Journal of Computer and System Sciences 80 (2014) 1245–1253

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Range LCP

Amihood Amir [a,b,1], Alberto Apostolico [c,d,2], Gad M. Landau [e,f,3], Avivit Levy [g,*,4], Moshe Lewenstein [a], Ely Porat [a,5]

[a] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
[b] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, United States
[c] College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30318, United States
[d] Dipartimento di Ingegneria dell' Informazione, Università di Padova, Via Gradenigo 6/A, 35131 Padova, Italy
[e] Department of Computer Science, University of Haifa, Mount Carmel, Haifa 31905, Israel
[f] Department of Computer Science and Engineering, Polytechnic Institute of New York University, 6 Metrotech Center, Brooklyn, NY 11201, United States
[g] Department of Software Engineering, Shenkar College, 12 Anna Frank, Ramat-Gan, Israel

ARTICLE INFO

Article history:
Received 29 January 2013
Received in revised form 21 January 2014
Accepted 5 February 2014
Available online 14 February 2014

Keywords:
Data structures
LCP
Pattern matching

ABSTRACT

In this paper, we define the *Range LCP* problem as follows. Preprocess a string S, of length n, to enable efficient solutions of the following query: Given $[i, j]$, $0 < i \leqslant j \leqslant n$, compute $\max_{\ell,k\in[i..j]} LCP(S_\ell, S_k)$, where $LCP(S_\ell, S_k)$ is the length of the longest common prefix of the suffixes of S starting at locations ℓ and k. This is a natural generalization of the classical LCP problem. We provide algorithms with the following complexities:

1. Preprocessing Time: $O(|S|)$, Space: $O(|S|)$, Query Time: $O(|j-i|\log\log n)$.
2. Preprocessing Time: none, Space: $O(|j-i|\log|j-i|)$, Query Time: $O(|j-i|\log|j-i|)$. However, the query just gives the pairs with the longest LCP, *not* the LCP itself.
3. Preprocessing Time: $O(|S|\log^2|S|)$, Space: $O(|S|\log^{1+\varepsilon}|S|)$ for arbitrary small constant ε, Query Time: $O(\log\log|S|)$.

* Corresponding author.
E-mail addresses: amir@cs.biu.ac.il (A. Amir), axa@cc.gatech.edu (A. Apostolico), landau@cs.haifa.ac.il (G.M. Landau), avivitlevy@shenkar.ac.il (A. Levy), moshe.lewenstein@gmail.com (M. Lewenstein), porately@cs.biu.ac.il (E. Porat).

[1] Partly supported by NSF grant CCR-09-04581, ISF grant 347/09, and BSF grant 2008217.
[2] Partly supported by BSF grant 2008217.
[3] Partly supported by the National Science Foundation Award 0904246, Israel Science Foundation grant 347/09, Yahoo, grant No. 2008217 from the United States–Israel Binational Science Foundation (BSF) and DFG.
[4] Partly supported by the Israel Science Foundation grant 347/09.
[5] Partly supported by BSF grant 2006334, ISF grant 1484/08, and Google Award.

http://dx.doi.org/10.1016/j.jcss.2014.02.010
0022-0000/

1. Introduction

The Longest Common Prefix (LCP) has been historically an important tool in Combinatorial Pattern Matching:

1. The connection between Edit Distance and Longest Common Prefix (LCP) calculation has been shown and exploited in the classic Landau–Vishkin paper in 1989 [14]. It was shown in that paper that computing mismatches and LCPs is sufficient for computing the Edit Distance.
2. The LCP is the main tool in various Bioinformatics algorithms for finding maximal repeats in a genomic sequence.
3. The LCP plays an important role in compression. Its computation is required in order to compute the Ziv–Lempel compression, for example [15].

Therefore, the LCP has been amply studied and generalized versions of the problem are of interest. Generalizations of classical problems shed light on the original problem and lead to a better understanding of it, but also enable to develop new, usually stronger, tools for the solution of the problem. Generalization is, therefore, a classical approach in the study of algorithms.

A first natural generalization of the LCP problem is a "range" version. Indeed, "range" versions of the LCP problem were considered in the literature. Cormode and Muthukrishnan [5] consider a version they call the *Interval Longest Common Prefix* (*ILCP*) *Problem*. In that version, the maximum LCP between a given suffix and all suffixes in a given interval, is sought. They provide an algorithm whose preprocessing time is $O(|S|\log^2|S|\log\log|S|)$, and whose query time is $O(\log|S|\log\log|S|)$. This result was then improved by [12] to $O(|S|\log|S|)$ preprocessing time and $O(\log|S|)$ query time. Ilie and Tinta [9] consider a different "range" version, where a set of pairs of suffixes is given as input and the pair that has the maximum LCP is sought.

This paper provides efficient algorithms for a more general version of the Range LCP problem.

Problem definition. The formal definitions of LCP and the range LCP problem are given below.

Definition 1. Let $A = A[1]\cdots A[n]$ and $B = B[1]\cdots B[n]$ be strings over alphabet Σ. The *Longest Common Prefix* (*LCP*) of A and B is the empty string if $A[1] \neq B[1]$. Otherwise it is the string $a_1\cdots a_k$, $a_i \in \Sigma$, $i = 1,\ldots,k$, $k \leqslant n$, where $A[i] = B[i] = a_i$, $i = 1,\ldots,k$, and $A[k+1] \neq B[k+1]$ or $k = n$.

We abuse notations and sometimes refer to the *length of the LCP*, k, as the LCP. We, thus, denote the length of the LCP of A and B by $LCP(A, B)$.

Given a constant c and string $S = S[1]\cdots S[n]$ over alphabet $\Sigma = \{1,\ldots,n^c\}$, one can preprocess S in linear time in a manner that allows subsequent LCP queries in constant time. I.e., any query of the form $LCP(S_i, S_j)$, where S_i and S_j are the suffixes of S starting at locations i, j of S, respectively, $1 \leqslant i, j, n$, can be computed in constant time. For general alphabets, the preprocessing takes time $O(n\log n)$.

This can be done either via suffix tree construction [19,16,17,6] and Lowest Common Ancestor (LCA) queries [7,3], or suffix array construction [10] and LCP queries [11].

Our problem is formally defined as follows.

Definition 2. The *Range LCP* problem is the following:

INPUT: String $S = S[1]\cdots S[n]$ over alphabet Σ.
Preprocess S in a manner allowing efficient solutions to queries of the form:
QUERY: i, j, $1 \leqslant i \leqslant j \leqslant n$.
Compute $RangeLCP(i, j) = \max_{\ell,k\in[i..j]} LCP(S_\ell, S_k)$, where S_ℓ (resp. S_k) is the suffix of S starting at location ℓ (resp. k), i.e. $S_\ell = S[\ell]S[\ell+1]\cdots S[n]$.

Simple baseline solutions. For the sake of completeness, we describe two straight-forward algorithms, which we henceforth call Algorithm Base1 and Algorithm Base2, to solve the Range LCP problem. They are the baseline to improve. Algorithm Base1 has a linear-time preprocessing but a quadratic Range LCP query time. The algorithm preprocesses the input string S for LCP queries. Then, given an interval $[i, j]$, it computes $LCP(k, \ell)$ for every pair k, ℓ, $i \leqslant k \leqslant \ell \leqslant j$, and chooses the pair with the maximum LCP. As seen in Section 1.1 below, the preprocessing can be accomplished in linear time for alphabets that are fixed polynomials of n and in time $O(n\log n)$ for general alphabets. The query processing has $|j-i|^2$ LCP calls, each one taking a constant time for a total time $O(|j-i|^2)$. Algorithm Base2 uses two-dimensional range-maximum queries, defined as follows.

Definition 3. Let M be an $n \times n$ matrix of natural numbers. The *Two-dimensional Range-Maximum Query* (2*dRMQ*) problem is the problem of preprocessing the matrix M so that subsequent 2*dRM* queries can be answered efficiently.[6] A 2*dRM* query is of the form:

Query: $2dRM([i_1, j_1], [i_2, j_2]) = \max\{M[k, \ell] \mid i_1 \leqslant k \leqslant j_1;\ \ i_2 \leqslant \ell \leqslant j_2\}$, where $1 \leqslant i_1 \leqslant j_1 \leqslant n$ and $1 \leqslant i_2 \leqslant j_2 \leqslant n$.

Yuan and Atallah [20] showed that the 2dRMQ problem can be solved with preprocessing time linear in the matrix size and subsequent constant-time queries. As seen in Section 1.1, the LCP preprocessing can be accomplished in linear time for alphabets that are fixed polynomials of n and in time $O(n \log n)$ for general alphabets. Constructing matrix M requires n^2 LCP calls, each one taking a constant time for a total time $O(n^2)$. This is also the time required by the Yuan–Atallah [20] preprocessing algorithm as we have $O(n^2)$-size matrix. The subsequent 2dRM queries are done in constant time and provide the requested Range LCP query results, since it is precisely the maximum of all LCPs in the interval $[i, j]$. Therefore, Algorithm Base2 has a quadratic preprocessing time that enables subsequent constant-time Range LCS queries.

Paper contributions. The contributions of the paper are three-fold:

- A formalization of a natural generalization of the classical LCP problem.
- An efficient algorithm for the generalized Range LCP version.
- Introducing the notion of *bridges* and *optimal bridges* to the computation of LCP which enabled our efficient Range LCP algorithm.

The rest of the paper is organized as follows. In Section 2, we present a simple algorithm that solves the Range LCP problem with linear preprocessing and query-time linear in the range size. In Section 3 we show that, if we relax the query to only ask for a *pair* of indices in the range that provide the LCP (and not require the length of the LCP), then there is a solution whose time is linear in the range size with *no preprocessing at all*. Finally, in Section 4 we provide our main result, which is an algorithm whose preprocessing time is $O(n \log^2 n)$ that can answer Range LCP queries in time $O(\log \log n)$. Note that our preprocessing algorithm has an additional factor of $\log n$ compared to the algorithm for the *limited* version of [12] but our query time is smaller.

The results are summarized in Theorem 1.

Theorem 1. *Given a string S, the range LCP problem can be solved in*:

1. Preprocessing Time: $O(|S|)$, Space: $O(|S|)$, Query Time: $O(|j - i| \log \log n)$.
2. Preprocessing Time: *none*, Space: $O(|j - i| \log |j - i|)$, Query Time: $O(|j - i| \log |j - i|)$. *However, the query just gives the pairs with the longest LCP,* not *the LCP itself.*
3. Preprocessing Time: $O(|S| \log^2 |S|)$, Space: $O(|S| \log^{1+\varepsilon} |S|)$ *for arbitrary small constant* ε, Query Time: $O(\log \log |S|)$.

1.1. Preliminaries: suffix arrays, trees, LCA, and LCP

In this subsection we present the background needed to understand the solutions presented in this paper. The reader familiar with them can skip this subsection.

Definition 4. Let $S = S[1] \cdots S[n]$ be a string over alphabet Σ. Let $\{S_1, \ldots, S_n\}$ be the set of suffixes of S, where $S_i = S[i]S[i+1] \cdots S[n]$, $i = 1, \ldots, n$. Let $S_{i_1}, \ldots, S_{i_n}$ be the suffixes S_i, $i = 1, \ldots, n$, ordered lexicographically in increasing order. The *suffix array* of S is array A of length n where $A[1] = i_1, A[2] = i_2, \ldots, A[\ell] = i_\ell, \ldots, A[n] = i_n$.

Theorem 2. *(See Kärkkäinen and Sanders [10].) For alphabet $\Sigma = \{1, \ldots, n^c\}$, where c is a fixed constant, the suffix array can be constructed in time $O(n)$. For general alphabets it can be constructed in time $O(n \log \sigma)$, where $\sigma = \min(|\Sigma|, n)$.*

Theorem 3. *(See Bender and Farach-Colton [2].) The suffix array can be preprocessed in time $O(n)$ to enable constant time LCP query computations on the suffixes.*

Definition 5. Let $S_1, \ldots, S_k$ be strings over alphabet Σ and let $\$ \notin \Sigma$. We assume that every string S_i, $i = 1, \ldots, k$, ends with a \$ symbol.

An *uncompacted trie of strings* $S_1, \ldots, S_k$ is an edge-labeled tree with n leaves. Every path from the root to a leaf corresponds to a string S_i, with the edges labeled by the symbols of S_i. Strings with a common prefix start at the root and follow the same path of the prefix, and the paths split where the strings differ.

[6] Note that in this definition the matrix is not sparse. In Section 4, we need a different version of the problem where the matrix is sparse and therefore use other tools.

A *compacted trie* is the homeomorphic image of the uncompacted trie, i.e., every chain of edges connected by degree-2 nodes is contracted to a single edge whose label is the concatenation of the symbols on the edges of the chain.

Let $S = S[1] \cdots S[n]$ be a string over alphabet Σ. Let $\{S_1, \ldots, S_n\}$ be the set of suffixes of S, where $S_i = S[i]S[i+1] \cdots S[n]$, $i = 1, \ldots, n$. The *suffix tree* of S is the compacted trie of the suffixes $S_1, \ldots, S_n$.

Theorem 4. *(See Weiner [19].) For finite alphabet Σ, the suffix tree of a length-n string can be constructed in time $O(n)$. For general alphabets it can be constructed in time $O(n \log \sigma)$, where $\sigma = \min(|\Sigma|, n)$.*

Sketch of the construction. Weiner's construction is on-line. It begins with the last suffix S_n and constructs the tree for it (trivial). Subsequently, at each stage one more symbol is added, i.e. S_{n-1} is added to the tree, then S_{n-2}, down to S_1. The method of inserting suffix S_i is by starting at the leaf of suffix S_{i+1}, climbing up the tree until encountering a suitable crossover to the node where the leaf of S_i should be inserted. The entire process including maintenance is done in amortized linear time. For our purposes, it is important to note that maintaining, for every node, the length of the substring on the path to that node, can be easily added to the Weiner algorithm without affecting the algorithm's asymptotic complexity.

Since we now have our string in a tree data structure, one can exploit tree properties as they translate to strings. A particularly important such property is the *Lowest Common Ancestor* (*LCA*).

Definition 6. Let T be a tree, s, t nodes in T. Node r is the *Lowest Common Ancestor* (*LCA*) of s and t, if r is an ancestor of both s and t, and every other common ancestor of s and t is also an ancestor of r.

Landau and Vishkin [13] made the crucial observation that the substring from the root of a suffix tree to the LCA of nodes s and t is the LCP of the substrings on the paths from the root to nodes s and t. The following theorem allows us to compute the LCP efficiently using suffix trees.

Theorem 5. *(See Harel and Tarjan [7].) Given an n node tree, it can be preprocessed in time $O(n)$ allowing subsequent LCA queries in constant time.*

2. An algorithm linear in the range length

The quadratic time of Algorithm Base1 comes from choosing the maximum LCP of all pairs. We will show that it is sufficient to choose the maximum LCP of a judicially chosen linear number of pairs that will guarantee that the maximum LCP of *all* pairs is indeed chosen. The following lemma provides the necessary key property required for the choice of pairs.

Lemma 1. *(See Kasai et al. [11].) Let A be the suffix array of S. For any i, k, j such that $1 \leqslant i \leqslant k \leqslant j \leqslant n$ we have that $LCP(S_{A[i]}, S_{A[j]}) \leqslant LCP(S_{A[i]}, S_{A[k]})$ and $LCP(S_{A[i]}, S_{A[j]}) \leqslant LCP(S_{A[k]}, S_{A[j]})$.*

Example. Consider for example the string $S = mississippi$. The suffix array S_A of S is:

i	1	2	3	4	5	6	7	8	9	10	11
$S_{A[i]}$	11	8	5	2	1	10	9	7	4	6	3

Now, for $i = 3$, $j = 7$ and $k = 5$, we get: $S_{A[3]} = S_5$, $S_{A[7]} = S_9$ and $S_{A[5]} = S_1$. Thus, $LCP(S_{A[i]}, S_{A[j]}) = LCP(S_5, S_9) = 0$, $LCP(S_{A[i]}, S_{A[k]}) = LCP(S_5, S_1) = 0$ and $LCP(S_{A[k]}, S_{A[j]}) = LCP(S_1, S_9) = 0$.

Lemma 1 means that for any set X of suffixes, the largest LCP must be between a pair of suffixes S' and S'' in X such that there is no suffix $S''' \in X$ that is lexicographically between S' and S''. Formally,

Corollary 1. *Let $X = \{S_{A[i_1]}, \ldots, S_{A[i_k]}\}$ be a set of suffixes where $i_1 \leqslant i_2 \leqslant \cdots \leqslant i_k$, in other words, the suffixes are sorted lexicographically. Let $mlcp = \max\{LCP(S', S'') \mid S', S'' \in X\}$. Then,*

$$mlcp \in \left\{LCP(S_{A[i_1]}, S_{A[i_2]}), LCP(S_{A[i_2]}, S_{A[i_3]}), \ldots, LCP(S_{A[i_{k-1}]}, S_{A[i_k]})\right\}.$$

Proof. Assume to the contrary that $mlcp = LCP(S_{A[i_j]}, S_{A[i_{j'}]})$ such that $i_j < i_{j'} - 1$, $i_{j'} \leqslant k$, $i_j \geqslant 1$. Then, there exists $i_{j''}$, $i_j < i_{j''} < i_{j'}$. By Lemma 1, it holds that

$$mlcp \leqslant LCP(S_{A[i_j]}, S_{A[i_{j''}]})$$

and

$$mlcp \leqslant LCP(S_{A[i_{j''}]}, S_{A[i_{j'}]}).$$

Algorithm LinearRange

Preprocessing:
Preprocess string S for LCP queries.
Construct suffix array A of S.

Query: Given interval $[i, j]$.
Construct set $X = \{\ell_0, \ldots, \ell_{|j-i|}\}$ where $i + h = A[\ell_h],\ h = 0, \ldots, |j-i|$.
Let $Y[0], \ldots, Y[|j-i|-1]$ be an array of the elements of X sorted in ascending order.
For $h = 0, \ldots, |j-i|-1$, compute $LCP(S_{A[Y[h]]}, S_{A[Y[h+1]]})$.
Choose the pair with the maximum LCP.

Fig. 1. The linear-time query algorithm.

If one of these two inequalities is a strict inequality then this is in contradiction to the maximality of the *mlcp*, therefore, both inequalities are actually equalities. Since the above claim holds for every $i_{j''}$, $i_j < i_{j''} < i_{j'}$, in particular it holds for $i_{j''} = i_{j+1}$, which contradicts the assumption. The corollary follows. □

The corollary means that if the suffixes are lexicographically sorted, then only the LCP of pairs of adjacent suffixes needs to be tested, and there is only a linear number of such pairs, not a quadratic number. Algorithm LinearRange (see Fig. 1) is an adaptation of Algorithm Base1 that exploits Corollary 1 to achieve time linear in the range size.

Algorithm LinearRange's time. As seen in the analysis of the preprocessing time of Algorithm Base1, the preprocessing can be accomplished in linear time for alphabets that are fixed polynomials of n and in time $O(n \log n)$ for general alphabets. The query processing has $|j-i|$ LCP calls, each one taking a constant time for a total time $O(|j-i|)$. Sorting X can be done in time $O(|j-i| \log\log n)$ by noting that the indices to be sorted are from the range $\{1, \ldots, n\}$ and thus a data structure such as [18] can be used.

We have, therefore, proven the first part of Theorem 1.

3. Finding the pair with the longest common prefix

The linear-time preprocessing algorithm of Section 2 is, nevertheless, quite heavy in that it requires constructing a suffix array of the text. If we relax the query to require only a pair of indices whose LCP is the longest in the range, it is possible to eliminate the need for the preprocessing and solve the query in time and space proportional to the range length. The idea of this algorithm leads to the efficient algorithm of Section 4.

Algorithm *Pair*'s idea. Until now, we have considered the LCP of all pairs in the range and chose the largest. We reverse our outlook. We now consider the largest common prefix in the range, and then choose a pair of indices that have that LCP. As mentioned in Section 1.1, the suffix tree provides all common prefixes, and can be constructed in linear time. The problem is that we do not want to construct the suffix tree from the range till the end, since that construction time will not be linear in the range size.

Let the query be *RangeLCP*(i, j), and let $d = |j-i|+1$. Construct the suffix tree of $S' = S[i]S[i+1]\cdots S[i+3d]$ (a \$ is added at the end of S'), and mark on that tree the leaves representing suffixes that start in the range $[i, j]$. Consider the lowest suffix tree node that has at least two marked descendants. Call that node x. There are two cases:

Case a: If x has no marked child whose edge is the special *end-of-text* symbol \$, then the substring ending at node x is the *RangeLCP*(i, j), and any pair of marked leaves in the subtree rooted at x is a pair whose LCP is longest in the range (see Fig. 2(a)).

Case b: If x has a marked child whose edge label is the symbol \$, then it could be that the LCP extends beyond the end of the substring of length $3d$ for which we have constructed the suffix tree (see Fig. 2(b)). In this case, we do not know how far the LCP extends. However, we choose the pair of marked leaves of x to be the marked child whose edge is labeled \$ (assume that it represents the suffix starting at index ℓ), and the marked leaf representing the suffix that is closest to ℓ.

Lemma 3 guarantees that the pair of indices we choose in the second case has the largest common prefix in the range. The well-known periodicity lemma is needed in order to prove Lemma 3.

Lemma 2 *(Periodicity lemma). If a string S of length n has periods of sizes p and q such that $p + q \leqslant n$ then S has also a period of size $gcd(p, q)$, where $gcd(\,,\,)$ stands for the greatest common divisor of two integers.*

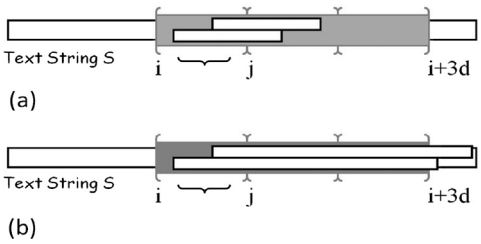

Fig. 2. (a) The LCP of all pairs in interval $[i, j]$ is within the length $3d$ substring S'. (b) The LCP of all pairs in interval $[i, j]$ is longer than the length $3d$ substring S'.

Fig. 3. The periodicity of $S[i+d]\cdots S[i+3d]$.

Algorithm Pair

Query: Given interval $[i, j]$.
Construct suffix tree for $S' = S[i]S[i+1]\cdots S[i+3|j-i|]$.
Mark all leaves of the suffix tree of S' that are the suffixes $S'_i, \ldots, S'_j$.
Let $x_1, \ldots, x_k$ be the nodes with the longest path from the root,
which have at least two marked leaves in their subtrees.
If there is no $x \in \{x_1, \ldots, x_k\}$ that has a marked child where the label on the edge to that child is \$:
Then any pair of leaves in any of the $x_1, \ldots, x_k$ subtrees has the LCP.
Otherwise, let $x \in \{x_1, \ldots, x_k\}$ be such that the substring on the path from the root to x
is the suffix $S[\ell]S[\ell+1]\cdots S[i+3|j-i|]$, and that one of the children of x has an edge labeled \$.
Let S_t be the marked suffix of S' that is in x's subtree and that is closest to ℓ,
i.e., every marked suffix S'_p in the subtree of x has $t \geqslant p$.
Choose ℓ and t as the pair with the LCP.

Fig. 4. The no-preprocessing linear-time query algorithm.

Lemma 3. *Let x be a lowest node in the suffix tree of S' that has more than one marked leaf. Assume also that the substring on the path from the root to x is the suffix $S[\ell]S[\ell+1]\cdots S[i+3d]$, and that one of the marked children of x has edge labeled \$. Let S_t be the marked suffix of S' that is in x's subtree and that is closest to ℓ, i.e., every marked suffix S'_p in the subtree of x has $t \geqslant p$. Then ℓ and t are the indices between i and j whose common prefix is largest.*

Proof. Note that $|\ell - t| < d$ yet the length of the common prefix of the two suffixes S_ℓ and S_t is at least $2d$. The situation is, therefore, as depicted in Fig. 3, which means that the suffix S_ℓ is periodic, and its period's length is $|\ell - t|$. Clearly, the LCP will extend as long as the period continues in S. Also, the substring $S[i+d]\cdots S[i+3d]$ is periodic with the same period as S_ℓ and S_t. Therefore, it is impossible that within the range $[i, j]$ there is a pair with a longer common prefix, because such a pair either implies the existence of another different period and contradicts the periodicity lemma or contradicts the choice of x as the lowest node having at least two marked nodes. □

The pseudo-code for Algorithm Pair is presented in Fig. 4.

Algorithm *Pair*'s time. Algorithm Pair constructs the suffix tree of $S[i]\cdots S[i+3d]$, thus its time is $O(|j-i|)$ for fixed finite alphabets and $O(|j-i|\log|j-i|)$ for general alphabets. All other manipulations of the suffix tree are linear in the size of the tree, which is $O(|j-i|)$.

We have, therefore, proven the second part of Theorem 1.

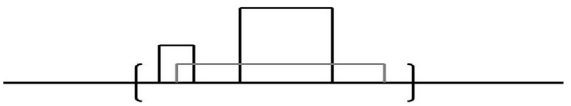

Fig. 5. Bridges and optimal bridges in an interval. The x-axis shows the start–end indices of the interval, i.e., the length and position of the bridge, and the y-axis shows the LCP of this pair of indices, i.e. the height of the bridge. The black intervals are optimal bridges. The grey interval is not optimal since it contains a nested optimal bridge.

4. An efficient range LCP algorithm

The algorithm presented in this section is based on efficiently calculating and using *optimal bridges*. We define this concept below.[7]

Definition 7. A *bridge of height* ℓ *over a string* S of length n is a pair of indices $\langle i, j\rangle$, $1 \leqslant i < j \leqslant n$, where ℓ is the length of $LCP(S_i, S_j)$. We say that $|j - i|$ is the *length* of the bridge.

Bridge $\langle i', j'\rangle$ is said to be *nested* in bridge $\langle i, j\rangle$, if $i' \geqslant i$ and $j' \leqslant j$.

Bridge $\langle i', j'\rangle$ is said to be *interleaved* with bridge $\langle i, j\rangle$, if one of the following two conditions holds: (1) $i' > i$ and $j' > j$ and $j > i'$, or (2) $i' < i$ and $j' < j$ and $i < j'$.

An *optimal bridge* is a bridge $\langle i, j\rangle$ of height ℓ such that there is no bridge $\langle i', j'\rangle$ of height ℓ', $\ell' \geqslant \ell$, where $\langle i', j'\rangle$ is nested in $\langle i, j\rangle$. (See Fig. 5.)

Algorithm *Bridges*' idea. The algorithm exploits the following property:

Observation 1 *(The Bridge-LCP property).* $RangeLCP(a, b)$ is the maximum height of all the bridges $\langle i, j\rangle$ contained in interval $[a, b]$, i.e. where $a \leqslant i < j \leqslant b$.

The problem is that there is a quadratic number of bridges. The idea of the efficient *RangeLCP* algorithm is to use optimal bridges, rather than bridges. It is easy to verify that the Bridge-LCP property holds when using only optimal bridges, rather than all bridges, however, it is also clear that there are fewer optimal bridges than bridges. In the sequel, we will prove that there are $O(n \log n)$ optimal bridges, and show how to construct them in time $O(n \log n)$. In addition, we will show a suitable reduction to two dimensional points where an orthogonal range query provides the maximum length bridge within a given interval.

To prove the bound on the number of optimal bridges, we need the following concept.

Definition 8. A substring S' of string S is a *maximal LCP*, if there exist indices i, j such that $S' = LCP(S_i, S_j)$ and there are no indices k, ℓ where S' is a proper prefix of $LCP(S_k, S_\ell)$.

Example. In the string $S = ABABCDECDE$, the maximal LCPs are: B, E, AB, DE and CDE. C and CD are not maximal, since they are prefixes of CDE which is $LCP(S_5, S_8)$, and A is not maximal since it is a prefix of AB.

Lemma 4. *There are* $O(n \log n)$ *optimal bridges in a string of length* n.

Proof. We consider a recursive counting of the optimal bridges.

Base case: Let $\ell_{\max}$ be the length of a maximal LCP P in S. Clearly, there are a number of bridges for that P, all their heights are $\ell_{\max}$. A succinct way of representing all the optimal bridges that have LCP P is the following. List, in ascending order, all indices $i_1, \ldots, i_m$ such that $LCP(S_i, S_j) = P$, $i, j \in \{i_1, \ldots, i_m\}$, $i \neq j$. The optimal bridges are $\langle i_1, i_2\rangle, \langle i_2, i_3\rangle, \ldots, \langle i_{m-1}, i_m\rangle$.

The lengths of all succinct representations of all the maximal LCPs do not exceed n since they represent bridges starting at different indices.

Merging case: Let $L = \{L_1, \ldots, L_k\}$ be the lists of succinct representations defined thus far. For any pair $L_i, L_j \in L$, where L_i is the succinct representation list of P_i and L_j is the succinct representation list of P_j, consider $P = LCP(P_i, P_j)$. Let $L_{i_1}, L_{i_2} \in L$ be a pair such that $P' = LCP(P_{i_1}, P_{i_2})$ is the longest. If P' is the null string we are done. Otherwise, proceed as follows. Any bridge for P' that is nested in a bridge of P_{i_1} or P_{i_2} is not optimal, and thus irrelevant to our count. However, whenever a bridge $\langle x_1, y_1\rangle$ of height equal to the length of P_{i_1} interleaves with a bridge $\langle x_2, y_2\rangle$ of height equal to the length of P_{i_2}, it creates three new bridges of height equal to the length of P': $\langle x_1, x_2\rangle$, $\langle x_2, y_1\rangle$, and $\langle y_1, y_2\rangle$. In general, if we merge the succinct representation lists of P_{i_1} and P_{i_2}, we get a

[7] The methods used in this section do not use assumptions on the alphabet and, therefore, apply to general alphabets.

new list that is the succinct representation of P'. Both the lists of P_{i_1} and P_{i_2} provide bridges that the Range LCP algorithm needs, but as far as the recursive counting, their lists are taken out of L and the merged list is inserted in L.

All lists that were created during this recursive process provide bridges. At this point, all non-optimal bridges are taken out. We will count the number of bridges this process creates (including the non-optimal ones) and show that it is bounded by $O(n \log n)$.

As mentioned previously, the number of maximal bridges cannot be more than n. Whenever two lists are merged, we keep the bridges of the lists, as well as those of the merged list. Note that the number of bridges added by a merge is the number of interleaves, which is bounded above by the size of the smaller of the two lists. Because no element can be merged more than $\log n$ times (the smaller-half feature), the total number of bridges is $O(n \log n)$. □

A lower bound on the number of optimal bridges. As noted, our analysis gives an upper bound on the bridges, but cleaning out the non-optimal bridges may reduce that number. Unfortunately, the Fibonacci word provides a lower bound of $\Omega(n \log n)$ on the number of optimal brides. The Fibonacci words are defined as follows.

$S_0 = A$, $S_1 = AB$, …, $S_n = S_{n-2}S_{n-1}$ (the concatenation of the previous two Fibonacci words).

A *repetition* is a string of the form $U^k U'$ where $k \geqslant 2$ and U' is a prefix of U. Clearly, every such U is an optimal bridge. A Fibonacci word of length n has $O(n \log n)$ repetitions, providing a lower bound of $\Omega(n \log n)$ optimal bridges.

4.1. Constructing the optimal bridges

We present an algorithm to construct the succinct representation of the optimal bridges. Our algorithm constructs the lists from the highest bridge down in time linear in their number. Lemma 4 assures that the number of bridges is $O(n \log n)$.

Algorithm *Construct-Optimal-Bridges* idea. Let T_S be a suffix tree of string S. Consider a node x in T_S such that the length of the substring from the root to x is maximal. Let the substring from the root to x be P. All the children of x are leaves (suffixes) and thus all pairs of children of x define bridges. However, if $x_1, \ldots, x_k$ are the children of x (leaves in x's subtree, i.e., suffixes of S) sorted by their index, then they provide us with the succinct representation of the bridges of LCP P.

In the initial stage of the algorithm all the maximal LCPs have their succinct bridge representations. We inductively climb up T_S, merging the succinct lists of the lower nodes. The merge is done in a fashion whereby the smaller list is merged into the larger one, to create the succinct representation of bridges of a smaller size. When the root is reached we have the succinct representation of all bridges.

Algorithm *Construct-Optimal-Bridges* time. Because of the fact that the smaller lists are merged into the larger ones, this algorithm can be trivially implemented in time $O(n \log^2 n)$. However, Apostolico and Preparata [1] show how this process can be done in time $O(n \log n)$. Alternatively, it can be done using the mergeable dictionaries of [8] with the same complexity bounds.

4.2. Finding the smallest bridge in an interval

Assume that we have a list of the optimal bridges. We consider every bridge $\langle x, y \rangle$ as a point on the plane with coordinates (x, y). Every such point has weight $\ell = LCP(S_x, S_y)$. For any interval $[i, j]$, $RangeLCP(i, j)$ is the point with the maximum value in the orthogonal range $[i, j]$, $[i, j]$.

Orthogonal range queries on the plane, where k points are assigned weights can be answered using algorithms whose preprocessing time is $O(k \log k)$ giving a data structure of $O(k \log^{\varepsilon} k)$-space, for arbitrary small ε, and whose query time is $O(\log \log k)$ [4].

Algorithm's complexity. Our preprocessing algorithm's time is $O(n \log n)$ to produce the optimal bridges, and $O(k \log k)$ for the orthogonal range query preprocessing, where k is the number of optimal bridges. This number may be as large as $O(n \log n)$. Thus the total preprocessing algorithm time is $O(n \log^2 n)$ in the worst case, with $O(\log \log n)$ query time. The space is the dominated by the space needed for the orthogonal range query data structure of [4] for the $O(n \log n)$ points, which is $O(n \log^{1+\varepsilon} n)$.

We have, therefore, proven the third part of Theorem 1.

5. Conclusion and open problems

The LCP is a powerful tool that can be efficiently computed. The RangeLCP problem defined here is a natural extension. Surprisingly, it seems that with an almost linear preprocessing time, one can compute the RangeLCP in sub-logarithmic time. It would be interesting to know if the Range LCP problem can be solved with a faster query time, preferably constant time, and faster preprocessing time, preferably linear.

References

[1] A. Apostolico, F.P. Preparata, Optimal off-line detection of repetitions in a string, Theor. Comput. Sci. 22 (1983) 297–315.
[2] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: LATIN, 2000, pp. 88–94.
[3] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, U. Vishkin, Highly parallelizable problems, in: Proc. 21st ACM Symposium on Theory of Computation, 1989, pp. 309–319.
[4] T.M. Chan, K.G. Larsen, M. Pătraşcu, Orthogonal range searching on the ram, revisited, in: Proc. 27th ACM Symposium on Computational Geometry (SoCG), 2011, pp. 1–10.
[5] G. Cormode, S. Muthukrishnan, Substring compression problems, in: Proc. 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), 2005, pp. 321–330.
[6] M. Farach, Optimal suffix tree construction with large alphabets, in: Proc. 38th IEEE Symposium on Foundations of Computer Science, 1997, pp. 137–143.
[7] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestor, J. Comput. Syst. Sci. 13 (1984) 338–355.
[8] J. Iacono, O. Özkan, Mergeable dictionaries, in: Proc. 37th International Colloquium on Automata, Languages and Programming (ICALP), vol. 1, in: Lect. Notes Comput. Sci., vol. 6198, Springer, 2010, pp. 164–175.
[9] L. Ilie, L. Tinta, Practical algorithms for the longest common extension problem, in: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE), in: Lect. Notes Comput. Sci., vol. 5721, Springer, 2009, pp. 302–309.
[10] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP), in: Lect. Notes Comput. Sci., vol. 2719, 2003, pp. 943–955.
[11] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Proc. 12th Symposium on Combinatorial Pattern Matching (CPM), 2001, pp. 181–192.
[12] O. Keller, T. Kopelowitz, S. Landau, M. Lewenstein, Generalized substring compression, in: Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM), Springer, 2009, pp. 26–38.
[13] G.M. Landau, U. Vishkin, Efficient string matching in the presence of errors, in: Proc. 26th IEEE FOCS, 1985, pp. 126–136.
[14] G.M. Landau, U. Vishkin, Fast parallel and serial approximate string matching, J. Algorithms 10 (2) (1989) 157–169.
[15] A. Lempel, J. Ziv, On the complexity of finite sequences, IEEE Trans. Inf. Theory 22 (1976) 75–81.
[16] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (1976) 262–272.
[17] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (1995) 249–260.
[18] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10 (1977) 99–127.
[19] P. Weiner, Linear pattern matching algorithm, in: Proc. 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[20] H. Yuan, M.J. Atallah, Data structures for range minimum queries in multidimensional arrays, in: Proc. 21st ACM–SIAM Symposium on Discrete Algorithms (SODA), 2010, pp. 150–160.