. In $O(|P|)$ we can build T_s^P and produce an euler tour of $\mathcal{E}(T_s^P)$ such that the nodes appear in the same order in $\mathcal{E}(T_s^P)$ that the pattern nodes do in in $\mathcal{E}(T_s^{P\#T})$. Now given any node $v\in T_s^{P\#T}$, we can find the nearest pattern node to the left and right in $\mathcal{E}(T_s^{P\#T})$ by doing a binary search in $\mathcal{E}(T_s^P)$ in $O(\log|P|)$ time. Let l_v and r_v be the left and right neighbor found by this method. We now have the following case analysis:

- l_v and r_v are both ancestors of v. Then $l_v=r_v$ and $Rss(\mathcal{L}(v))=\mathcal{L}(l_v)$.
- l_v is an ancestor of v but r_v is not. Then $Rss(\mathcal{L}(v))=\mathcal{L}(r_v)$.

- r_v is an ancestor of v but l_v is not. Then $Rss(\mathcal{L}(v)) = \mathcal{L}(l_v)$.
- Neither r_v nor l_v is an ancestor of v. Then $Rss(\mathcal{L}(v)) = \mathcal{L}(lca(l_v, r_v))$.

These operations can all be done in $O(1)$ using the constant time lca algorithm of Harel and Tarjan [8].

Lemma 5.3 For each segment, we can find its representing substring in $O(\log|P|)$ time.

6 Wrapping Up

6.1 Updates

From Algorithm A, we see that insertions and deletions affect the the decomposition of $T^{(i)}$ into segments. All other steps simply require that the segments be presented in increasing order by location in the text. It is trivial to maintain such a list by any balanced tree method in $O(\log|T^{(i)}|)$ time per operation, thus matching our claimed bounds.

6.2 Preprocessing

The text must be preprocessed to find $Rss(W_j)$ and chunk matches. All other operations are on pattern substrings and their complexity is not counted in with the text preprocessing. To find $Rss(W_j)$ and chunk matches, we need, by Section 5, to build a suffix tree and prefix tree on T and preprocess them for lca queries. Finally, we need an Euler tour of the suffix tree. As pointed out above, all these operation can be accomplished in linear time.

6.3 Complexity of Pattern Matching

We annotate Algorithm A with complexities.

Algorithm A: Pattern Search — Algorithm to find pattern P in $T^{(i)}$.

A.1. Preprocess P and decompose $T^{(i)}$ into segments $T^{(i)} = W_1 \dots W_i$;

A.2. Find all segment matches in $O(|P| + tocc \log i)$ §5;

A.3. $W_0 = \lambda, Rp(W_0) = \lambda$;

A.4. For each $j = 1, \dots, i$, do

A.4.1. Find block matches ending in W_j:

A.4.1.1. Compute $Rs(W_j)$ in $O(\log|P|)$ §5.2 & §4.2.

A.4.1.2. Find all chunk matches within $Rp(W_0 \dots W_{j-1})Rs(W_j)$ in $O(\log|P|)$ §4.1.

A.4.2. Prepare for next iteration:

A.4.2.1. Find $Rss(W_j)$ in $O(\log |P|)$ §5.2

A.4.2.2. If $|Rss(W_j)| = W_j$, find $Rp(W_0 \ldots W_j)$ from $Rss(W_j)$
else find $Rp(W_0 \ldots W_j)$ from $Rp(W_0 \ldots W_{j-1})Rss(W_j)$ in $O(\log |P|)$ §4.2.

Theorem 6.1 We can solve the *On-line dynamic text indexing problem* in time:

Preprocessing: $O(|T|)$

Edit operations: $O(\log |T^{(i)}|)$

Matching: $O(|P| + tocc \log i + i \log |P|)$

References

[1] A. Amir, M. Farach, R. Giancarlo, Z. Galil, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 1993. In press.

[2] A. Amir, M. Farach, R. M. Idury, H. La Poutré, and A. A. Schäffer. Improved dictionary matching. *Proc. of the Fourth Ann. ACM-SIAM Symp. on Discrete Algorithms*, 1993.

[3] B. Baker. A theory of parametrized pattern matching: Algorithms and applications. *Proc. of the 25th Ann. ACM Symp. on Theory of Computing*, pages 71–80, 1993.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[5] 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.

[6] Z. Galil. Optimal parallel algorithms for string matching. *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, 67:144–157, 1984.

[7] R. Giancarlo. The l-suffix tree of square matrix, with applications. *Proc. of the Fourth Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 402–411, 1993.

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

[9] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[10] U. Manber and E. Myers. Suffix arrays: A new method for on-line string searches. *Proc. of the First Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 319–327, 1990.

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

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