View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Locally Maximal Common Factors as a Tool for Efficient Dynamic String Algorithms

Amihood Amir[1]
Bar-Ilan University and Johns Hopkins University
amir@cs.biu.ac.il

Itai Boneh
Bar-Ilan University
barbunyaboy2@gmail.com

Abstract

There has been recent interest in dynamic string algorithms, i.e. string problems where the input changes dynamically. One such problem is the longest common factor (LCF) problem. It is well known that the LCF of two strings S and D of length n over a fixed constant-sized alphabet Σ can be computed in time linear in n. Recently, a new challenge was introduced - finding the LCF of two strings in a dynamic setting. The problem is the *fully dynamic one sided LCF (FDOS-LCF) problem*. In the FDOS-LCF problem we get q consecutive queries of the form $< i, \alpha >$, where each such query means: "replace $D[i]$ by α, $\alpha \in \Sigma$ and output the LCF of S and (the updated) D. The goal is to initially preprocess S and D so that we do not need $\mathcal{O}(n)$ time to compute an LCF for each such query.

The state-of-the-art is an algorithm that preprocesses the two strings S and D in time $\mathcal{O}(n \log^4 n)$. Subsequently, the algorithm answers in time $\mathcal{O}(\log^3 n)$ a **single** query of the form: Given a position i on D and a letter α, return an LCF of S and D', where D' is the string resulting from D after substituting $D[i]$ with α. That algorithm is not extendable to multiple queries. In this paper we present a tool - Locally Maximal Common Factors (LMCF) - that proves to be quite useful in solving some restricted versions of the FDOS-LCF problem . The versions we solve are the *Decremental FDOS-LCS problem*, where every change $< i, \alpha >$ is of the form $< i, \omega >$, $\omega \notin \Sigma$, and the *Periodic FDOS-LCS problem*, where S is a periodic string with period length p.

For the decremental problem we provide an algorithm with linear time preprocessing and $\mathcal{O}(\log \log n)$ time per query. For the periodic problem our preprocessing time is linear and the query time is $\mathcal{O}(p \log \log n)$.

2012 ACM Subject Classification Theory of computation → Pattern matching, Theory of computation → Dynamic graph algorithms

Keywords and phrases Dynamic Algorithms, Periodicity, Longest Common Factor, Priority Queue Data Structures, Suffix Tree, Balanced Search Tree, Range Maximum Queries

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.11

[1] Partially supported by ISF grant 571/14

© Amihood Amir and I. Boneh;
licensed under Creative Commons License CC-BY
29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).
Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 11; pp. 11:1–11:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Recently, there has been a growing interest in dynamic pattern matching algorithms. A particular problem that received attention is finding the *longest common factor (LCF)* of two strings. Let S and D be strings of lengths n_1 and n_2, respectively, over a constant size alphabet Σ. Their longest common factor (substring) is the longest string F that is a sunstring of both S and D. The LCF can be computed in $\mathcal{O}(n_1 + n_2)$ time [10].

As mentioned in [13], the LCF problem is not robust and its solution can vary significantly when the input strings are changed even by one letter. It is, therefore, important to study the dynamic instance of the problem, i.e. finding the LCF between two strings after an arbitrary number of changes in one of the two sequences. Other than the purely theoretical interest, the problem has applications in the field of molecular biology. For instance, when we wish to find the LCFs by incorporating the single nucleotide polymorphisms (SNPs) observed in a population in one of the two sequences. The longest common factor with k-mismatches problem has also received much attention recently, in particular due to its applications in bioinformatics [11]. We refer the interested reader to [2, 7, 9, 13].

Recently, a solution to the restricted case, where a single edit operation (substitution, insertion or deletion) is allowed, was presented [1]. In that paper, two strings, S and D, of length n over a finite fixed alphabet Σ, are given. The strings are preprocessed in $\mathcal{O}(n \log^4 n)$ time and $\mathcal{O}(n \log^3 n)$ space. After preprocessing, the answer to a query replacing the symbol in index i of string D by α is computed in $\mathcal{O}(\log^3 n)$ time.

The solution in [1] is not extendable to more substitutions. The goal is solving the **fully dynamic one sided LCF (FDOS-LCF) problem**. Specifically, suppose we get q consecutive queries of the form $< i, \alpha >$, where each such query means: "replace $D[i]$ by α, $\alpha \in \Sigma$ and output the LCF of S and (the updated) D. The goal is to initially preprocess S and D so that we do not need $\mathcal{O}(n)$ time to compute an LCF for each such query.

This problem, as well as many other dynamic string problems, would be efficiently solvable in a straight-forward manner if one could efficiently maintain a suffix tree of a changing string. Alas, a fully dynamic suffix tree seems like a very difficult challenge. In this paper we present a tool – Locally Maximal Common Factors (LMCF) – that proves to be quite useful in solving some restricted versions of the FDOS-LCF problem .

The first problem we solve is the dynamic LCF problem for the *decremental* dynamic model. In this model, every substitution in D is of a symbol $\omega \notin \Sigma$. We provide an algorithm with two implementations. The first does a linear time preprocessing and answers a query in time $\mathcal{O}(\log n)$. It is presented for pedagogical reasons, to describe the power of the LMCF idea in a simple manner. The second implementation uses data structures such as the van Emde Boaz tree, or y-fast tries that enable solving the problem in time $\mathcal{O}(\log \log n)$ per query, following a linear time preprocessing.

The second problem we solve is the FDOS-LCF problem in the special case where S is periodic. Our solution has a linear time preprocessing and subsequent $\mathcal{O}(p \log \log n)$ time queries.

2 Preliminaries

We begin with basic definitions and notation generally following [4]. Let $S = S[1]S[2]...S[n]$ be a *string* of length $|S| = n$ over a finite ordered alphabet Σ of size $|\Sigma| = \mathcal{O}(1)$. By ε we denote an empty string. For two positions i and j on S, we denote by $S[i\,..\,j] = S[i]\,..\,S[j]$ the *factor* (sometimes called *substring*) of S that starts at position i and ends at position j (it equals ε if $j < i$). We recall that a prefix of S is a factor that starts at position 1

($S[1\,..\,j]$) and a suffix is a factor that ends at position n ($S[i\,..\,n]$). We denote the reverse string of S by S^R, i.e. $S^R = S[n]S[n-1]\ldots S[1]$. We denote the concatenation of two strings, S_1 and S_2, by S_1S_2.

Let Y be a string of length m with $0 < m \leq n$. We say that there exists an *occurrence* of Y in S, or, more simply, that Y *occurs in* S, when Y is a factor of S. Every occurrence of Y can be characterised by a starting position in S. Thus we say that Y occurs at the *starting position* i in S when $Y = S[i\,..\,i+m-1]$.

Given two strings S and D, a string Y that occurs in both is a longest common factor (LCF) of S and D if there is no longer factor of D that is also a factor of S; note that S and D can have multiple LCF strings. We introduce a natural representation of an LCF of S and T as a triple (m, p, q) such that $S[p\,..\,p+m-1] = T[q\,..\,q+m-1]$ is an LCF of S and T. The *decremental dynamic LCF* problem is formally defined as follows;

Decremental Dynamic LCF
Input: Two strings S and D of length n over an alphabet Σ, symbol $\omega \notin \Sigma$. Let $< i_1, \omega >, < i_2, \omega >, ..., < i_k, \omega >$ be a sequence of substitution operations in D, and let D' be the result of these k substitutions.
Output: An LCF of S and D'.

Clearly, the problem can be solved by computing an LCF after every change. We will see that such a computation can be done in linear time. We show an algorithm whose preprocessing time is linear, and where an LCF computation after each substitution can be done more efficiently. In Section 4 we present an implementation whose query complexity is logarithmic. In Section 5 we present an implementation whose query processing time is $\mathcal{O}(\log\log n)$.

3 Algorithm's Idea

We need some additional tools and definitions:

▶ **Definition 1.** Let S and D be two strings of length n over fixed finite alphabet Σ. A *locally maximal common factor (LMCF)* of S in D is a factor $D[i..j]$ of D that satisfies the following two conditions:

1. $D[i..j]$ is a factor of S.
2. Neither $D[i..j+1]$ nor $D[i-1..j]$ are factors of S.

The following observations are crucial.

▶ **Observation 2.** *An LCF of two strings S and D is an LMCF of S in D. Moreover, a longest LMCF of S in D is an LCF.*

▶ **Observation 3.** *There are at most n LMCF's of S in D.*

Proof. It is clear that only one LMCF can start at any index i of D. Otherwise, let i be an index such that both $D[i..j]$ and $D[i..\ell]$ are LMCF's, and wlog assume $j < \ell$. Then $D[i..j+1]$ is a factor of S, contradicting $D[i..j]$'s maximality.
Since D is of length n there are no more than n indices where an LMCF can start. ◀

▶ **Observation 4.** *Let $D[i_1..j_1]$ and $D[i_2..j_2]$ be LMCF's of S in D. Then $i_1 < i_2$ iff $j_1 < j_2$.*

Proof. Otherwise, one is contained in the other contradicting the fact that both are LMCF's. ◀

To achieve our goal, we will show that we can update and maintain a sorted list of LMCF's in logarithmic time per substitution. One more tool is needed.

3.1 The Suffix Tree

The *suffix tree* $\mathcal{T}(S)$ of a non-empty string S of length n is a compact trie representing all suffixes of S. The branching nodes of the trie as well as the terminal nodes, that correspond to suffixes of S, become *explicit* nodes of the suffix tree, while the other nodes are *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We let $\mathcal{L}(v)$ denote the *path-label* of a node v, i.e., the concatenation of the edge labels along the path from the root to v. We say that v is path-labelled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v. Node v is a *terminal* node if its path-label is a suffix of S, that is, $\mathcal{L}(v) = S[i\,..\,n]$ for some $1 \leq i \leq n$; here v is also labelled with index i. It should be clear that each factor of S is uniquely represented by either an explicit or an implicit node of $\mathcal{T}(S)$, called its *locus*. In standard suffix tree implementations, we assume that each node of the suffix tree is able to access its parent. Once $\mathcal{T}(S)$ is constructed, it can be traversed in a depth-first manner to compute the string-depth $\mathcal{D}(v)$ for each node v. For every two nodes, v, u, where $\mathcal{L}(u) = \alpha\mathcal{L}(v),\ \alpha \in \Sigma$, there is a pointer from v to u called a *suffix link*. It is known that the suffix tree, with the suffix links, of a string of length n, over a fixed-sized ordered alphabet, can be computed in time and space $\mathcal{O}(n)$ [16, 12, 14]. The suffix tree of an integer alphabet can also be built in linear time [5].

3.2 Preprocessing

Our algorithm has two parts: a preprocessing part, to set up the data structures; and a query part that maintains the data structures upon every substitution in a manner allowing to efficiently pull a maximum LMCF at every instant.

Finding All LMCF's:

The first preprocessing step is finding all LMCF's. This is easily done via the suffix tree. Algorithm $AllLMCF(S, D)$ computes, for each index i, the longest factor starting at $D[i]$ that occurs in S. Then, starting from $D[1]$, output every factor whose ending position is farther than the previous one.

Algorithm 1: Algorithm AllLMCF(S,D) – Finding all LMCF's

1. Construct suffix tree $\mathcal{T}(S\$D)$ for the string $S\$D$, where $\$ \notin \Sigma$.
2. For every leaf, indicate whether it represents a suffix starting in S or a suffix starting in D.
3. Traverse $\mathcal{T}(S\$D)$ (using, e.g., DFS) and mark as *blue* every node that has in its subtree at least one leaf from D and one leaf from S.
4. For every leaf representing suffix $D[i]$, mark the lowest blue node on its path from the root, i.e. the ending index of the longest factor that starts at $D[i]$ and appears in S. Denote it by $LD[i]$.
5. Set F to be $LD[1]$. $\{F$ will be the farthest common factor so far. $\}$
   $< 1, LD[1] >$ is an LMCF.
6. For i=2 to n do:
   If $LD[i] > F$ then $< i, LD[i] >$ is an LMCF and $LD[i]$ is the new value of F.
   endFor

Correctness: The algorithm simply follows the definition of LMCF.

Time: The construction and size of the suffix tree, as well as the tree traversals, are done in time $\mathcal{O}(n)$.

Let us now examine the effects on the LMCF's of a substitution of ω for the symbol in index k of D. Clearly, all LMCF's that end before index k and all those that start after index k are not affected. Let CM_k be the set of LMCF's $D[i..j]$ such that $i \leq k \leq j$. Since $\omega \notin S$ then each of the LMCF's in CM_k is cut at index k. But because of local maximality, for any two LMCF's $D[i_1..j_1]$ and $D[i_2..j_2]$ where $i_1 < i_2 \leq k \leq j_1 < j_2$, only $D[i_1..k-1]$ and $D[k+1..j_2]$ are potentially LCMF's after the substitution in k. We can conclude:

▶ **Observation 5.** *After a replacement of ω in index k of D, if CM_k is empty, there is no change to the LMCF's. Otherwise, all LMCF's that are elements in CM_k should be deleted and the following two strings should be inserted to the set of LMCF's:*

$L_1 = D[i'..k-1]$, *where* $i' = \min\{i | D[i,j] \in CM_k\}$

$L_2 = D[k+1..jw]$, *where* $j' = \max\{j | D[i,j] \in CM_k\}$

We now have an idea of what our algorithm should look like. We need a data structure that allows us to efficiently delete the appropriate sets of LMCF's, to add the two new LMCF's, if necessary, and to efficiently find the maximum. In particular, let LCMF $D[i..j]$ be represented by the pair $< i, j >$. We need a data structure that supports the following operations:

1. Costruct LMCF structure
2. Given index k, Delete CM_k
3. Insert LMCF $< i, j >$
4. Find maximum length LMCF.

4 Implementation 1: $\mathcal{O}(\log n)$ Query Processing

Consider a balanced binary search tree T of the pairs $< i, j >$ that represent LMCF's, sorted by the smaller index in every pair. Given a substitution index k, we can efficiently find the subtree of all indices i for which $i \leq k$. The problem is that some of the LMCF's in this subtree are not cut by k, i.e. their second index j has $j < k$, thus they should not be deleted. However, Observation 4 guarantees that the second indices of all LMCF's are *sorted by the same order as the first indices*. This means that the binary search tree allows us to find i_1, the smallest i for which $D[i,j]$ is cut by k. Similarly, we can find i_2, the largest i for which $D[i,j]$ is cut by k. We delete all the LMCF's starting between i_1 and i_2 and add L_1 and L_2. We will say a few words about maintaining the tree balanced.

As for the maximum. For each node in the tree write the maximum length LMCF in its subtree. Updating the maximum after every change means running up the tree, thus the time is $\mathcal{O}(\log n)$. Answering a query means starting from the root and running down the tree towards the maximum, again having time $\mathcal{O}(\log n)$.

Implementation Details: There are $\mathcal{O}(n)$ LMCF's in total, and each one is a pair $< i, j >$ starting at a unique i, $1 \leq i \leq n$. Therefore, we use the balanced binary tree of $\{1, ..., n\}$, but for every node we indicate (a) whether the node is "full", i.e., whether there is an LMCF that starts at the node of index i, and (b) whether the subtree rooted at node i is empty. Since the height of the tree is $\mathcal{O}(\log n)$, the search is bounded by the tree height even in the tree with "empty" nodes.

Time:

1. **Preprocessing:** The balanced tree for $\{1, ..., n\}$ can be constructed in linear time. The LMCF's can be found on the suffix tree as shown in Subsection 3.2 in linear time. The appropriate nodes are marked and denoted "full" and the rest of the nodes as well as the appropriate subtrees are marked "empty" in linear time using DFS. Every LMCF node also has a "length" field and every subtree maintains in its root the maximum length in the subtree. This is also done by DFS in linear time.
2. **Deleting CM_k:** The j's of the $< i, j >$ LMCF's are sorted in the same order as the i's. Thus finding the largest and smallest i for which $D[i, j]$ includes k is done in time $\mathcal{O}(\log n)$. The appropriate subtrees are marked "empty". Because the tree is balanced this requires only $\mathcal{O}(\log n)$ nodes to be marked. The appropriate maximum length fields along the modified paths are also updated appropriately, also in time $\mathcal{O}(\log n)$.
3. **Inserting LMCF $< i, j >$:** simply turn on the appropriate "full" field of node i in the tree, as well as the appropriate j and length. Walk up the path updating the subtree emptiness indicators and the maximum subtree length.
4. **Finding Maximum Length LMCF:** The maximum length appears in the subtree max length field of the root, and can be output in constant time. All LCF's can be found by going down the tree. The time is then $\mathcal{O}(tocc \log n)$, where $tocc$ is the number of LCF's. A single representative LCF can be found in time $\mathcal{O}(\log n)$.

5 Implementation 2: $\mathcal{O}(\log \log n)$ Query Processing

The implementation we suggested to our algorithm was based on a balanced search tree. The height of such a tree is generally $\mathcal{O}(\log n)$. However, the are data structures that allow searching in time $\mathcal{O}(\log \log u)$, where u is the size of the universe of keys [15, 17]. Our set of keys is $\{1, ..., n\}$, which would make the search time $\mathcal{O}(\log \log n)$. In this section we describe an implementation that uses a data structure that supports the following operations on a set $S \subset \{1, ..., n\}$: *insert, delete, lookup, findnext, findprev.* Using such a data structure we show how to implement the four operations described in Section 3.

Constructing the tree, searching for a key, and inserting a constant number of LMCF's, can be done naturally on the vEB tree as well as the y-fast trie. The challenge is implementing the deletion of CM_k and the maintenance of the maximum lengths. The difficulty arises in the fact that these trees don't have a constant number of children per node, as does a balanced search tree, thus these updates are more complex.

5.1 Data Structures

As mentioned, we assume a data structure that supports the following operations on a set $S \subset \{1, ..., n\}$: *insert, delete, lookup, findnext, findprev.* We call this a *fast tree.* The vEB tree [15] and the y-fast trie [17] can achieve this in space $\mathcal{O}(n)$ and time per operation $\mathcal{O}(\log \log n)$. Nevertheless, our algorithm can use any dynamic priority queue with predecessor and successor query capability.

In order to implement our four operations, we will need a number of fast trees. We define them below.

▶ **Definition 6.**

1. The *LMCF tree* is a fast tree of the pairs $< i, j >$ that represent LMCF's, sorted by the smaller index in every pair.

2. The *Interval tree* is a fast tree sorted by the starting indices of *valid* intervals in the range $\{1, ..., n\}$. Each such element also holds its length. Initially, the entire range is valid, so there is a single element starting at the first index and whose length is n, the length of the entire range. As our algorithm progresses, we may need to delete entire intervals of the range, in which no LMCF's start. We implement it by "cutting" them out of the interval tree.
3. The *max length* tree is a fast tree whose entries are the maximum length of the LMCF's in the valid intervals. There is also a link between each valid interval entry in the interval tree and the entry of its length in the max length tree. The max length tree is sorted by the length.

Our algorithm makes use of the *range maximum query* algorithm. The problem is defined as follows:

▶ **Definition 7.** The *Range Maximum Query (RMQ)* problem has as its input an array $A[1..n]$ of natural numbers. We wish to preprocess the array in a manner that will enable efficient solution to:

Query: Given $[i, j]$, where $1 \leq i \leq j \leq n$. Return index k, $i \leq k \leq j$ such that $A[k] \geq A[\ell]$, $\forall \ell$ s.t. $i \leq \ell \leq$ æ.

Time: It was shown [8, 3, 6] that the RMQ problem can be solved using linear time and space preprocessing and constant query time.

Space: Since both the fast tree, and the RMQ preprocessing are done in linear space, our constructions uses linear space.

5.2 The Algorithm

We show the implenmentation of each of our operations using the fast trees.

Preprocessing - Construct the fast trees: We find the LMCF pairs as in Implementation 1 in Section 4. We construct two sets of fast trees: (a) $MFT1$: sorted by the starting position of the LMCF's, along with the length of each LMCF. The lengths are entered into an M array which is preprocessed for RMQ queries. The length of the m array is n. $M[i]$ gets the value of the LMCF that starts in $D[i]$, if such an LMCF exists, otherwise $M[i]$ gets the value $-\infty$. The Interval and the max length fast trees are initialized. (b) $MFT2$ Sorted by the ending positions of the LMCF's. Each entry has its length. It comes with its accompanying interval and max length fast trees. $MFT2$ is symmetric to $MFT1$ and thus in the rest of the paper we will describe operations on $MFT1$. Similar operations are symmetrically done in $MFT2$.

Time: Construction time of the fast trees, and RMQ preprocessing: $\mathcal{O}(n)$.

Given Index k, Delete CM_k: Computing CM_k is straightforward in the LMCF fast trees. For any k, let j_1 be the smallest entry larger than k in the $MFT2$ set of trees. Initially, it is found in the LMCF tree. Subsequently, it is checked in the LMCF tree in the closest valid interval, as found in the interval fast tree. j represents an LMCF $< i_1, j_1 >$. If $i_1 < k$ then i_1 is the starting position of the LMCF with the smallest index that is cut by k. Therefore

all LMCF's starting between indices i_1 and $k-1$ should be removed and replaced by the new LMCF $< i_1, k-1 >$. Now, let i_2 be the largest entry smaller than k in $MFT1$. It represents an LMCF $< i_2, j_2 >$. If $j_2 > k$ then a new LMCF $< k+1, j_2 >$ needs to be added. Deleting an entry from a fast tree is simple. However, we need to efficiently delete many entries, as well as maintain the maximum. As mentioned before, we describe how to handle $MFT1$, the operations on $MFT2$ are symmetrical.

Recall that the interval fast tree is initialized to the entire range. Assume that $[i_1, k]$ is the first interval to be deleted, then the interval fast tree will have a node starting at the beginning of the interval and ending at $i_1 - 1$, a node starting at $k+1$ and ending at the end of the interval, and a node of length 1 at location i_1. In general, the interval fast tree only has non-overlapping intervals. Additionally, since LMCF's are cut at the insertion point, the following holds.

▶ **Observation 8.** *An interval is deleted from the interval fast tree only if it is entirely contained in a previous valid interval.*

Another crucial observation is the following:

▶ **Observation 9.** *Only intervals of length* 1 *(points) may be added and deleted in an interval that was declared "invalid".*

From the above two observations we get:

▶ **Conclusion 10.** *The interval tree is composed of intervals and points. Once an interval is deleted, the activity in the entire deleted interval consists only of adding and deleting single points.*

The deletion of an lnterval requires updating the maximum lengths of the remaining LMCF's. If the interval was a point, this is a single operation on the max length fast tree on the path of the change. If CM_k is an interval, then it caused a range change in the interval fast tree. We need to delete from the max length fast tree the maximum length of the range that is cut, and add the max length LMCF in the shortened range, as well as $k - i_1$, the length of the new LMCF. Note that because of Conclusion 10 the only non-point interval changes are the results of cuts in the initial ranges. But the initial ranges were preprocessed for RMQ queries. Consequently, we can, in time $\mathcal{O}(1)$ update the max length fast tree.

Time: Handling points clearly takes time $\mathcal{O}(\log\log n)$ since these are regular fast tree operations. Deleting an interval of LMCF's consists of a fast tree operation. which takes time $\mathcal{O}(\log\log n)$. Similarly, updating the max length fast tree takes time $\mathcal{O}(\log\log n)$, for a total of $\mathcal{O}(\log\log n)$ time.

Insert LMCF: Done at the LMCF tree and each of the interval and max length trees.

Time: $\mathcal{O}(\log\log n)$ on the fast trees.

Find Maximum Length LMCF: Find the maximum element in the root max length fast tree.

Time: $\mathcal{O}(\log\log n)$.

6 Dynamic LCF for a Static Periodic String

The LMCF's are an efficient tool for handling other dynamic versions of the problem. Our next result is an algorithm for the fully dynamic case. The changes to the text may replace a character in index i of D with some other character in Σ. We still assume that S is static and D is dynamic, however, we assume that S is a periodic string whose period length is p.

▶ **Definition 11.** Let S be a string of length n. S is called *periodic* if $S = P^i\text{pref}(P)$, where $i \in \mathbb{N},\ i \geq 1,\ P$ is a substring of S such that $|P| < n$, P^i is the concatenation of P to itself i times, and $\text{pref}(P)$ is a prefix of P. The smallest such substring P is called the *period* of S. If S is not periodic it is called *aperiodic*.

▶ Remark. Throughout the paper we use p to denote a period length and P the period string, i.e., $p = |P|$.

Formally our problem is:

Periodic Dynamic LCF
Input: Two strings S and D of length n over an alphabet Σ, S is periodic with period length p. Let $< i_1, \sigma_1 >, < i_2, \sigma_2 >, ..., < i_k, \sigma_k >$ be a sequence of substitution operations in D, where the symbol $D[i_j]$ is replaced by $\sigma_j \in \Sigma$, $j = 1, ..., k$, and let D' be the result of these k substitutions.
Output: An LCF of S and D'.

Our algorithm has linear preprocessing time and takes time $\mathcal{O}(p \log\log n)$ for a substitution and LCF query.

6.1 Algorithm's Idea

The periodic static string algorithm also maintain the LMCF's and their maximum length, as the deremental algorithm did. In order to limit the maintenance time at every substitution, we need to prove some properties of LMCF's in a periodic string.

▶ **Observation 12.** *Every substring of S whose length is larger than p also has a period of size p. In particular this, of course, applies to the LMCF's of S in D.*

▶ **Lemma 13** (Periodicity unity)**.** *Let $D_1 =< i_1, j_1 >$ and $D_2 =< i_2, j_2 >$ be two LMCFs of S in D. If the length of the overlap of D_1 and D_2 is at least p then $D_1 = D_2$ ($i_1 = i_2$ and $j_1 = j_2$).*

Proof. Let $D_1 = D[i_1..j_1]$ and $D_2 = D[i_2..j_2]$ be two LMCF's of S in D s.t the overlap of D_1 and D_2 is an interval of at least p characters. We assume, wlog, that $i_1 \leq i_2$. That means that $i_1 \leq j_1 - p$, $i_2 \leq j_2 - p$ and, because the overlap is of length at least p, $i_2 \leq j_1 - p$. As substrings of S, both D_1 and D_2 have a period of size p. Let $S[i_3..j_3]$ be an instance of D_1 in S. According to the local maximum property of D_1 - it must be satisfied that $D[j_1 + 1] \neq S[j_3 + 1]$. Otherwise D_1 could have been extended. S has a period of size p so $S[j_3 + 1] = S[j_3 + 1 - p]$. The index $j_1 + 1 - p$ is still within the range of D_1 because $i_1 \leq j_1 - p$. so $S[j_3 + 1 - p] = D[j_1 + 1 - p]$. The index $j1 + 1 - p$ is also within the Range of D_2 because $i_2 \leq j_1 - p$. **Assuming that** $D_1 \neq D_2$, The index $j_1 + 1$ is within the range of D_2 as well because $j_2 > j_1$ (Otherwise, D_2 is fully contained in D_1). On top of all, D_2 is a common factor of S that is larger than p. so it has a period of size p as well and it satisfies: $D[j_1 + 1 - p] = D[j_1 + 1]$. According to transitivity : $D[j_1 + 1] = S[j_3 + 1]$, in contradiction to D_1's local maximum property. ◀

▶ **Lemma 14** (Periodicity singular extention). *Let $D_1 = D[i..j]$ be a common factor of D and S of length greater than p. Then $D[i..j+1]$ is also a factor of S iff $D[j+1] = D[j+1-p]$.*

Proof. $\Rightarrow$ Assume $D[i..j+1]$ is a factor of S. Its length is greater than p, therefore it has a period of size p. D_1 has length greater than p so $j-p+1$ is within its range. From the indexes presence in the interval and the period we get : $D[j+1] = D[j+1-p]$.

$\Leftarrow$ Assume $D[j+1] = D[j+1-p]$. Let $S[i_3..j_3]$ be an insance of D_1 in S. With the same reasoning as in the proof of Lemma 13 we get that $D[j+1-p] = S[j_3+1-p] = S[j_3+1]$. The final conclusion is derived from transitivity. ◀

Note: Both proofs assume that there is an index j_3+1 in S, which is not necessarily true. However, every substring starting after the first p letters of S is equal to a substring that begins in the first p symbols, and thus there is an instance that can be extended to j_3+1. There are only at most p possible substrings where this shift can not be done - those that already start within the first p symbols of S and extend all the way to the end. The lemmas don't hold for these strings, but there are only at most p of them and they are handled separately by the algorithm.

The above lemmas indicate that given a change in index x in D, the number of LMCF's that are affected by this change and are "far" from x is small. "Far" means starting or ending in an index whose distance from x exceeds $\mathcal{O}(p)$. The algorithm will handle "far" LMCF's and "close" LMCF's separately. There are only $\mathcal{O}(p)$ "close" LMCF's, thus they can be handled in a brute force manner and still cost only $\mathcal{O}(p)$ per query. The two lemmas guarantee a constant number of affected "far" LMCF's, so they also are handled efficiently.

7 The algorithm

Preprocessing: The preprocessing stage consists of finding all of the LMCF's, and putting them in a convenient data structure. The LMCF's are found using algorithm $AllLMCF(S, D)$, as presented in Subsection 3.2. The LMCF's are put in an efficient dynamic priority queue data structure, (e.g. the fast tree mentioned above) containing the indexes, sorted by the i value. Additionally, each node contains extra information about the maximum value of $j-i$ in the subtree rooted in this node. Denote this priority queue by T.

Handling an Edit Operation: Assume a change is made at location x of D. We can apply algorithm $AllLMCF$ on the area $D[x-p..x+p]$ and, in time $\mathcal{O}(p)$ get all the LMCF's starting in that area and ending in $D[x+p]$. This is almost all we need. The only corrections necessary are: (1) LMCF's that started before $D[x-p]$ and extended past $D[x]$; (2) LMCF's that started before $D[x-p]$ and ended at $D[x-1]$ (maybe they need to be extended); and (3) new LMCF's that start at the interval $D[x-p..x]$ and extend past $D[x+p]$.

Because of the Periodicity Unity Lemma, we know that there is at most one LMCF that starts before $D[x-p]$ and reaches to or past $D[x-1]$. If it passed $D[x]$ (Case (1) above), its endpoint should be replaced by $x-1$. If it ends at $x-1$ (Case (2) above), then its endpoint should be extended. We can spend p time to see how much ahead it can be extended. If it can be extended by more than p positions, then by the Periodicity Unity Lemma, we can merge it with the previous LMCF that started at $D[x+1]$. This leaves us with Case (3) - all LMCF's that start at the interval $D[x-p..x]$ and extend past $D[x+p]$. Again, due to the Periodicity Unity Lemma, there is at most one such LMCF. We merge it with the old LMCF that started at $D[x+1]$. All old LMCF's that started in $D[x-p..x]$ are deleted. The total number of changes is $\mathcal{O}(p)$ and, for each change the maximum length in the subtree is

Algorithm 2: pseudocode for algorithm UpdateLMCF

Algorithm UpdateLMCF(x,α) – substitute location $D[x]$ with α

1. Update D : $D[x] \leftarrow \alpha$.
2. Find a pair $\pi_1 =< i_1, j_1 >$ in T that satisfies $j_1 \geq x-1$ and $i_1 \leq x-p$. If there isn't one: $\pi_1 \leftarrow nill$.
3. Find a pair $\pi_2 =< i_2, j_2 >$ in T that satisfies $i_2 \leq x+1$ and $j_2 \geq x+p$. If there isn't one: $\pi_2 \leftarrow nill$.
4. Set two binary flags $f_i, i \in \{1,2\}$. $f_i = 1 \iff \pi_i = nill$.
5. For $c = 0, 1, 2...p$ do:
   a. If $D[x+c] \neq D[x+c-p]$ and $f_1 = 0$: Remove π_1 from T and add the pair $< i_1, x+c-1 >$. Then set $f_1 = 1$.
   b. If $D[x-c] \neq D[x-c+p]$ and $f_2 = 0$: Remove π_2 from T and Add the pair $< x-c+1, j_2 >$. Then set $f_2 = 1$.
6. If both flags are 0 in the end of the loop: Remove both π_1 and π_2 from T and add $< i_1, j_2 >$.
7. Remove from T all the pairs $< i, j >$ s.t $i \geq x-p$ and $j \leq x+p$.
8. Use $ALLLMCF(S[1..p]^3, D[x-p..x+p])$ to get all the LMCF's contained in this interval. Add all the new LMCF's found to T. Except the one with the minimal i value and the one with the maximal j value, denoted as π_{left} and π_{right} respectively.
9. Check if there is an LMCFs that contains π_{right} in T (smaller i value and greater j value). If there is not then add π_{right} to T. Do the same for π_{left}.

end Algorithm

maintained in $\mathcal{O}(\log\log n))$time, for the fast tree data stucture used. A pseudocode of the algorithm can be found in Algorithm 2.

7.1 Correctness

As previously indicated, the algorithm handles two types of LMCF's separately - "far" LMCFs and "close" ones. A far LMCF is an LMCF that is cut (or touched) by x, the location of the edit operation, but the diffenece between x and either i or j is at least p. An important observation is that according to the Periodicity Unity Lemma, there is no more than one such possible LMCF from each side of x (left and right).

Consequently, our algorithm finds the, possibly, single "far" LMCF from each side of x and figures out how it should be modified after D has changed.

Another useful property of the "far" LMCF's is that even if the edit operation cuts them - they are still at least of size p in the updated D. That property enables the use of the Singular Extention Lemma to check how far they extend in the modified D.

The final observation to be made in dealing with the "far" LMCF's is that checking $p+1$ matches after (or before) x is enought. If a mismatch was found - that's as far as the LMCF can extend (Singular Extention Lemma). If the p first letters after x match, that means that there were "far" LMCFs from both sides of x, and that after the edit operation, they overlap within an interval greater than p. That makes them the same LMCF due to the Periodicity Unity Lemma. At that point the algorithm will stop checking symbol by symbol. Rather, it will combine the two "far" LMCF's.

Handling the close LMCF's is done in straightforward way - using $AllLMCF$ on the small interval in wich "close" LMCF's can be found. It is easy to observe that they must be a substring of $S[1..p]^3$.

The algorithm, witout any modifications, accually answers a slighly different question from the one asked. It finds the LCF of the dynamic string D and some infinite period of the first p letters of S. if S is in size $|D| + p$, the questions are equivalent. Any other size of S will bring into play the issue mentioned in the note that follows the proof of the two lemmas in Subsection 6.1. For simplicity's sake we presented the algorithm with the assumption of the appropriate length of S. However, a slight adjustment, that can be done without changing the time complexity, can solve this problem. Every time we add to T an LMCF larger than $n - p$, wich can only happen twice in a single change, we should check if it is an actual factor of S and fix it if its not. This can be done by locating the first instance in S of the new LMCF and use its known size to detect if it is accually a factor. If it is not - then the starting index of S in the LMCF and its length can be used to deduce the way it should be partitioned, in $\mathcal{O}(1)$ time.

7.2 Complexity

Lines 2 and 3 are standard priority queue searches. Since the size of the LMCF collection is bounded by n, they take $\mathcal{O}(\log\log n)$ time.
The loop in line 5 repeats at most p times. The operations in every repeat are a constant amount of symbol comparison or priority queue manipulations, for a total of $\mathcal{O}(p + loglogn)$.
Line 6 is also a priority queue manipulation.
Line 7 requires a few priority queue manipulations. There are no more than $\mathcal{O}(p)$ LMCF's starting at that area so the total time complexity is $\mathcal{O}(p \log\log n)$.
Line 8 uses $AllLMCF$ on two inputs of size $\mathcal{O}(p)$. that has $\mathcal{O}(p)$ time complexity. Adding the $\mathcal{O}(p)$ new LMCF's to the priority queue takes time $\mathcal{O}(p \log\log n)$.
Line 9 makes a constant amount of balanced tree searches and additions for a total of $\mathcal{O}(\log\log n)$.
Every change or addition causes a percolation up of the changes in the maximum LMCF length in the subtree. In a balanced search tree, this is $\mathcal{O}(\log\log n)$ time per change.

Total Query Time: $\mathcal{O}(p \log\log n)$.

8 Conclusions

We have presented a tool - the LMCF - that enables efficient solutions to two variants of the dynamic longest common factor problem. In both variants we have a static string S and a dynamic string D. For the decremental case, i.e. where symbols of D are substituted by a new symbol not in the alphabet, the update time is $\mathcal{O}(\log\log n)$, and for the case where S is periodic with period p, the update time is $\mathcal{O}(p \log\log n)$. In both cases the preprocessing is linear.

Our algorithm is designed for strings over a constant-sized alphabet. However, with a $\mathcal{O}(n \log n)$ pre-sorting of the strings, and converting to the alphabet $\{1, ..., n\}$, the same algorithms will apply.

An open question is to extend this result to a fully dynamic case, that is, to propose a data structure that allows subsequent edit operations on one or both of the strings S and D for a general S, and reports the LCF after each operation in an efficient time complexity. Of course the ultimate challenge is a fully dynamic suffix tree algorithm. That problem seems hard. In the meanwhile it is important to consider dynamic versions of specific pattern matching problems. We believe that the LMCF idea can prove useful in other dynamic string algorithms as well.

References

1. A. Amir, P. Charalampopoulos, C.S. Iliopoulos, S.P. Pissis, and J. Radoszewski. Longest common factor after one edit operation. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS, pages 14–26. Springer, 2017. best paper award.
2. M.A. Babenko and T.A. Starikovskaya. Computing the longest common substring with one mismatch. *Probl. Inf. Transm.*, 47(1):28–33, 2011.
3. O. Berkman and U. Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–229, 1994.
4. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
5. M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.
6. J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 4009 in LNCS, pages 36–48. Springer-Verlag, 2006.
7. T. Flouri, E. Giaquinta, K. Kobert, and E. Ukkonen. Longest common substrings with k mismatches. *Information Processing Letters*, 115(6-8):643–647, 2015.
8. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing*, 67:135–143, 1984.
9. S. Grabowski. A note on the longest common substrings with k mismatches problem. *Information Processing Letters*, 115(6-8):640–642, 2015.
10. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
11. C.-A. Leimeister and B. Morgenstern. KMACS: the k-mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014.
12. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.
13. T. Starikovskaya. Longest common substrings with approximately k mismatches. In *Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPIcs*, pages 21:1–21:11, 2016.
14. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
15. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
16. P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.
17. D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.