Dynamic Text and Static Pattern Matching

Amihood Amir* Gad M. Landau† Moshe Lewenstein‡ Dina Sokol§

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. We note that the complexity is not proportional to the number of pattern occurrences since all new occurrences can be reported in a succinct form.

1 Introduction

Historical pattern matching had been motivated by text editing (e.g. [22, 19, 23]). Some later motivation stemmed from molecular biology (e.g. [27]). While initially automated string processing needed to handle texts whose change was mainly a very slow growth (for example, [1, 2]), recent advances in digital libraries (such as news, medical, etc.) and the advent of the World Wide Web, pitted the area of combinatorial pattern matching against rapidly changing and highly dynamic texts.

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 [19] and many more algorithms with different flavors have been developed for this problem since.

*Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel, +972 3 531-8770; amir@cs.biu.ac.il; and College of Computing, Georgia Tech, Atlanta, GA 30332-0280. Partly supported by NSF grant CCR-01-04494 and ISF grant 282/01.

†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 grant CCR-0104307, by the Israel Science Foundation grant 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities and by IBM Faculty Partnership Award.

‡IBM TJ Watson Research Center; email: moshe@watson.ibm.com.

§Brooklyn College of the City University of New York, sokol@sci.brooklyn.cuny.edu; Part of this work was done when the author was at Bar-Ilan University, supported by the Israel Ministry of Industry and Commerce Magnet grant (KITE) .

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*. An efficient solution to the problem, using suffix trees, was given by Weiner [30]. Additional algorithms were presented in [24, 28, 11]. For a finite fixed alphabet, Weiner's algorithm preprocesses 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. Weiner's paper was succeeded by a myriad of algorithms for suffix tree and suffix array construction (e.g. [18]), handling infinite alphabets, improving the space constants, handling real-time constraints, and more.

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 [15, 13, 25, 4]. The Sahinalp and Vishkin algorithm [25] achieves the same time bounds as the Weiner 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 [4] 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. Imagine a database that is continuously updated in a multi-user environment which we desire to protect from viruses. One common method of virus detection software is signature-based virus detection. In this setting we are given a collection of patterns (the virus signatures) which we want to detect-online. This is exactly the dynamic text and static pattern problem.

Another setting is SDI scenarios (selective dissemination of information), e.g. news clippings, where incoming news are searched looking for a set of patterns extracted from user profiles. Another example of SDI scenarios is in a financial markets database where users predefine what information they are looking for. We note that while mentioning this motivation we view our paper primarily as a theoretical result.

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 $\Sigma = \{1, \ldots, m^c\}$, for an arbitrary[1] constant c.

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

[1]We note that in section 4.1.1 we change the alphabet assumption to $\Sigma = \{1, \ldots, m\}$, and explain there why this can be done without loss of generality.

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.

Using the dynamic indexing algorithm of Sahinalp and Vishkin [25], a text update can be performed in $O(\log^3 n)$ time, and all occurrences of P can be reported in $O(m + aocc)$ time, where *aocc* is the number of all pattern occurrences in the text (including those that were present in both the old and updated text). Given the fact that the pattern does not change, it seems a waste to pay the $O(m)$ penalty for every query, as well as reporting all occurrences, even those that did not change. In addition, one would like a more efficient way for handling text updates, considering the facts that the pattern is static and the text length does not change. The algorithm of [4] offers a slight improvement, since the text updates can be performed in $O(\log^2 n \log\log n \log^* n)$ time and the pattern search in time $O(\log n \log\log n + aocc)$.

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. Section 6 contains the description of the data structures that are used for the pattern. We conclude with a summary in Section 7 and open problems in Section 8.

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, $S = \tau_1 \tau_2 \ldots \tau_v$, each τ_i, $i = 1, ..., v-1$, satisfying -*
(1) substring property: *τ_i is a substring of S',*
(2) maximality property: *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 [21]. 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 [19], 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.

On-line 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 [15], as we describe in Section 5.2, 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 will 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 single 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. 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.

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 [15], 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 $S[i : j]$. A *proper* prefix of S is a prefix of S that does not equal S, inclusive of the empty string. (A proper suffix is defined in the same way.) $S[1 : j]$ is a *border* of S if it is both a proper prefix and proper suffix of S. We refer to the index of the suffix part of the border as the *start* of the border in S. Let x be the length of the longest border of S. S is *periodic*, with period size $n - x$, if $x \geq n/2$. Otherwise, S is *non-periodic*.

A string S is *cyclic* in string π if it is of the form π^k, $k > 1$. In other words, a cyclic string is a periodic string with an integral number of periods. A *primitive* string is a string which is not cyclic in any string. Let $S = \pi'\pi^k$, where $|\pi|$ is the period size of S and π' is a (possibly empty) suffix of π. S can be expressed as $\pi'\pi^k$ for one unique primitive π, called *the period* of S. 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 on-line 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 n/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, thus at most two text substrings of size $2m$ must be dealt with. The cover can be divided to allow constant time access to the cover of a given substring of length $2m$.

In the following lemma we show that for a text of length $2m$, all of the output can be stored in $O(1)$ space.

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.*

Proof: If P is non-periodic, there are at most two pattern occurrences in the text, since the overlap of occurrences, by definition, is $< m/2$. If P is periodic with period size $|\pi|$ then we store information about the *chains* of occurrences of P. A chain consists of a sequence of overlapping occurrences of P at jumps of $|\pi|$. We store the beginning of each chain, its length, and the period size. There are at most two chains of P in a text of size $2m$. □

4 The Algorithm

The algorithm has two stages, the *static stage* and the *dynamic stage*. The static stage consists of constructing the data structures for the pattern and the text, and reporting all initial occurrences of P in T.

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

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 [19], 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.

4.1.1 Pattern Preprocessing

Without loss of generality, we can assume that $\Sigma = \{1, \ldots, m\}$. In the problem definition, P is a string of length m over the alphabet $\Sigma = \{1, \ldots, m^c\}$. Thus, although $|\Sigma| = m^c$, there are at most m distinct characters in P, and since P is static, these are the only characters of interest. We can construct a *perfect* deterministic hash function to map the m distinct characters in P to the numbers $\{1, \ldots, m\}$. The hash function $h : \{1, \ldots, m^c\} \rightarrow \{1, \ldots, m\}$ can be created in linear time, and evaluated in $O(\log \log m)$ time.

To create the desired hash function it is sufficient to have a static predecessor structure over the alphabet $\Sigma = \{1, ..., m^c\}$, which can be created in linear time and which supports queries in $O(\log \log m)$ time. The reason is as follows. For each of the (at most) m distinct characters of P we save $h(x)$ as satellite data within the static predecessor structure. Then, when given y for which we desire to compute $h(y)$, we ask for the predecessor of $y + 1$. If it is y, then $h(y)$ is obtainable from the satellite data. If it is not y, then y is not a character of P and we return $m + 1$.

The details of the static predecessor structure are included in Section 6.2.1, see the section on Static Predecessors. Henceforth, we assume an alphabet $\Sigma = \{1, \ldots, m\}$.

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 queries to be answered efficiently. The first two queries are used in the *text update step* and the third is used for *finding new matches*. We defer the description of the data structures to Section 6 since 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 the pattern P. Is $S' = S''$? If not, output the position of the first mismatch.

Query Time [21]: $O(1)$.

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

Query Time [9, 3, 31]: $O(\log \log m)$.

Range Maximum Prefix Query: Let Suf_i denote the suffix of the pattern P that begins at location i. Given a range of suffixes of the pattern P, $Suf_i \ldots Suf_j$. Find the suffix which maximizes the $LCP(Suf_\ell, P)$ over all $i \leq \ell \leq j$.

Query Time [14]: $O(1)$.

4.1.2 Text Preprocessing

In this section we describe how to construct the initial cover of T by P for the input text T. After constructing the initial cover, it is stored in a van Emde Boas [29] data structure, sorted by the indices of the elements in the text, to handle the changes to the cover that will occur from the replacement operations. The van Emde Boas tree maintains an ordered set over a bounded universe $\{1, \ldots, U\}$. In our case $|U| = 2m$, since we assume that the text size is $2m$ (see section 3.2), and the keys in the tree are the indices into the text. 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.

Algorithm Construct the Initial Cover

Input: A pattern $P = p_1, \ldots, p_m$ preprocessed for the queries listed in Section 4.1.1, and a text $T = t_1, \ldots, t_n$.

Output: A cover of T by P. (Definition 1)

Begin Algorithm

1. Create an array A, of size m, containing the first location in P of each character in Σ. If an element $1 \leq i \leq m$ does not appear in P, set $A[i] = m + 1$. Thus, if $A[i] = j$, then either $p_j = i$ or $j = m + 1$. Since $\Sigma = \{1, \ldots, m\}$, one pass over P is sufficient to fill A.

2. For each location in the text, $1 \leq i \leq n$,
   identify a location of P, say p_j, where t_i appears. Set $j = m + 1$ if t_i does not appear in P. Create $\tau_i = [j, j, i]$.

3. $i = 1$ (Set the current element to τ_1.)
   While $i < n$ do
   Take the current element (τ_i) and the element to its right (τ_{i+1}) and merge them, if possible, by performing a *substring concatenation query*. If the substrings do not concatenate, set $i = i + 1$.

4. Store the cover in a van Emde Boas tree with elements sorted by their indices in the text.

End Algorithm

Time Complexity: The algorithm for constructing the cover runs in deterministic $O(n \log \log m)$ time. The amount of extra space used is $O(n)$. Since we assume that the alphabet is $\{1, \ldots, m\}$, creating an array of the pattern elements takes $O(m)$ time and identifying the elements of T takes $O(n)$ time. If we assume that $\Sigma = \{1, \ldots, m^c\}$ then labeling the text would take $O(n \log \log m)$

time (see Section 4.1.1). $O(n)$ *substring concatenation queries* are performed, each one takes $O(\log \log m)$ time. The van Emde Boas data structure costs $O(n \log \log m)$ time and $O(n)$ space for its construction [29].

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 achieve the following three goals.

On-line Algorithm

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

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.

4.2.1 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 delete the one or two pattern occurrences that are within distance -m of the change. If P is periodic, we truncate the chain(s) according to the position of the change.

4.2.2 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. In this section we describe how to update the cover of T by P after each replacement operation. Updating the cover consists of the following four steps.

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.
4. Update the van Emde Boas tree which stores the text cover.

Step 1: Locate the desired element
Recall that the cover is stored in a van Emde Boas tree [29] indexed by the start of each element

in the text. Let x be the location in T at which the character replacement occurred. Then, the element in the cover in which the replacement occurs will be the element in the van Emde Boas tree indexed by the largest number that is smaller than or equal to x, i.e. the *predecessor* of $x+1$.

Step 2: Break operation
Let $[i, j, k]$ be the element in the cover which covers the position x at which a replacement occurred. The break operation divides the element $[i, j, k]$ into three parts: (1) the part before the replacement, (2) the replaced character, (3) the part after the replacement. Calculating the indices of parts 1 and 3 are trivial. To find the position of the replaced text character in the pattern we access the array of distinct characters of P that was created in step 1 of Algorithm Construct the Initial Cover (Section 4.1.2). Let q be a position in the pattern of the new text element. The following are the three new elements in the text cover:

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.

Example 1: The pattern P and text T are shown with their characters numbered. (Although $\Sigma = \{1, \ldots, m\}$, we use characters for the pattern and text, and numbers as indices, to ease exposition.)

	1	2	3	4	5	6
$P =$	a	b	b	a	c	b

	1	2	3	4	5	6	7	8	9	10	11	12
$T =$	c	b	a	b	b	a	b	b	b	a	c	b
$Cover =$	$[5,6,1]$		$[1,4,3]$				$[2,3,7]$		$[3,6,9]$			

The array of distinct pattern characters is:

a	b	c
1	2	5

Let a replacement occur in the text at location 4, changing the b to a c. During the break operation, the element of the cover $[1, 4, 3]$ is broken into 3 parts:
1. [1,1,3] = "a"
2. [5,5,4] = "c"
3. [3,4,5] = "ba"

Step 3: Restore maximality property
The maximality property is a local property, it holds for each pair of adjacent elements in the cover. We show in the following lemma that 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.*

Proof: The four pairs of elements that possibly violate the maximality property are the pairs of adjacent elements in which the left element of the pair is either one of the three new elements (the parts created by the break operation) or the element to their immediate left. If any other element is non-maximal, then it contradicts the maximality of the cover prior to the break. □

Referring back to Example 1, the element [1,1,3] is combined with [5,5,4] to give [4,5,3] = "ac". Thus, the maximality is restored to the text cover.

Step 4: Update the van Emde Boas tree
The cover is stored in a van Emde Boas tree, which must be updated to reflect the change. At most three deletions are necessary, one for the element with the change, and one for its neighbor to the right and to the left. At most three insertions are necessary for the new elements of the cover.

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 [29]. In our case, since the text is assumed to be of length $2m$, and since elements of the cover are stored in the van Emde Boas tree by their indices in the text, we have $|U| = 2m$. Thus, the predecessor, insertions and deletions of Steps 1 and 4 can be performed 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 this section we describe two improved algorithms for finding the pattern starts in τ_x. The algorithms use the border tree of [15], which we describe in detail in Section 5.1. The first algorithm, described in Section 5.2, has time $O(\log m)$. Our main result is presented in Section 5.3, where the total time for announcing all new pattern occurrences is $O(\log\log m)$.

5.1 Border Groups and Border Trees

In the classical KMP algorithm, an automaton is constructed for the given pattern $P[1:m]$. The KMP automaton has three components: (1) nodes, one for each prefix of P, numbered 1 through

m, (2) success links, pointing from node i to node $i+1$, and (3) failure links that point from a node i, to the node that represents the *longest border* of $P[1:i]$. The failure links of the KMP automaton form a tree called the *failure tree* of P, denoted by FT_P (for e.g., see figure 1). A path in the failure tree of P to a node i represents *all* of the borders of the prefix $P[1:i]$.

In our algorithm, we would like to consider all borders of a given pattern prefix. However, given a single prefix of P, $P[1:i]$, the number of borders of $P[1:i]$ can be as large as $O(i)$ (e.g., $P = a^i$). We would like a way to compress this information. Using the definition of Cole and Hariharan [10] we will impose a grouping on the borders of a single pattern prefix into $O(\log i)$ groups.

Definition 2 (border groups [10]) *The borders of a given string $P[1:i]$ can be partitioned into $g = O(\log i)$ groups $B_1, B_2, \ldots, B_g$. The groups preserve the left to right ordering on the suffixes of $P[1:i]$, i.e. the suffixes in group B_j are to the left of the suffixes in group B_{j+1} for all $1 \leq j < g$. For each B_j, either $B_j = \{\pi'_j\pi_j^{k_j}, \ldots, \pi'_j\pi_j^3, \pi'_j\pi_j^2\}$ or $B_j = \{\pi'_j\pi_j^{k_j}, \ldots, \pi'_j\pi_j\}$ where $k_j \geq 1$ is maximal, π'_j is a proper suffix of π_j, and π_j is primitive.*[2]

The border groups divide the borders of a string $P[1:i]$ into disjoint sets, in left to right order on the suffixes of $P[1:i]$. 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 $P[1:i]$. $\{\pi'\pi^k, \ldots, \pi'\pi^3, \pi'\pi^2\}$ 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.

The $\log i$ bound on the number of groups follows from the fact that the leftmost element in each group has length no more than 2/3 the length of the last element in the previous group (since $|\pi'| \leq |\pi'\pi|/2$ and $|\pi'\pi| \leq \frac{2}{3}|\pi'\pi^2|$.)

Example 2: The string *S=bababbababcbababbabab* has four borders: $\{bababbabab, babab, bab, b\}$. The division into border groups will create three groups, $\{bababbabab\}, \{babab, bab\}$ and $\{b\}$. For the first group, $\pi = babab$ and π' is empty, for the second group, $\pi = ab$ and $\pi' = b$, and for the third group, $\pi = b$, π' is empty.

We can now use this grouping to compress the failure tree into a data structure called the *border tree*[3], introduced by Gu-Farach-Beigel [15]. The advantage of the border tree is that it provides the border groups for *every* prefix $P[1:i]$ of a given pattern $P[1:m]$. The *border tree* is derived from the failure tree by essentially compressing each group of borders into a single node. The node of the shortest border in each group will represent the group. First, we label each node i in the failure tree FT_P as follows.

1: If $P[1:i]$ is periodic, then $P[1:i] = \pi'\pi^k$ and the label is $(|\pi'|, |\pi|, k)$.

2a: If $P[1:i]$ is non-periodic, then it can be denoted as $\pi'\pi$, where π' is its longest border. The label is $(|\pi'|, |\pi|, 1)$.

[2]The definition of Cole and Hariharan [10] includes a third possibility, $B_i = \{\pi'_j\pi_j^{k_j}, \ldots, \pi'_j\pi_j, \pi'_j\}$ when π'_j is the empty string. In the current paper we do not list the empty string as a border.

[3]A conference proceedings with the results of Cole and Hariharan [10] preceeded the results of [15].

2b: There is one exception to rule 2a. For a non-periodic prefix $P[1:i]$, with a child with $k = 2$, the label is $(|\pi'|, |\pi|, 1)$ with π equal to that of the child with $k = 2$.

The labeling can be done in linear time in two rounds. First, all nodes with $k \geq 2$ are labeled. A node $P[1:i]$ has $k \geq 2$ if the length of its parent (i.e. its longest border) is longer than $i/2$. The period size of $P[1:i]$ is $i - |parent(i)|$. When a node with $k = 2$ is encountered, we label its parent according to rules 2a and 2b. In the second round, all remaining nodes are labeled according to rule 2a. Rule 2b states that if a node $\pi'\pi$ is non-periodic, and its child is $\pi'\pi^2$, its label should be $(|\pi'|, |\pi|, 1)$ whether or not it has a border longer than $|\pi'|$. This will ensure that the nodes are both included in the same group. This is illustrated in the following example.

Example 3. Let $\pi' = aba$ and $\pi = dabacabadaba$. Then, $\pi'\pi = abadabacabadaba$, and its longest border is $abadaba$.

The border tree results from a compression of the failure tree. The general rule is: remove all periodic nodes, i.e. all nodes having $k \geq 2$. The knowledge about the periodic nodes can be included in the nodes' ancestor that is non-periodic. This corresponds to representing an entire border group by its shortest element, and the length of the longest element. The one exception is the case where the border group's shortest element has $k = 2$. In this case, we would like the periodic node to remain in the border tree. According to the construction of the border groups, a group includes $\pi'\pi^2$ as its shortest element iff $\pi'\pi$ is periodic. Thus, any node with $k = 2$ whose parent is periodic, is also included in the border tree.

Formally, the border tree $BT_s = (R, E, \mathcal{L})$ for a string S, is a tree with:

1. Node set R which is a subset of the nodes of FT_s. Let v be a node in FT_s with label (i, j, k). $v \in R$ iff $k < 2$, or $k = 2$ and $parent(v)$ is periodic.
2. Edge set E derived by setting $parent(v) = u$ if u is the closest ancestor of v in FT_s which is included in BT_s. (Note that u is a border of v, and there is no node $w \in R$ such that u is a border of w and w is a border of v.)
3. Node label $\mathcal{L}(v) = (|\pi'|, |\pi|, k)$, where π and π' are the same as in FT_s, and k is the maximum integer, $0 < k \leq n$, such that $\pi'\pi^k$ is a prefix of S.

Remark. We note that the meaning of the label k in the failure tree differs from the label k in the border tree. In the failure tree, k is simply the number of periods of a pattern prefix. In the border tree, k is maximal over all pattern prefixes with a given period.

While constructing the border tree it is trivial to store a pointer for each node *not* included in BT_s to its closest ancestor in the failure tree that is included in BT_s. This allows us to refer to the node in the border tree of a given pattern prefix. See figure 1 for an example of a border tree and the groups represented by each node.

Observation 2 *A path from the root to a given node v in BT_s represents the groups of borders of the node v.*

Let w be an ancestor of a node v, with label $(|\pi'|, |\pi|, k)$. Although w represents a border group of v, the value of the label k is not necessarily correct for v since k represents the longest pattern prefix with period π. For our purposes, the correct k for a given prefix is not needed; only the largest k is of interest. However, if it is necessary to determine the appropriate k value for w in relation to v, then it can be done by performing one LCP query between the reverse of π^k and the reverse of v, and then dividing the result by $|\pi|$. For example, consider node 10, in figure 1. Its parent, with label $(0, 2, 3)$, represents prefix 6, *ababab*. The longest common suffix of prefix 10 and prefix 6 gives the longest border of 10 with period *ab*, which has length 2.

Time Complexity: The border tree for P can be built in time and space $O(m)$ and it has depth $O(\log m)$ [15].

5.2 Algorithm 1: Check each border group

Let τ_x be the element of the text cover in which we are searching for new pattern starts. τ_y and τ_z are the following two elements in the cover. The main idea of the first algorithm was first used in [15]: It is necessary to check only the positions in τ_x which represent a prefix of P. Furthermore, it is possible to group all prefixes which belong to the same border group, and to check them together. We show how to check a given border group for pattern starts in the following algorithm. This algorithm is applied once for each border group which is a suffix of τ_x. The border groups of τ_x can be retrieved from the border tree of P [15] (described in Section 5.1). If $\tau_x = P[i : j]$, then the path in the border tree of the node representing $P[1 : j]$ gives all borders of τ_x. Only borders that begin after position i will be checked. There are at most $O(\log m)$ border groups of any given pattern prefix, which correspond to the nodes on the path of the given prefix.

The following algorithm checks all possible pattern starts in a single border group of a pattern prefix. If the border group has one or two borders, we can check each border separately in constant time. The difficult situation is when a border group contains several pattern prefixes that are periodic with the same period. Given a border group of a pattern prefix, $\{\pi'\pi^j, \pi'\pi^{j-1}, \ldots\}$, its node in the border tree of P has a label, $(|\pi'|, |\pi|, k)$. The label k represents the *largest* integer such that $\pi'\pi^k$ is a prefix of P. Thus, the first step of this algorithm will be to check how many times the period π recurs in the text by comparing π^{k+1} to $\tau_y\tau_z$.

Since P begins with $\pi'\pi^k$ and it does not begin with $\pi'\pi^{k+1}$ there are two possible cases. The first is that P has a period size $|\pi|$ (i.e. $P = \pi'\pi^k\pi''$, π'' is a prefix of π). The second case is that there exists a location $|\pi'\pi^k| < q \leq |\pi'\pi^{k+1}|$ at which point the periodicity of size $|\pi|$ is broken, i.e. $p_i \neq p_{q-|\pi|}$. By comparing P with itself at position $|\pi| + 1$ we can determine which case is true for the given P.

Algorithm: Check One Border Group for Pattern Starts

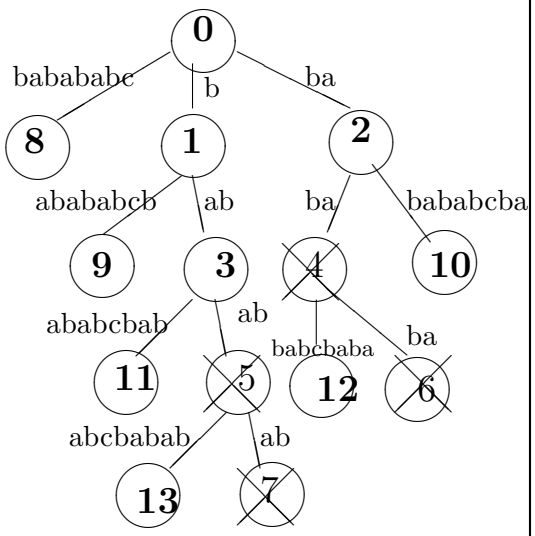
Input: (1) A border group B_g of a pattern prefix with node label $(|\pi'|, |\pi|, k)$.

(2) t_i which is the start in the text of the leftmost border of B_g that is covered by element τ_x. (The two elements in the cover that follow τ_x are τ_y and τ_z.)

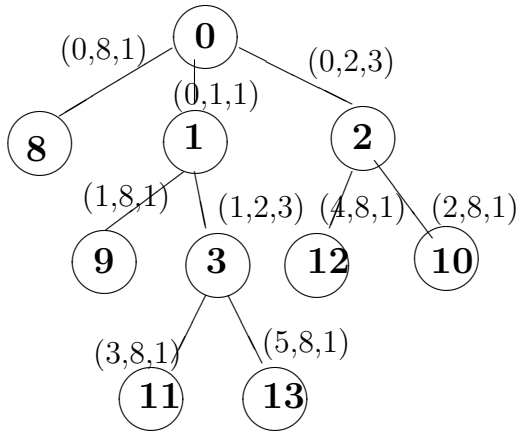
Output: All pattern starts in τ_x that begin with a border in B_g.

Pattern with length 13: $b\ a\ b\ a\ b\ a\ b\ c\ b\ a\ \ b\ \ a\ \ b$
KMP Automaton states: 0 1 2 3 4 5 6 7 8 9 10 11 12 13

Failure Tree:



Border Tree:



Node/ Prefix Length	*Label in Failure Tree* π'	π	k	*Label in Border Tree*
1		b	1	(0,1,1)
2		ba	1	(0,2,3)
3	b	ab	1	(1,2,3)
4		ba	2	
5	b	ab	2	
6		ba	3	
7	b	ab	3	
8		bababab c	1	(0,8,1)
9	b	abababcb	1	(1,8,1)
10	ba	bababcba	1	(2,8,1)
11	bab	ababcbab	1	(3,8,1)
12	baba	babcbaba	1	(4,8,1)
13	babab	abcbabab	1	(5,8,1)

Figure 1: The failure tree and border tree for $S = bababab cbabab$.

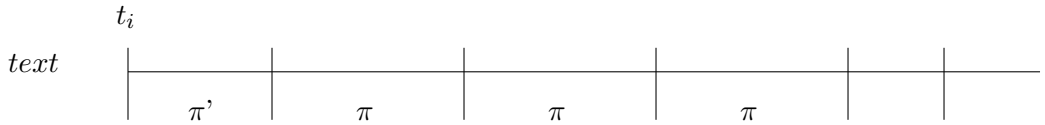


Figure 2: A chain of recurring π's begins at position t_i in the text. If P is periodic in π, then we can report matches at all positions $t_i, t_{i+|\pi|}, t_{i+2|\pi|}, \ldots$, as long as the match following the starting position has length longer than m.

Begin Algorithm

1. Match π^{k+1} to $\tau_y\tau_z$. π^k can be matched against $\tau_y\tau_z$ by an LCP query between $P[|\pi'|+1 \ldots m]$ and τ_y, followed by an LCP query between the continuation of P and τ_z. In the event that π^k matches a prefix of $\tau_y\tau_z$, we use $P[|\pi'|+1 \ldots m]$ in a third LCP query to check for π^{k+1}.

2. Match $P[1 \ldots m]$ to $P[|\pi|+1 \ldots m]$ using an LCP query. We use this to determine whether P is periodic in $|\pi|$.

3. **Case 1:** P is periodic in $|\pi|$.
   Report matches beginning at t_i and at jumps of π, as long as the remaining match is longer than m.

   Case 2: P is not periodic in $|\pi|$, i.e. $\exists$ a location q, $|\pi'\pi^k| < q \leq |\pi'\pi^{k+1}|$, such that $p_q \neq p_{q-|\pi|}$. There is only one possible pattern start in border group B_g of τ_x since the border group is periodic in π and P is not. The only location necessary to check is the location of the rightmost occurrence of $\pi'\pi^k$ beginning in τ_x, which was determined in step 1. We check this location against the pattern by performing one or two LCP queries (between $P[|\pi'\pi^k|+1 \ldots m]$ and the appropriate location in $\tau_y\tau_z$.)

End Algorithm

Lemma 3 *It is sufficient to compare π^{k+1} to the text to find all pattern starts in the group B_g.*

Proof: Since $\pi'\pi^k$ is the longest border in B_g, then $\pi'\pi^{k+1}$ is not a proper prefix of the pattern. We are looking for pattern starts within τ_x, thus, it is not necessary to check any further than $|\pi^{k+1}|$ characters to the right of τ_x. □

Time Complexity: The time complexity of the algorithm for checking one border group is $O(1)$ since the LCP query (Section 4.1.1) is done in constant time. In addition, we can report many matches in $O(1)$ time, by Lemma 1. The algorithm for finding all pattern starts in τ_x has time $O(\log m)$ since there are at most $O(\log m)$ border groups for every prefix of P (definition 2).

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

Our algorithm improves upon this further. The idea is that it is not necessary to check all $O(\log m)$ border groups. Indeed, we can achieve 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 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. Let ℓ be the length of the desired suffix.
2. Using the Algorithm Check One Border Group (described in previous section), check the group of $P[1 : \ell]$.
3. Choose $O(1)$ remaining border groups and check them using the Algorithm Check One Border Group.

In this paragraph we give a high-level description of Step 1 of Algorithm 2; the following paragraph specifies the details. Let $\tau_x = P[i : j]$. Step 1 is to find the longest suffix of $P[i : j]$ that is a prefix of the pattern P. Suppose that we are given the list of all borders of $P[1 : j]$. A binary search on the given list would give the answer in $O(\log m)$ time. Furthermore, we can compress the border list to be a list of the border groups of $P[1 : j]$, of which there are at most $O(\log m)$ (def. 2). Then, since the border groups order the borders from left to right, a binary search on the list of border groups would answer the query in $O(\log \log m)$ time.

Given a prefix $P[1 : j]$ of P, its list of border groups is represented by the path of $P[1 : j]$ in the border tree of P. Thus, it is possible to perform a binary search on the path of $P[1 : j]$ in the border tree of P. One method for implementing the binary search on trees is to use the level-ancestor queries from [6], where each level-ancestor query takes $O(1)$ time. Thus, a binary search on a path in the tree is done in $O(\log \log m)$ time. Hence, Step 1 has time $O(\log \log m)$.

Step 2 uses the algorithm explained in the previous section. 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. The method that we use to find this occurrence (if it exists) is to choose a constant number of border groups of P, and to check these border groups using the Algorithm Check One Border Group. In Section 5.3.1 we describe some properties of the borders/border groups from Cole and Hariharan [10]. We use these ideas in Section 5.3.2 to eliminate all but $O(1)$ border groups.

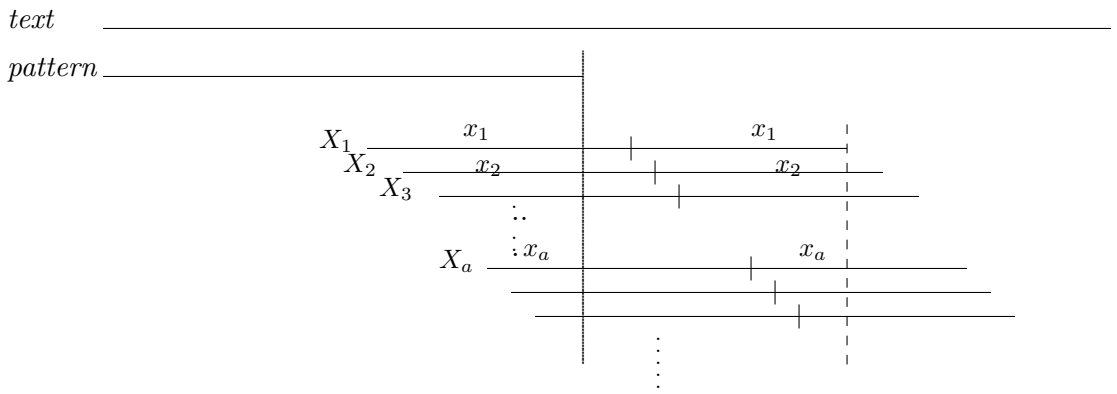


Figure 3: The pattern P matches the text, and its list of borders is shown. The borders are numbered $\{x_1, x_2, \ldots\}$ with pattern instances $\{X_1, X_2, \ldots\}$. X_1 is the pattern instance of the longest border of P.

5.3.1 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 [10] relates the overlapping pattern instances of the borders of P.

Definition 3 ([10]) *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 4 *[10] 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. ($\{x_a, x_b, x_c\}$ is a subset of the set of borders of P, $\{x_1, x_2, \ldots\}$, with x_1 being the longest border of P.) 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.*

Figure 3 accompanies Lemma 4. Shown in the figure is an alignment of the pattern instances which correspond to the borders of P. The borders, as well as the border groups, are left-to-right ordered.

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 5 *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.*

Proof: By Lemma 4, at most one border that cannot form a clone set with x_a, can match x_a ending at position $X_1[m]$. In addition, since x_a is the rightmost element in its group, at most two shorter borders can form a clone set with x_a. Either the borders that form a clone set with x_a (of which there are at most two), **or** a single border that cannot form a clone set with x_a will match x_a aligned with the suffix of X_1. □

Let $r = m - |x_1| + 1$. Note that $P[r]$ is the location of the suffix x_1 in P (see figure 3). 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 5.

Claim 1 *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 $P[r - |x_a|, r]$.*

Proof: Let $P[r - \delta]$, $\delta < |x_a|$, be the location of an occurrence of x_a in the specified interval (see figure 4). The occurrence of x_a at $P[r - \delta]$ overlaps the x_a beginning at $P[r]$. The overlap is both a prefix and a suffix of x_a and thus is a border to the right of x_a with length $|x_a| - \delta$. Let x_b be the border of P with length $|x_a| - \delta$. The pattern instance X_b is shifted δ characters to the right of X_a, and hence $X_b[r - \delta]$ aligns with $X_a[r]$. Thus, X_b has an occurrence of x_a aligned with location r of X_a. By Lemma 5 there are at most two such borders to the right of x_a. Thus, there are at most two occurrences of x_a in the interval $P[r - |x_a|, r]$. □

5.3.2 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. As shown in figure 3, and by Lemma 4, 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

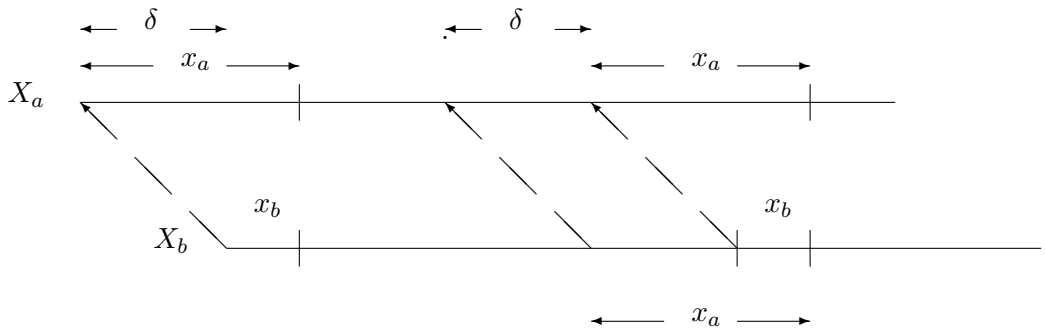


Figure 4: Two borders of P are shown with their corresponding pattern instances, X_a and X_b. The instance X_b begins δ characters to the right of X_a. If X_b has the string x_a at position $r - \delta$ (where $r = m - |x_1|$) then it tells of an overlapping occurrence of x_a in the pattern beginning at location $r - \delta$. The length of the overlap between the two x_a's is $|x_b|$.

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 $|x_a|$ (in figure 3 borders above 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 5).

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 5.

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. The position of the mismatch is found in constant time (Step **A**). A binary search on the border groups will find x_a (Step **B**). Once X_a is found, we know that all pattern instances to its left (above, in figure 3) mismatch the text. It remains to find the possibilities to the right of X_a which are referred to in Lemma 5. Claim 1 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 1 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: Step 1 is a binary search which takes $O(\log \log m)$ time (using level-ancestor queries of [6]). Step 2 is done in constant time. Step 3A is a constant time LCP query and 3B is a binary search done in $O(\log \log m)$ time. Steps 3C and 3D are both done in constant time since there are at most three groups to check and the checking of a group is done in time $O(1)$ using the Algorithm Check One Border Group. Thus, overall, the Algorithm Find New Matches has time complexity $O(\log \log m)$.

6 Data Structures for the Pattern

The cover of T by P stores the text in terms of substrings of the pattern. In the course of the algorithm, it is necessary to perform various operations on these pattern substrings. In order to perform these operations efficiently, we construct several data structures for the pattern. In this section we describe the data structures for the pattern and how they are used to answer the queries listed in Section 4.1.1. In Section 6.1 we describe the longest common prefix query, Section 6.2 the substring concatenation query, and Section 6.3 the range maximum prefix query.

6.1 The Longest Common Prefix Query

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

The method used for the longest common prefix query is well known, and it originates in [20]. It uses the suffix tree of P, defined as follows.

Suffix tree: The suffix tree of a string $S = s_1 s_2 \dots s_n$ over alphabet Σ is a compacted trie of all suffixes of $S\$$ where $\$ \notin \Sigma$. The leaves of the tree represent the suffixes of S. We denote the suffixes of a string of length n with $Suf_1, \dots, Suf_n$. Each edge of the tree has a label, $w \in \Sigma^*$, such that for each suffix Suf_u associated with leaf u the concatenation of the labels on the path of root-to-u forms the suffix Suf_u. Associated with each node v of the suffix tree is a height $h(v)$, which is the sum of the lengths of the labels on the path root-to-v. The suffix tree can be constructed in linear time and space [11].

The suffix tree of P is preprocessed for the *lowest common ancestor* (LCA) queries. To compare two substrings $S' = P[i : j]$ and $S'' = P[k : l]$ we take the LCA of the nodes of the suffixes Suf_i and Suf_k. The height of the LCA is exactly the length of the prefix of S' and S'' that is a match.

Time Complexity: A suffix tree for an arbitrary alphabet can be constructed in $O(m)$ time using Farach's method [11] for alphabets taken from a polynomially bounded range. Preprocessing a tree for LCA queries is done in $O(m)$ time and the LCA query time is $O(1)$ by the algorithm of Harel

and Tarjan [17]. Other LCA algorithms appear in [26, 6].

6.2 Substring Concatenation Query

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

The algorithm for the *substring concatenation query* uses the suffix tree of the pattern, denoted by ST, and the suffix tree of the reverse of P, $P_{\texttt{rev}}$, denoted by ST^R. The idea of the algorithm is as follows. $S'S''$ is a substring of P iff there is a location in $P_{\texttt{rev}}$ at which the reverse of S' occurs which **equals** a location in P at which S'' occurs. To check whether such a position exists, we locate the node of S'' in ST and the node of the reverse of S' in ST^R. Any element in the intersection of the leaves of these two nodes is a position in P of the concatenation $S'S''$.

Substring Concatenation Algorithm

Input: Two substrings of P, $S' = P[i:j]$ and $S'' = P[k:l]$.
Output: A location in P of the concatenation $S'S''$ if one exists.

1. Locate the node u in the suffix tree of P which represents the substring S''. This is done by performing a weighted ancestor query on the leaf Suf_k in ST with weight $l - k + 1$.
2. Locate the node v in the suffix tree of P_{rev} which represents the reverse of S'. This is done by performing a weighted ancestor query on the leaf of the reverse suffix Suf_j in ST^R with weight $j - i + 1$.
3. Check the intersection of $leaves(u)$ and $leaves(v)$, by performing a node intersection query. If non-empty, an element in the intersection is the location of the concatenated substring $S'S''$ in P.

Steps 1 and 2 of the algorithm consist of locating a weighted ancestor in a bounded weighted tree (to be defined shortly). Step 3 consists of taking the intersection of the leaves of two nodes. In the following two subsections we describe how each of these operations is performed.

6.2.1 Weighted Ancestors

The weighted ancestor problem was defined in [12] and is a generalization of the level-ancestor problem [6]. A tree containing m nodes is a *weighted increasing tree*, or *weighted tree* for short, if each node v has an associated positive integer weight $w(v)$ that satisfies: (1) $w(\text{parent}(v)) < w(v)$, where parent$(v)$ is the parent node of v in the tree. Also, $w(root) = 0$. The tree is said to be a *bounded weighted tree* if (2) $w(v) \leq m$ for all v. In the *weighted ancestor problem* we wish to preprocess a weighted, or weighted bounded, tree in order to answer the following subsequent queries:

Query: Given a node v and a height h, find the ancestor u on the path from v to the root where $w(u) \geq h$ and $w(\text{parent}(u)) < h$.

Here, as in [12], we will be primarily interested in the weighted ancestor problem on bounded weighted trees. In [12] an $O(m)$ time randomized preprocessing algorithm was suggested supporting subsequent queries in $O(\log\log m)$ time. Also an $O(m^{1+\epsilon})$ time deterministic algorithm was suggested supporting queries in $O(1)$ time. We propose an $O(m)$ time *deterministic* preprocessing algorithm so that subsequent queries can be answered in $O(\log\log m)$ time.

Heavy Path Decomposition

We use a variation of one of the solutions in [12], in which the bounded weighted tree T is partitioned into paths by a heavy path decomposition. In a heavy path decomposition the edge set of T is decomposed into disjoint paths as follows. Greedily choose a path by walking from the root down the tree picking, at each step, the child which has the most nodes in its subtree. Once a path is chosen, we remove its edges and are left with a forest. The process is recursively reiterated in each of the trees of the forest. The *decomposition tree* $D(T)$ is defined by the heavy path decomposition, with nodes in $D(T)$ corresponding to paths in the heavy path decomposition. The edges of $D(T)$ are defined from node v_i (corresponding to path p_i) to node v_j (corresponding to path p_j), if the node v at the head of the path p_j also appears on the path p_i (in other words, path p_j exits from path p_i). A well-known, crucial property of the decomposition tree is that its height is $O(\log m)$. This follows from the greedy choice of the path decomposition.

Let T be a bounded weighted tree and $D(T)$ its decomposition tree. We maintain with each node v_i of $D(T)$ a weight, defined to be the weight of the node v in T which is the head of path p_i, the path corresponding to v_i. Now, let u be a leaf in T and let h be the height in the weighted ancestor query. If we perform a weighted ancestor query in $D(T)$ with the same height h and the leaf v_j (in $D(T)$) corresponding to path p_j which contains u as a tail-node, then the answer is a node v_i corresponding to path p_i, where p_i contains the node v which is the answer to the weighted ancestor query on T. Hence, our desire is to answer the weighted ancestor query in $D(T)$ quickly.

Weighted Ancestor Queries in $D(T)$

To answer weighted ancestor queries in $D(T)$, it is sufficient to implement a binary search on the (upwards) path from v_j to the root. Since the height of $D(T)$ is $O(\log m)$ this will yield an $O(\log\log m)$ time binary search. To implement the search, we use the level-ancestor queries from [6], where each level-ancestor query takes $O(1)$ time.

Hence, we can find, in time $O(\log\log m)$, the node v_i in $D(T)$ which corresponds to path p_i in T which contains the node v that is the answer to the weighted ancestor query problem. So, our problem is to find the correct node v within path p_i. To solve our problem we transform it into a static predecessor query problem. In the static predecessor problem (see [31, 8]) one preprocesses a set of integers for subsequent predecessor queries.

The transformation is as follows. Consider a path in the path decomposition. The weights on this path are bounded by m. Consider each path as a sequence of its weights. Note that there are $\Theta(m)$ paths in the path decomposition (as each ends in a unique leaf) and, although nodes of T may appear in several paths, the number of nodes in all paths is $\leq 2m$ (as the decomposition partitions the edge set). Enumerate the paths of the decomposition p_1, ..., p_{cm} in an order of your choice. We define the *weight sequence* of p_i, $\bar{p_i}$, to be the sequence of weights of the nodes of p_i with weight im added to each of the weights. Now concatenate the sequence $\bar{p_1}, \cdots, \bar{p_{cm}}$ to receive an increasing sequence A of size $\leq 2m$ with all weights on the sequence bounded by m^2. To find the node v on

path p_i which is the answer to the weighted ancestor query on T with h we ask a predecessor query on A with $h + im$ from the location where $\bar{p_i}$ ends in A.

Static Predecessors

We will use the y-fast trie of Willard [31] to solve the static predecessor query. First, we describe the x-fast-trie and then extend to the similar data structure of the y-fast-trie. In addition, we combine the Willard structure with the deterministic perfect hash function construction[4] of Alon and Naor [3]. This is to avoid the random hash functions used in [31].

Willard [31] proposed x-fast tries for the predecessor problem over a universe $U = \{1, ..., u\}$. The x-fast trie over an ordered set $S \subset U$ of size m is a trie on the binary representations of S. The size of the x-fast trie is $O(m \log u)$. For fast access, all nodes (more precisely, the locus associated with the node) are stored using a perfect hash function. Predecessor queries are answered in $O(h_q \log \log u)$ time, where h_q is the hash query time, by binary searching on the height of the trie. The time it takes to construct the x-fast-trie over an ordered set S is linear in the space of the trie plus the time it takes to construct the perfect hash function. A perfect hash function for k elements can be constructed deterministically in $O(k \log^5 k)$ time and $O(k)$ space such that subsequent queries are answered in $O(1)$ time [3]. In our case $u = m^2$, $\log u = O(\log m)$ and hence $k = O(m \log u) = O(m \log m)$. Therefore, in $O(m \log^6 m)$ time and $O(m \log m)$ space one can construct the x-fast trie where subsequent queries are answered in $O(\log \log m)$ time.

To be more space and time efficient we partition S into sets of size $O(\log^6 m)$ each, following the idea of the y-fast trie data structure [31]. For each set we choose one arbitrary element from the set as a representative. The overall number of representatives is $O(\frac{m}{\log^6 m})$. Hence, for the representatives an x-fast trie can be constructed in $O(m)$ time with $O(\log \log m)$ time queries. In addition for each set (of size $O(\log^6 m)$) a binary search tree can be constructed in linear time with query time $O(\log \log^6 m) = O(\log \log m)$. To find a predecessor (of, say x) in S one does as follows. Ask a predecessor query of x on the x-fast trie of the representatives. Let y be the representative answer. The predecessor in S must appear in the set of y or in the set of the "representative successor" of y. To find it a search is applied in the binary search tree of these two sets. Hence, in $O(m)$ time and space one can deterministically preprocess for weighted ancestor queries in $O(\log \log m)$ time.

6.2.2 Node Intersections

Assume we are given two trees each with $O(m)$ nodes and m leaves uniquely labeled with the numbers from 1 to m. Let $leaves(u)$ denote the set of labels on the leaves in the subtree rooted at node u. Given a node u in one tree and a node v in the other tree a node intersection query returns an element of the intersection of $leaves(u)$ and $leaves(v)$, if one exists. The algorithm of Buchsbaum et al. [9] answers node intersection queries.

[4]We note that better constructions that Alon and Naor exist, specifically see [16], however they assume stronger models which we prefer to avoid.

6.2.3 Time Complexity

The preprocessing for the Substring Concatenation Algorithm takes $O(m\sqrt{\log m})$ time and space. The weighted ancestor algorithm takes $O(m)$ time and space, and the node intersection algorithm of Buchsbaum et al. [9] takes $O(m\sqrt{\log m})$ time and space. Subsequent queries can be answered in $O(\log\log m)$ deterministic time.

6.3 Range Maxima Prefix Query

Query: Let Suf_i denote the suffix of the pattern P that begins at location i. Given a range of suffixes of the pattern P, $Suf_i \ldots Suf_j$. Find the k which maximizes the $LCP(Suf_\ell, P)$ over all $i \leq \ell \leq j$.

We construct an array of integers, $A[1 \ldots m]$, such that $A[i]$ equals the longest common prefix (LCP) of P and the suffix of P beginning at location i. Note that the array A can be constructed by performing m LCP queries.

We then preprocess the array A for range maxima queries, as in [14]. A range-maximum query returns the largest number in the given interval in the array. The answer to the range maximum query on the array $A[i:j]$ gives the location of the longest prefix of P in the interval $i \ldots j$.

Time Complexity: The preprocessing consists of constructing the array A and preparing it for range maxima queries. Performing m LCP queries takes $O(m)$ time. The algorithm of [14] takes $O(m)$ time for the preprocessing, and subsequent range-maximum queries can be answered in $O(1)$ time.

7 Summary

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: (Section 6)

1. Construct the suffix trees for P and $P_{\texttt{rev}}$: $O(m)$ time/space.
2. Preprocess the suffix trees for:
   (a) Lowest common ancestor queries: $O(m)$ time/space.
   (b) Weighted ancestor queries: $O(m)$ time/space.
   (c) Node intersection queries: $O(m\sqrt{\log m})$ time/space.
3. Construct the border tree for P: $O(m)$ time/space.
4. Construct the range-maximum prefix array for P: $O(m)$ time/space.

Text Preprocessing: (Section 4.1.2)

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: $O(n \log\log m)$ time and $O(n)$ 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.

8 Open Problems

There are a number of problems that remain open in the domain of dynamic text and static pattern matching. First, the changes that we dealt with are replacements in the text. An open challenge would be to allow insertions/deletions in the text. Another variation would be to allow a general alphabet, while our algorithm assumes an alphabet polynomially bounded by m.

Also, we propose a solution where one pattern exists. If there are multiple patterns, say r patterns, then our solution requires the space and time to be multiplied by r. Can this be solved more quickly?

Another important question is that of changes of substrings. Say a short substring (of length s) was changed, does it require $O(s \log\log n)$ time? Or can this be done faster? Say $O(s + \log\log n)$ time.

Another possible avenue of research is to solve approximate pattern matching with a changing text.

References

[1] The Waterloo University new Oxford English dictionary. `http://db.uwaterloo.ca/OED`.

[2] The Bar-Ilan University Responsa project. `http://www.biu.ac.il/JH/Responsa/`.

[3] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica* 16:434-449, 1996.

[4] S. Alstrup, G. S. Brodal, T. Rauhe. Pattern matching in dynamic texts. *Proc. of the Symposium on Discrete Algorithms*, pages 819–828, 2000.

[5] A. Amir, G. Landau, and D. Sokol. Inplace run-length 2d compressed search. *Theoretical Computer Science*, 290(3):1361–1383, 2003.

[6] O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. of Computer and System Sciences*, 48(2):214–230, 1994.

[7] O. Berkman, U. Vishkin. Recursive Star-Tree Parallel Data Structure. *SIAM J. on Computing*, 22(2):221–242, 1993.

[8] Paul Beame, Faith E. Fich. Optimal Bounds for the Predecessor Problem and Related Problems. *J. of Computer and System Sciences*, 65(1):38–72, 2002.

[9] A. Buchsbaum, M. Goodrich and J. Westbrook. Range searching over tree cross products. *Proc. of European Symposium of Algorithms*, pages 120-131, 2000.

[10] R. Cole and R. Hariharan. Tighter upper bounds on the exact complexity of string matching. *SIAM J. on Computing*, 26(3):803–856, 1997.

[11] Martin Farach. Optimal suffix tree construction with large alphabets. *Proc. of the Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[12] M. Farach and S. Muthukrishnan. Perfect hashing for strings: formalization and algorithms. *Proc. of Combinatorial Pattern Matching*, pages 130–140, 1996.

[13] P. Ferragina and R. Grossi. Fast incremental text editing. *Proc. of the Symposium on Discrete Algorithms*, pages 531–540, 1995.

[14] H.N. Gabow, J. Bentley, and R.E. Tarjan. Scaling and related techniques for geometric problems. *Proc. of the Symposium on Theory of Computing*, pages 135-143, 1984.

[15] M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. *Proc. of the Symposium on Discrete Algorithms*, pages 697–704, 1994.

[16] T. Hagerup, P.B. Miltersen and R. Pagh. Deterministic dictionaries. *J. of Algorithms*, 41(1):69–85, 2001.

[17] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13(2):338–355, 1984.

[18] Juha Karkkainen and Peter Sanders. Simple linear work suffix array construction. *Proc. 30th International Colloquium on Automata, Languages and Programming*, pages 943–955, 2003.

[19] D. Knuth, J. Morris and V. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6(2):323–350, 1977.

[20] G.M. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[21] G.M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988.

[22] V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.

[23] R. Lowrance and R. A. Wagner. An extension of the string-to-string correction problem. *J. of the ACM*, 22(2):177–183, 1975.

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

[25] 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*, pages 320–328, 1996.

[26] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. on Computing*, 17(6):1253–1262, 1988.

[27] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. of Molecular Biology*, 147:195–197, 1981.

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

[29] 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.

[30] P. Weiner. Linear pattern matching algorithm. *Proc. of the Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[31] D.E. Willard. Log-logarithmic worst case range queries are possible in space θ(n). *Information Processing Letters*, 17:81-84, 1983.