Dynamic Text and Static Pattern Matching

Amihood Amir[1], Gad M. Landau[2], Moshe Lewenstein[1], and Dina Sokol[1]

[1] Bar-Ilan University, Ramat Gan, Israel
{amir,moshe,sokold}@cs.biu.ac.il
[2] University of Haifa, Haifa 31905, Israel
landau@cs.biu.ac.il

Abstract. In this paper, we address a new version of dynamic pattern matching. The *dynamic text and static pattern matching problem* is the problem of finding a static pattern in a text that is continuously being updated. The goal is to report all new occurrences of the pattern in the text after each text update. We present an algorithm for solving the problem, where the text update operation is *changing* the symbol value of a text location. Given a text of length n and a pattern of length m, our algorithm preprocesses the text in time $O(n \log\log m)$, and the pattern in time $O(m\sqrt{\log m})$. The extra space used is $O(n + m\sqrt{\log m})$. Following each text update, the algorithm deletes all prior occurrences of the pattern that no longer match, and reports all new occurrences of the pattern in the text in $O(\log\log m)$ time.

1 Introduction

The *static pattern matching problem* has as its input a given text and pattern and outputs all text locations where the pattern occurs. The first linear time solution was given by Knuth, Morris and Pratt [12] and many more algorithms with different flavors have been developed for this problem since.

Considering the dynamic version of the problem, three possibilities need to be addressed.

1. A static text and dynamic pattern.
2. A dynamic text and a static pattern.
3. Both text and pattern are dynamic.

The static text and dynamic pattern situation is a traditional search in a non-changing database, such as looking up words in a dictionary, phrases is a book, or base sequences in the DNA. This problem is called the *indexing problem*. Efficient solutions to the problem, using suffix trees, were given in [18,14,16]. For a finite fixed alphabet, the algorithms preprocess the text T in time $O(|T|)$. Subsequent queries seeking pattern P in T can be solved in time $O(|P| + tocc)$, where $tocc$ is the number of occurrences of P in T. Farach [5] presented an improved algorithm, acheiving the same time bounds for large alphabets.

Generalizing the indexing problem led to the *dynamic indexing problem* where both the text and pattern are dynamic. This problem is motivated by making queries to a changing text. The problem was considered by [9,7,15,1]. The Sahinalp and Vishkin algorithm [15] achieves the same time bounds as the suffix tree algorithm for initial text preprocessing, $O(|T|)$, and for a search query for pattern P, $O(|P| + tocc)$, for bounded fixed alphabets. Changes to the text are either insertion or deletion of a substring S, and each change is performed in time $O(\log^3 |T| + |S|)$. The data structures of Alstrup, Brodal and Rauhe [1] support insertion and deletion of characters in a text, and movement of substrings within the text, in time $O(\log^2 |T| \log\log |T| \log^* |T|)$ per operation. A pattern search in the dynamic text is done in $O(\log |T| \log\log |T| + |P| + tocc)$.

Surprisingly, there is no direct algorithm for the case of a dynamic text and static pattern, as could arise when one is seeking a known and unchanging pattern in data that keeps updating. We were motivated for solving this missing version of dynamic pattern matching by the *two dimensional run-length compressed matching problem* [2]. The dynamic text pattern matching problem is a special case of the 2d run-length compressed matching problem where all pattern rows are *trivial*, i.e., consist of a single repeating symbol. This special case had no efficient solution in [2].

The **Dynamic Text and Static Pattern Matching Problem** is defined as follows:

Input: Text $T = t_1, ..., t_n$, and pattern $P = p_1, ..., p_m$, over alphabet Σ, where $\Sigma = \{1, ..., m\}$.

Preprocessing: Preprocess the text efficiently, allowing the following subsequent operation:

Replacement Operation: $\langle i, \sigma \rangle$, where $1 \leq i \leq n$ and $\sigma \in \Sigma$. The operation sets $t_i = \sigma$.

Output: Initially, report all occurrences of P in T. Following each replacement, report all **new** occurrences of P in T, and discard all **old** occurrences that no longer match.

The solutions of [15,1] can be adapted to solve our problem with the time bounds stated above. However, one would like a more direct and efficient way to answer queries for a static pattern and a text whose length does not change. In this paper we provide a direct answer to the dynamic text and static pattern matching problem, where the text update operation is *changing* the symbol value of a text location. After each change, both the text update and the reporting of new pattern occurrences are performed in only $O(\log\log m)$ time. The text preprocessing is done in $O(n \log\log m)$ time, and the pattern preprocessing is done in $O(m\sqrt{\log m})$ time. The extra space used is $O(n + m\sqrt{\log m})$. We note that the complexity for reporting the new pattern occurrences is not proportional to the number of pattern occurrences found since all new occurrences are reported in a succinct form.

We begin with a high-level description of the algorithm in Section 2, followed by some preliminaries in Section 3. In Sections 4 and 5 we present the detailed explanation of the algorithm. We leave the details of the data structures and proofs for the journal version.

2 Main Idea

2.1 Text Covers

The central theme of our algorithm is the representation of the text in terms of the static pattern. The following definition captures the notion that we desire.

Definition 1 (cover). *Let S and $S' = s'_1 \cdots s'_n$ be strings over alphabet Σ. A* cover *of S by S' is a partition of S, $S = \tau_1\tau_2 \ldots \tau_v$, satisfying -*
(1) substring property: *for each $1 \leq i \leq v$, τ_i is either a substring of S', or a character that does not appear in S'*
(2) maximality property: *for each $1 \leq i < v$, the concatenation of $\tau_i\tau_{i+1}$ is* not *a substring of S'.*

When the context is clear we call a cover of S by S' simply a cover. We also say that τ_h is an *element* of the cover. A cover element τ_h is represented by a triple $[i, j, k]$ where $\tau_h = s'_i \cdots s'_j$, and k, the *index* of the element, is the location in S where the element appears, i.e. $k = \sum_{l=1}^{h-1} |\tau_l| + 1$.

A cover of T by P captures the expression of the text T in terms of the pattern P. We note that a similar notion of a covering was used by Landau and Vishkin [13]. Their cover had the substring property but did not use the maximality notion. The maximality invariant states that each substring in the partition must be maximal in the sense that the concatenation of a substring and its neighbor is *not* a new substring of P. Note that there may be numerous different covers for a given P and T.

2.2 Algorithm Outline

Initially, when the text and pattern are input, any linear time and space pattern matching algorithm, e.g. Knuth-Morris-Pratt [12], will be sufficient for announcing all matches. The challenge of the Dynamic Text and Static Pattern Matching Problem is to find the new pattern occurrences efficiently after each replacement operation. Hence, we focus on the on-line part of the algorithm which consists of the following.
Online Algorithm

1. Delete old matches that are no longer pattern occurrences.
2. Update the data structures for the text.
3. Find new matches.

Deleting the old matches is straightforward as will be described later. The challenge lies in finding the new matches. Clearly, we can perform any linear time string matching algorithm. Moreover, using the ideas of Gu, Farach and Beigel [9], it is possible find the new matches in $O(\log m + pocc)$ time, where *pocc* are the number of pattern occurrences. The main contribution of this paper is the reduction of the time to $O(\log \log m)$ time per change. We accomplish this goal by using the cover of T by P. After each replacement, the cover of T must first be updated to represent the new text. We *split* and then *merge* elements to update the cover.

Once updated, the elements of the cover can be used to find all new pattern occurrences efficiently.

Observation 1 *Due to their maximality, at most one complete element in the cover of T by P can be included in a pattern occurrence.*

It follows from Observation 1 that all new pattern occurrences must begin in one of three elements of the cover, the element containing the replacement, its neighbor immediately to the left, or the one to the left of that. To find all new pattern starts in a given element of the cover, τ_x, it is necessary to check each suffix of τ_x that is also a prefix of P. We use the data structure of [9], the *border tree*, to allow checking many locations at once. In addition, we reduce the number of checks necessary to a constant.

3 Preliminaries

3.1 Definitions

In this section we review some known definitions on string periodicity, which will be used throughout the paper. Given a string $S = s_1 s_2 \dots s_n$, we denote the substring of S, $s_i \dots s_j$, by $S[i : j]$. $S[1 : j]$ is a *border* of S if it is both a proper prefix and proper suffix of S. Let x be the length of the longest border of S. S is *periodic*, with period $n - x$, if $x > n/2$. Otherwise, S is *non-periodic.*

A string S is *cyclic* in string π if it is of the form π^k, $k > 1$. A *primitive* string is a string which is not cyclic in any string. Let $S = \pi'\pi^k$, where $|\pi|$ is the period of S and π' is a (possibly empty) suffix of π. S can be expressed as $\pi'\pi^k$ for one unique primitive π. A *chain of occurrences* of S in a string S' is a substring of S' of the form $\pi'\pi^q$ where $q \geq k$.

3.2 Succinct Output

In the online part of the algorithm, we can assume without loss of generality that the text is of size $2m$. This follows from the simple observation that the text T can be partitioned into $2n/m$ overlapping substrings, each of length $2m$, so that every pattern match is contained in one of the substrings. Each replacement operation affects at most m locations to its left. The cover can be divided to allow constant time access to the cover of a given substring of length $2m$.

The following lemma can be easily proven using the properties of string periodicity. The full proof will appear in the journal version of the paper.

Lemma 1. *Let P be a pattern of length m and T a text of length $2m$. All occurrences of P in T can be stored in constant space.*

4 The Algorithm

The algorithm has two stages, the *static stage* and the *dynamic stage*. The static stage, described in Section 4.1, consists of preprocessing data structures and reporting all initial occurrences of P in T.

The *dynamic stage* consists of the processing necessary following each replacement operation. The main idea was described in Section 2. The technical and implementation details are discussed in Sections 4.2 and 5.

%vspace-.2 cm

4.1 The Static Stage

The first step of the static stage is to use any linear time and space pattern matching algorithm, e.g. Knuth-Morris-Pratt [12], to announce all occurrences of the pattern in the original text. Then, several data structures are constructed for the pattern and the text to allow efficient processing in the dynamic stage.

Pattern Preprocessing. Several known data structures are constructed for the static pattern P. Note that since the pattern does not change, these data structures remain the same throughout the algorithm. The purpose of the data structures is to allow the following four queries to be answered efficiently. The first two queries are used in the *text update step* and the second two are used for *finding new matches*. We defer the description of the data structures to the full version of the paper. The query list is sufficient to enable further understanding of the paper.

Query List for Pattern P

Longest Common Prefix Query (LCP): Given two substrings, S' and S'', of P. Is $S' = S''$? If not, output the position of the first mismatch.

$\Rightarrow$ Query Time [13]: $O(1)$.

Substring Concatenation Query: Given two substrings, S' and S'', of P. Is the concatenation $S'S''$ a substring of P? If yes, return a location j in P at which $S'S''$ occurs.

$\Rightarrow$ Query Time [3,6,10,19]: $O(\log\log m)$.

Longest Border Query: Given a substring S' of P, such that $S' = P[i:j]$, what is the longest border of $P[1:j]$ that is a suffix of S'?

$\Rightarrow$ Query Time [9]: $O(\log\log m)$.

Range Maximum Prefix Query: Given a range of suffixes of the pattern P, $S_i \dots S_j$. Find the suffix which maximizes the $LCP(S_\ell, P)$ over all $i \le \ell \le j$.

$\Rightarrow$ Query Time [8]: $O(1)$.

Text Preprocessing. In this section we describe how to find the cover of T by P for the input text T. Recall that we assume that the alphabet is linearly bounded in m. Thus, it possible to create an array of the distinct characters in P. The initial step in the cover construction is to create an element, τ_i, for each location i of the text. Specifically, for each location, $1 \le i \le n$, of the text, we identify a location of P, say $P[j]$, where t_i appears. We set $j = m + 1$ if t_i does not appear in P, and create $\tau_i = [j, j, i]$. Then, moving from left to right, we attempt to merge elements in the cover using the *substring concatenation query.* The initial cover is stored in a van Emde Boas [17] data structure, sorted by the indices of the elements in the text.

Time Complexity. The algorithm for constructing the cover runs in deterministic $O(n \log\log m)$ time. The amount of extra space used is $O(n)$. Creating an array of the pattern elements takes $O(m)$ time, and identifying the elements of T takes $O(n)$ time. $O(n)$ *substring concatenation queries* are performed, each one takes $O(\log\log m)$ time. The van Emde Boas data structure costs $O(n)$ time and space for its construction [17].

4.2 The Dynamic Stage

In the on-line part of the algorithm, one character at a time is replaced in the text. Following each replacement, the algorithm must delete the old matches that no longer match, update the text cover, and report all new matches of P in T. In this section we describe the first two steps of the dynamic stage. In Section 5 we describe the third step, finding the new matches.

Delete Old Matches. If the pattern occurrences are saved in accordance with Lemma 1 then deleting the old matches is straightforward. If P is non-periodic, we check whether the one or two pattern occurrences are within distance -m of the change. If P is periodic, we truncate the chain(s) according to the position of the change.

Update the Text Cover. Each replacement operation replaces exactly one character in the text. Thus, it affects only a constant number of elements in the cover.

```
Algorithm: Update the Cover
1. Locate the element in the current cover in which the replacement occurs.
2. Break the element into three parts.
3.Concatenate neighboring elements to restore the maximality property.
```

Step 1: Locate the desired element. Recall that the partition is stored in a van Emde Boas tree [17] which allows predecessor queries. Let x be the location in T at which the character replacement occurred. Then, the element in the partition in which the replacement occurs will be the $pred(x)$.

Step 2: Break Operation. Let $[i, j, k]$ be an element in the partition which covers the position x at which a replacement occurred. The break operation divides the element $[i, j, k]$ into the following three parts. We assume that the new character is at position q of the pattern. To find the new text character in the pattern we do as described in the algorithm for constructing the cover (Section 4.1).

(1) $[i, i + x - k - 1, k]$, the part of the element $[i, j, k]$ prior to position x.
(2) $[q, q, x]$, position x, the position of the replacement.
(3) $[i + x - k + 1, j, x + 1]$, the part of the element after position x.

Step 3: Restore maximality property. The maximality property is a local property, it holds for each pair of adjacent elements in the cover. As stated in the following lemma, each replacement affects the maximality property of only a constant number of pairs of elements. Thus, to restore the maximality it is necessary to attempt to concatenate a constant number of neighboring elements. This is done using the *substring concatenation query.*

Lemma 2. *Following a replacement and break operation to a cover of T, at most four pairs of elements in the new partition violate the maximality property.*

Time Complexity of Updating the Cover: The van Emde Boas tree implements the operations: insertion (of an element from the universe), deletion, and predecessor, each in $O(\log\log|U|)$ time using $O(|U|)$ space [17]. In our case, since the text is assumed to be of length $2m$, we have $|U| = m$. Thus, the predecessor of x in the cover (Step 1) can be found in $O(\log\log m)$ time. Step 2, the break operation, is done in constant time. Step 3, restoring the maximality property, performs a constant number of substring concatenation queries. These can be done in $O(\log\log m)$ time. Overall, the time complexity for updating the cover is $O(\log\log m)$.

5 Find New Matches

In this section we describe how to find all new pattern occurrences in the text, after a replacement operation is performed. The new matches are extrapolated from the elements in the updated cover.

Any new pattern occurrence must include the position of the replacement. In addition, a pattern occurrence may span at most three elements in the cover (due to the maximality property). Thus, all new pattern starts begin in three elements of the cover, the element containing the replacement, its neighbor immediately to the left, or the one to the left of that. Let the three elements under consideration be labeled τ_x, τ_y, τ_z, in left to right order. The algorithm *Find New Matches* finds all pattern starts in a given element in the text cover, and it is performed separately for each of the three elements, τ_x, τ_y, and τ_z. We describe the algorithm for finding pattern starts in τ_x.

The naive approach would be to check each location of τ_x for a pattern start (e.g. by performing $O(m)$ *LCP* queries). The time complexity of the naive algorithm is $O(m)$. In the following two subsections we describe two improved algorithms for finding the pattern starts in τ_x. The first algorithm has time $O(\log m)$ and the basic approach comes from [9]. Our algorithm, described in Section 5.2, improves upon this further. Our algorithm also uses the border tree of [9], but we use additional properties of the border groups (defined below) which allow a significant improvement in the time complexity. The total time for announcing all new pattern occurrences is $O(\log\log m)$.

Definition 2 (border groups [4]). *The borders of a given string $S[1:m]$ can be partitioned into $g = O(\log m)$ groups $B_1, B_2, \ldots, B_g$. The groups preserve the left to right ordering of the borders. For each B_i, either $B_i =$*

$\{\pi'_i\pi_i^{k_i}, \ldots, \pi'_i\pi_i^3, \pi'_i\pi_i^2\}$ or $B_i = \{\pi'_i\pi_i^{k_i}, \ldots, \pi'_i\pi_i\}$ where $k_i \geq 1$ is maximal, π'_i is a proper suffix of π_i, and π_i is primitive.[1]

The border groups divide the borders of a string S into disjoint sets, in left to right order. Each group consists of borders that are all (except possibly the rightmost one) periodic with the same period. The groups are constructed as follows. Suppose $\pi'\pi^k$ is the longest border of $S[1 : m]$. $\{\pi'\pi^k, \ldots, \pi'\pi^3, \pi'\pi^2\}$ are all added to group B_1. $\pi'\pi$ is added to B_1 if and only if it is *not* periodic. If $\pi'\pi$ is not periodic, it is the last element in B_1, and its longest border begins group B_2. Otherwise, $\pi'\pi$ is periodic, and it is the first element of B_2. This construction continues inductively, until π' is empty and π has no border.

5.1 Algorithm 1: Check Each Border Group

It is possible to use the algorithm of [9] to obtain a $O(\log m)$ time algorithm for finding all new pattern occurrences. The idea is to check all suffixes of τ_x which are prefixes of P. We group together all prefixes that belong to the same border group, and check them in *constant time*. The $O(\log m)$ time bound follows from the fact that there are at most $O(\log m)$ border groups to check. The border groups for any pattern prefix can be retrieved from the border tree of P.

Check one border group for pattern starts. Given a border group, $B_g = \{\pi'\pi^k, \pi'\pi^{k-1}, \ldots\}$, of which some element is a suffix of τ_x, compare π^{k+1} with the text following τ_x (using one or two LCP queries), to see how far right the period π recurs in the text. Depending on the length of the pattern prefix with period π, we locate all pattern starts in τ_x that begin with a border from B_g.

5.2 Algorithm 2: Check $O(1)$ Border Groups

Rather that checking *all* $O(\log m)$ border groups, our algorithm accomplishes the same goal by checking only a constant number of border groups. We use the algorithm for checking one border group to check the leftmost border group in τ_x, and at most one or two additional border groups.

Algorithm: Find New Matches

Input: An element in the cover, $\tau_x = [i, j, k]$.
Output: All starting locations of P in the text between t_k and t_{k+j-i}.

1. Find the longest suffix of τ_x which is a prefix of P. The *longest border query* (Section 4.1) returns the desired location. Let ℓ be the length of the suffix returned by the query.

[1] The definition of Cole and Hariharan [4] includes a third possibility, $B_i = \{\pi'_i\pi_i^{k_i}, \ldots, \pi'_i\pi_i, \pi'_i\}$, when π'_i is the empty string. In the current paper we do not include empty borders.

2. Using the Algorithm Check One Border Group (described in previous section), check the group of $P[1 : \ell]$, where ℓ is the length found in Step 1.
3. Choose $O(1)$ remaining border groups and check them using the Algorithm Check One Border Group.

Steps 1 and 2 were explained previously. It remains to describe how to choose the $O(1)$ border groups that will be checked in Step 3.

For ease of exposition we assume that the entire pattern has matched the text (say $\tau_x = P$), rather than some pattern prefix. This assumption does not limit generality since the only operations that we perform use the border tree, and the border tree stores information about each pattern prefix. Another assumption is that the longest border of P is $< m/2$. This is true in our case, since if P were periodic, then all borders with length $> m/2$ would be part of the leftmost border group. We take care of the leftmost border group separately (Step 2), thus all remaining borders will have length $< m/2$.

Thus, the problem that remains is the following. An occurrence of a non-periodic P has been found in the text, and we must find any pattern occurrence which begins in the occurrence of P. Note that there is at most one overlapping pattern occurrence since P is non-periodic. In Section 5.2 we describe some properties of the borders/border groups from Cole and Hariharan [4]. We use these ideas in Section 5.2 to eliminate all but $O(1)$ border groups.

Properties of the Borders. A *pattern instance* is a possible alignment of the pattern with the text, that is, a substring of the text of length m. The pattern instances that interest us begin at the locations of the borders of P. Let $\{x_1, x_2, \ldots\}$ denote the borders of P, with x_1 being the longest border of P. Let X_i be the pattern instance beginning with the border x_i.

Note that $|x_1| < m/2$ and P is non-periodic. Thus, although there may be $O(m)$ pattern instances, only one can be a pattern occurrence. The properties described in this section can be used to isolate a certain substring of the text, overlapping *all* pattern instances, which can match at most three of the overlapping pattern instances. Moreover, it possible to use a *single* mismatch in the text to discover which three pattern instances match this "special" text substring.

The following lemma from Cole and Hariharan [4] relates the overlapping pattern instances of the borders of P.

Definition 3 ([4]). *A* clone set *is a set $Q = \{S_1, S_2, \ldots\}$ of strings, with $S_i = \pi'\pi^{k_i}$, where π' is a proper suffix of primitive π and $k_i \geq 0$.*

Lemma 3. *[4] Let X_a, X_b, X_c, $a < b < c$, be pattern instances of three borders of P, x_a, x_b, x_c, respectively. If the set $\{x_a, x_b, x_c\}$ is not a clone set, then there exists an index d in X_1 with the following properties. The characters in $X_1, X_2, \ldots, X_a$ aligned with $X_1[d]$ are all equal; however, the character aligned with $X_1[d]$ in at least one of X_b and X_c differs from $X_1[d]$. Moreover, $m - |x_a| + 1 \leq d \leq m$, i.e. $X_1[d]$ lies in the suffix x_a of X_1.*

Each border group is a clone set by definition, since every border within a group has the same period. However, it is possible to construct a clone set from elements in two different border groups. The last element in a border group can have the form $\pi'\pi^2$, in which case the borders $\pi'\pi$ and π' will be in (one or two) different border groups. It is not possible to construct a clone set from elements included in more than three distinct border groups. Thus, we can restate the previous lemma in terms of border groups, and a single given border, as follows.

Lemma 4. *Let x_a be a border of P with pattern instance X_a, and let x_a be the rightmost border in its group (definition 2). At most two different pattern instances to the right of X_a can match x_a at the place where they align with the suffix x_a of X_1.*

Let $r = m - |x_1| + 1$. Note that $P[r]$ is the location of the suffix x_1 in P. Since all pattern instances are instances of the same P, an occurrence of a border x_a in some pattern instance below X_a, aligned with $X_a[r]$, corresponds exactly to an occurrence of x_a in P to the left of $P[r]$. The following claim will allow us to easily locate the two pattern instances which are referred to in Lemma 4.

Claim. Let x_a be a border of P, and let x_a be the rightmost border in its group (definition 2). Let $r = m - |x_1| + 1$, where x_1 is the longest border of P. There are at most two occurrences of x_a beginning in the interval $P[r - |x_a|, r]$.

The Final Step. Using ideas from the previous subsection, our algorithm locates a *single mismatch* in the text in constant time. This mismatch is used to eliminate all but at most three pattern instances. Consider the overlapping pattern instances at the mth position of X_1. By Lemma 3, we have an identical alignment of all borders of P at this location. Each x_i is a suffix of all x_j such that $i > j$, since all x_i are prefixes and suffixes of P. Thus, suppose that the algorithm does the following. Beginning with the mth location of X_1, match the text to the pattern borders from right to left. We start with the shortest border, and continue sequentially until a mismatch is encountered. Let x_a be the border immediately below the border with the mismatch. The first mismatch tells two things. First, *all* borders with length longer than $|x_a|$ mismatch the text. In addition, at most two pattern instances with borders shorter than $|x_a|$ match x_a at the location aligned with the suffix x_a of X_1 (Lemma 4).

The algorithm for choosing the $O(1)$ remaining borders is similar to the above description, however, instead of sequentially comparing text characters, we perform a single LCP query to match the suffix x_1 with the text from right to left.

Algorithm: Choose $O(1)$ Borders (Step 3 of Algorithm *Find New Matches*)
A: Match P from *right to left* to the pattern instance of x_1 by performing a single LCP query.
B: Find the longest border that begins following the position of the mismatch found in Step A.

C: Find the $O(1)$ remaining borders referred to in Lemma 4.
D: Check the borders found in Steps B and C using the algorithm for checking one border group.

An *LCP* query is performed to match the suffix x_1 of X_1, with the text cover from right to left. (Step **A**). The position of the mismatch is found in constant time, and then a *longest border query* is used to find x_a (Step **B**). Once X_a is found, we know that all pattern instances to its left mismatch the text. It remains to find the possibilities to the right of X_a which are referred to in Lemma 4. Claim 5.2 is used for this purpose.

Step C: Let $r = m - |x_1| + 1$. The possible occurrences of x_a in pattern instances to the right of X_a correspond to occurrences of x_a in the interval $P[r - |x_a|, r]$.

By Claim 5.2 there are at most two occurrences of x_a in the specified interval. Since x_a is a pattern prefix, three *range maximum prefix queries* will give the desired result. The first query returns the maximum in the range $[r - |x_a|, r]$. This gives the longest pattern prefix in the specified range. If the length returned by the query is $\geq |x_a|$, then there is an occurrence of x_a prior to position r. Otherwise, there is no occurrence of x_a aligned with $X_a[r]$, and the algorithm is done. If necessary, two more maxima can be found by subdividing the range into two parts, one to the left and one to the right of the maximum.

Step D: The final step is to check each border group, of which there are at most three, using the Algorithm Check One Border Group.

Time Complexity of Algorithm Find New Matches: As shown previously, each step of the algorithm takes either constant time or $O(\log\log m)$ time. Thus, overall, the algorithm has time complexity $O(\log\log m)$.

We summarize the algorithm, including the time and space complexity of each step.
Preprocessing: $O(n\log\log m + m\sqrt{\log m})$ time and $O(n + m\sqrt{\log m})$ space.
On-line algorithm: $O(\log\log m)$ time per replacement.

Pattern Preprocessing: The following data structures are necessary to answer the queries listed in Section 4.1.
(1) The suffix trees for P and the reverse of P: $O(m)$ time/space [5]. The suffix trees must be preprocessed for:
(a) lowest common ancestor queries: $O(m)$ time/space [11],
(b) weighted ancestor queries: $O(m)$ time/space, combined results of [6,10, 19], and
(c) node intersection queries: $O(m\sqrt{\log m})$ time/space [3].
(2) The border tree for P is constructed in $O(m)$ time/space [9], and
(3) a range-maximum prefix array for P is created in $O(m)$ time/space [8].

Text Preprocessing: (Section 4.1)
(1) Construct the cover of T by P: $O(n\log\log m)$ time, $O(n)$ space.
(2) Store the cover in a van Emde Boas data structure: $O(n)$ time/space.

The Dynamic Algorithm: (Sections 4.2,5)
(1) Delete old matches that are no longer pattern occurrences: $O(\log\log m)$ time.

(2) Update the data structures for the text: $O(\log\log m)$ time.
(3) Find new matches: $O(\log\log m)$ time.

6 Conclusion

In this paper we presented an algorithm for the Dynamic Text and Static Pattern Matching Problem, allowing character replacements to be performed on the text. Solving this problem for insertions and deletions in the text remains an interesting open problem. In addition, we would like to extend our algorithm to allow a general alphabet; currently the assumption is that the alphabet is linearly bounded by m. Other directions would be to solve *approximate* pattern matching or *multiple* pattern matching over a dynamic text.

References

1. S. Alstrup, G. S. Brodal, T. Rauhe: Pattern matching in dynamic texts. *Proc. of the Symposium on Discrete Algorithms* (2000) 819–828
2. A. Amir, G. Landau, and D. Sokol: Inplace run-length 2d compressed search. *Theoretical Computer Science* **290**, 3 (2003) 1361–1383
3. A. Buchsbaum, M. Goodrich and J. Westbrook: Range searching over tree cross products. *Proc. of European Symposium of Algorithms* (2000) 120–131
4. R. Cole and R. Hariharan: Tighter upper bounds on the exact complexity of string matching. *SIAM J. on Computing* **26**,3(1997) 803–856
5. Martin Farach: Optimal suffix tree construction with large alphabets. *Proc. of the Symposium on Foundations of Computer Science* (1997) 137–143
6. M. Farach and S. Muthukrishnan: Perfect hashing for strings: formalization and algorithms. *Proc. of Combinatorial Pattern Matching* (1996) 130–140
7. P. Ferragina and R. Grossi: Fast incremental text editing. *Proc. of the Symposium on Discrete Algorithms* (1995) 531–540
8. H.N. Gabow, J. Bentley, and R.E. Tarjan. Scaling and related techniques for geometric problems. *Proc. of the Symposium on Theory of Computing* (1984) 135–143
9. M. Gu, M. Farach, and R. Beigel: An efficient algorithm for dynamic text indexing. *Proc. of the Symposium on Discrete Algorithms* (1994) 697–704
10. T. Hagerup, P.B. Miltersen and R. Pagh: Deterministic dictionaries. *J. of Algorithms* **41** (2000) 69–85
11. D. Harel and R. E. Tarjan: Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing* **13**,2, (1984) 338–355
12. D. Knuth, J. Morris and V. Pratt: Fast pattern matching in strings. *SIAM J. on Computing* **6**,2 (1977) 323–350
13. G.M. Landau and U. Vishkin: Fast string matching with k differences. *Journal of Computer and System Sciences* **37**,1 (1988) 63–78
14. E. M. McCreight: A space-economical suffix tree construction algorithm. *J. of the ACM* **23** (1976) 262–272
15. S. C. Sahinalp and U. Vishkin: Efficient approximate and dynamic matching of patterns using a labeling paradigm. *Proc. of the Symposium on Foundations of Computer Science* (1996) 320–328
16. E. Ukkonen: On-line construction of suffix trees. *Algorithmica* **14** 249–260

17. P. van Emde Boas: An $O(n \log \log n)$ on-line algorithm for the insert-extract min problem. *Technical Report, Department of Computer Science, Cornell University*, Number TR 74-221 (1974)
18. P. Weiner: Linear pattern matching algorithm. *Proc. of the Symposium on Switching and Automata Theory* (1973) 1–11
19. D.E. Willard: Log-logarithmic worst case range queries are possible in space θ(n). *Information Processing Letters* **17** (1983) 81–84