Time-Space Trade-Offs for Longest Common Extensions[⋆]

Philip Bille[1], Inge Li Gørtz[1], Benjamin Sach[2], and Hjalte Wedel Vildhøj[1]

[1] Technical University of Denmark, DTU Informatics, {phbi,ilg,hwvi}@imm.dtu.dk
[2] University of Warwick, Department of Computer Science, sach@dcs.warwick.ac.uk

Abstract. We revisit the longest common extension (LCE) problem, that is, preprocess a string T into a compact data structure that supports fast LCE queries. An LCE query takes a pair (i, j) of indices in T and returns the length of the longest common prefix of the suffixes of T starting at positions i and j. We study the time-space trade-offs for the problem, that is, the space used for the data structure vs. the worst-case time for answering an LCE query. Let n be the length of T. Given a parameter τ, $1 \leq \tau \leq n$, we show how to achieve either $O(n/\sqrt{\tau})$ space and $O(\tau)$ query time, or $O(n/\tau)$ space and $O(\tau \log(|\mathrm{LCE}(i,j)|/\tau))$ query time, where $|\mathrm{LCE}(i,j)|$ denotes the length of the LCE returned by the query. These bounds provide the first smooth trade-offs for the LCE problem and almost match the previously known bounds at the extremes when $\tau = 1$ or $\tau = n$. We apply the result to obtain improved bounds for several applications where the LCE problem is the computational bottleneck, including approximate string matching and computing palindromes. Finally, we also present an efficient technique to reduce LCE queries on two strings to one string.

1 Introduction

Given a string T, the *longest common extension* of suffix i and j, denoted $\mathrm{LCE}(i, j)$, is the length of the longest common prefix of the suffixes of T starting at position i and j. The *longest common extension problem* (LCE problem) is to preprocess T into a compact data structure supporting fast longest common extension queries.

The LCE problem is a basic primitive that appears as a subproblem in a wide range of string matching problems such as approximate string matching and its variations [3, 6, 18, 20, 25], computing exact or approximate tandem repeats [10, 19, 22], and computing palindromes. In many of the applications, the LCE problem is the computational bottleneck.

In this paper we study the time-space trade-offs for the LCE problem, that is, the space used by the preprocessed data structure vs. the worst-case time used by LCE queries. We assume that the input string is given in read-only memory and is not counted in the space complexity. There are essentially only two time-space

[⋆] This work was partly supported by EPSRC.

trade-offs known: At one extreme we can store a suffix tree combined with an efficient nearest common ancestor (NCA) data structure [12] (other combinations of $O(n)$ space data structures for the string can also be used to achieve this bound). This solution uses $O(n)$ space and supports LCE queries in $O(1)$ time. At the other extreme we do not store any data structure and instead answer queries simply by comparing characters from left-to-right in T. This solution uses $O(1)$ space and answers an $\mathrm{LCE}(i,j)$ query in $O(|\mathrm{LCE}(i,j)|) = O(n)$ time. Using succinct data structures the space for the first solution can be slightly improved to $O(n)$ *bits* (see e.g. [8]). The second solution was recently shown to be very practical [13].

1.1 Our Results

We show the following main result.

Theorem 1. *For a string T of length n, and any parameter τ, $1 \leq \tau \leq n$, we can solve the longest common extension problem*

(i) in $O(\frac{n}{\sqrt{\tau}})$ space, $O(\tau)$ query time, and $O(\frac{n^2}{\sqrt{\tau}})$ preprocessing time, or in

(ii) in $O(\frac{n}{\tau})$ space, $O\left(\tau \log\left(\frac{|\mathrm{LCE}(i,j)|}{\tau}\right)\right)$ query time, and $O(n \log n)$ preprocessing time. The preprocessing time bound is achieved with high probability, that is, with probability at least $1 - 1/n^c$ for any constant c. All other bounds are worst case.

Hence, we provide a smooth time-space trade-off that allows several new and non-trivial bounds. For instance, with $\tau = \sqrt{n}$ Theorem 1(i), gives a solution using $O(n^{3/4})$ space and $O(\sqrt{n})$ time. If we allow randomisation, we can use Theorem 1(ii) to further reduce the space to $O(\sqrt{n})$ while using query time $O(\sqrt{n}\log(|\mathrm{LCE}(i,j)|/\sqrt{n})) = O(\sqrt{n}\log n)$. Note that at both extremes of the trade-off ($\tau = 1$ or $\tau = n$) we almost match the previously known bounds.

Furthermore, we also consider LCE queries between two strings, i.e. the pair of indices to an LCE query is from different strings. We present a general result that reduces the query on two strings to a single one of them. When one of the strings is significantly smaller than the other, we can combine this reduction with Theorem 1 to obtain even better time-space trade-offs.

1.2 Techniques

The high-level idea in Theorem 1 is to combine and balance out the two extreme solutions for the LCE problem. For Theorem 1(i) we use *difference covers* to sample a set of suffixes of T of size $O(n/\sqrt{\tau})$. We store a compact trie combined with an NCA data structure for this sample using $O(n/\sqrt{\tau})$ space. To answer an LCE query we compare characters from T until we get a mismatch or reach a pair of sampled suffixes, which we then immediately compute the answer for. By the properties of difference covers we compare at most $O(\tau)$ characters before reaching a pair of sampled suffixes. Similar ideas have previously been used to achieve trade-offs for suffix array and LCP array construction [15, 26].

For Theorem 1(ii) we show how to use Rabin-Karp fingerprinting [16] instead of difference covers to reduce the space further. We show how to store a sample of $O(n/\tau)$ fingerprints, and how to use it to answer LCE queries using doubling search combined with directly comparing characters. This leads to the output-sensitive $O(\tau \log(|\mathrm{LCE}(i,j)|/\tau))$ query time. We reduce space compared to Theorem 1 by computing fingerprints on-the-fly as we need them. Initially, we give a Monte-Carlo style randomised data structure that may answer queries incorrectly (see Theorem 5). However, this solution uses only $O(n)$ preprocessing time and is therefore of independent interest in applications that can tolerate errors. To get the error-free Las-Vegas style bound of Theorem 1(ii) we need to verify the fingerprints we compute are collision free; i.e. two fingerprints are equal iff the corresponding substrings of T are equal. The main challenge is to do this in only $O(n \log n)$ time. We achieve this by showing how to efficiently verify fingerprints of composed samples which we have already verified, and by developing a search strategy that reduces the fingerprints we need to consider.

Finally, the reduction for LCE on two strings to a single string is based on a simple and compact encoding of the larger string using the smaller string. The encoding could be of independent interest in related problems, where we want to take advantage of different length input strings.

1.3 Applications

With Theorem 1 we immediately obtain new results for problems based on LCE queries. We review some the most important below.

Approximate String Matching Given strings P and T and an error threshold k, the *approximate string matching problem* is to report all ending positions of substrings of T whose *edit distance* to P is at most k. The edit distance between two strings is the minimum number of insertions, deletions, and substitutions needed to convert one string to the other. Let m and n be the lengths of P and T. The Landau-Vishkin algorithm [20] solves approximate string matching using $O(nk)$ LCE queries on P and substrings of T of length $O(m)$. Using the standard linear space and constant time LCE data structure, this leads to a solution using $O(nk)$ time and $O(m)$ space (the $O(m)$ space bound follows by the standard trick of splitting T into overlapping pieces of size $O(m)$). If we plug in the results from Theorem 1 we immediately obtain the following result.

Theorem 2. *Given strings P and T of lengths m and n, respectively, and a parameter τ, $1 \leq \tau \leq m$, we can solve approximate string matching*

(i) in $O\big(\frac{m}{\sqrt{\tau}}\big)$ space and $O(nk \cdot \tau + \frac{nm}{\sqrt{\tau}})$ time, or
(ii) in $O\big(\frac{m}{\tau}\big)$ space and $O(nk \cdot \tau \log m)$ time with high probability.

To the best of our knowledge these are the first non-trivial bounds for approximate string matching using $o(m)$ space.

Palindromes Given a string T the *palindrome problem* is to report the set of all *maximal palindromes* in T. A substring $T[i \dots j]$ is a maximal palindrome iff $T[i \dots j] = T[i \dots j]^R$ and $T[i-1 \dots j+1] \neq T[i-1 \dots j+1]^R$. Here $T[i \dots j]^R$ denotes the reverse of $T[i \dots j]$. Any palindrome in T occurs in the middle of a maximal palindrome, and thus the set of maximal palindromes compactly represents all palindromes in T. The palindrome problem appears in a diverse range of applications, see e.g. [2, 4, 9, 14, 17, 21, 24].

We can trivially solve the problem in $O(n^2)$ time and $O(1)$ space by a linear search at each position in T to find the maximal palindrome. With LCE queries we can immediately speed up this search. Using the standard $O(n)$ space and constant time solution to the LCE problem this immediately implies an algorithm for the palindrome problem that uses $O(n)$ time and space (this bound can also be achieved without LCE queries [23]). Using Theorem 1 we immediately obtain the following result.

Theorem 3. *Given a string of length n and a parameter τ, $1 \leq \tau \leq n$, we can solve the palindrome problem*

(i) in $O\big(\frac{n}{\sqrt{\tau}}\big)$ space and $O\big(\frac{n^2}{\sqrt{\tau}} + n\tau\big)$ time.
(ii) in $O\big(\frac{n}{\tau}\big)$ space and $O(n \cdot \tau \log n)$ time with high probability.

For $\tau = \omega(1)$, these are the first sublinear space bounds using $o(n^2)$ time. Similarly, we can substitute our LCP data structures in the LCP-based variants of palindrome problems such as *complemented palindromes*, *approximate palindromes*, or *gapped palindromes*, see e.g. [17].

Tandem Repeats Given a string T, the *tandem repeats problem* is to report all squares, i.e. consecutive repeated substrings in T. Main and Lorentz [22] gave a simple solution for this problem based on LCP queries that achieves $O(n)$ space and $O(n \log n + \text{occ})$ time, where occ is the number of tandem repeats in T. Using different techniques Gąsieniecs et al. [11] gave a solution using $O(1)$ space and $O(n \log n + \text{occ})$ time. Using Theorem 1 we immediately obtain the following result.

Theorem 4. *Given a string of length n and parameter τ, $1 \leq \tau \leq n$, we can solve the tandem repeats problem*

(i) in $O\big(\frac{n}{\sqrt{\tau}}\big)$ space and $O\big(\frac{n^2}{\sqrt{\tau}} + n\tau \cdot \log n + occ\big)$ time.
(ii) in $O\big(\frac{n}{\tau}\big)$ space and $O\big(n\tau \cdot \log^2 n + \text{occ}\big)$ time with high probability.

While this does not improve the result by Gąsieniecs et al. it provides a simple LCP-based solution. Furthermore, our result generalizes to the approximate versions of the tandem repeats problem, which also have solutions based on LCP queries [19].

2 The Deterministic Data Structure

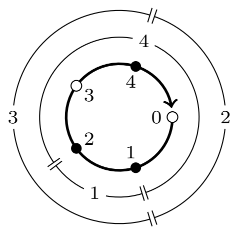
We now show Theorem 1(i). Our deterministic time-space trade-off is based on sampling suffixes using *difference covers*.

2.1 Difference Covers

A *difference cover modulo* τ is a set of integers $D \subseteq \{0, 1, \ldots, \tau - 1\}$ such that for any distance $d \in \{0, 1, \ldots, \tau - 1\}$, D contains two elements separated by distance d modulo τ (see example 1).

Example 1. The set $D = \{1, 2, 4\}$ is a difference cover modulo 5.

d	0	1	2	3	4
i, j	$1, 1$	$2, 1$	$1, 4$	$4, 1$	$1, 2$



A difference cover D can cover at most $|D|^2$ differences, and hence D must have size at least $\sqrt{\tau}$. We can also efficiently compute a difference cover within a constant factor of this bound.

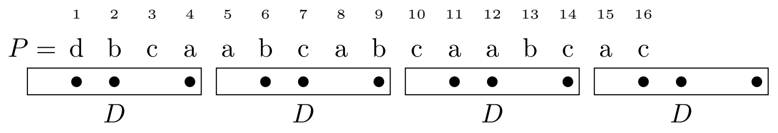
Lemma 1 (Colbourn and Ling [5]). *For any τ, a difference cover modulo τ of size at most $\sqrt{1.5\tau} + 6$ can be computed in $O(\sqrt{\tau})$ time.*

2.2 The Data Structure

Let T be a string of length n and let τ, $1 \leq \tau \leq n$, be a parameter. Our data structure consists of the compact trie of a sampled set of suffixes from T combined with a NCA data structure. The sampled set of suffixes $\mathcal{S}$ is the set of suffixes obtained by overlaying a difference cover on T with period τ, that is,

$$\mathcal{S} = \{i \mid 1 \leq i \leq n \ \wedge \ i \bmod \tau \in D\} \ .$$

Example 2. Consider the string T = dbcaabcabcaabcac. As shown below, using the difference cover from Example 1, we obtain the suffix sample $\mathcal{S} = \{1, 2, 4, 6, 7, 9, 11, 12, 14, 16\}$.



By Lemma 1 the size of $\mathcal{S}$ is $O(n/\sqrt{\tau})$. Hence the compact trie and the NCA data structures use $O(n/\sqrt{\tau})$ space. We construct the data structure in $O(n^2/\sqrt{\tau})$ time by inserting each of the $O(n/\sqrt{\tau})$ sampled suffixes in $O(n)$ time.

To answer an LCE(i, j) query we explicitly compare characters starting from i and j until we either get a mismatch or we encounter a pair of sampled suffixes. If we get a mismatch we simply report the length of the LCE. Otherwise, we do a NCA query on the sampled suffixes to compute the LCE. Since the distance to a pair of sampled suffixes is at most τ the total time to answer a query is $O(\tau)$. This concludes the proof of Theorem 1(i).

3 The Monte-Carlo Data Structure

We now show Theorem 5 below which is an intermediate step towards proving Theorem 1(ii) but is also of independent interest, providing a Monte-Carlo time-space trade-off. The technique is based on sampling suffixes using *Rabin-Karp fingerprints.* These fingerprints will be used to speed up queries with large LCE_P values while queries with small LCE_P values will be handled naively.

Theorem 5. *For a string T of length n, and any parameter τ, $1 \leq \tau \leq n$, we can solve the* LCE *problem in $O\left(\frac{n}{\tau}\right)$ space, $O\left(\tau \log\left(\frac{|\mathrm{LCE}(i,j)|}{\tau}\right)\right)$ query time, and $O(n)$ preprocessing. The solution is randomised (Monte-Carlo); with high probability, all queries are answered correctly.*

3.1 Rabin-Karp fingerprints

Rabin-Karp fingerprints are defined as follows. Let $2n^{c+4} < p \leq 4n^{c+4}$ be some prime and choose $b \in \mathbb{Z}_p$ uniformly at random. Let S be any substring of T, the fingerprint $\phi(S)$ is given by,

$$\phi(S) = \sum_{k=1}^{|S|} S[k]b^k \bmod p\,.$$

Lemma 2 gives a crucial property of these fingerprints (see e.g. [16] for a proof). That is with high probability we can determine whether any two substrings of T match in constant time by comparing their fingerprints.

Lemma 2. *Let ϕ be a fingerprinting function picked uniformly at random (as described above). With high probability,*

$$\begin{aligned} \phi(T[i\ldots i+\alpha-1]) &= \phi(T[j\ldots j+\alpha-1]) \\ \textit{iff}\ T[i\ldots i+\alpha-1] &= \quad T[j\ldots j+\alpha-1] \quad \textit{for all } i,j,\alpha. \end{aligned} \qquad (1)$$

3.2 The Data Structure

The data structure consists of the fingerprint, ϕ_k, of each suffix of the form $T[k\tau\ldots n]$ for $0 < k < n/\tau$, i.e. $\phi_k = \phi(T[k\tau\ldots n])$. Note that there are $O(n/\tau)$ such suffixes and the starting points of two consecutive suffix are τ characters apart. Since each fingerprint uses constant space the space usage of the data structure is $O(n/\tau)$. The n/τ fingerprints can be computed in left-to-right order by a single scan of T in $O(n)$ time.

Queries The key property we use to answer a query is given by Lemma 3.

Lemma 3. *The fingerprint $\phi(T[i\ldots i+\alpha-1])$ of any substring $T[i\ldots i+\alpha-1]$ can be constructed in $O(\tau)$ time. If i, α are divisible by τ, the time becomes $O(1)$.*

Proof. Let $k_1 = \lceil i/\tau \rceil$ and $k_2 = \lceil (i+\alpha)/\tau \rceil$ and observe that we have ϕ_{k_1} and ϕ_{k_2} stored. By the definition of ϕ, we can compute $\phi(T[k_1\tau \ldots k_2\tau - 1]) = \phi_{k_1} - \phi_{k_2} \cdot b^{(k_2-k_1)\tau} \bmod p$ in $O(1)$ time. If $i, \alpha = 0 \bmod \tau$ then $k_1\tau = i$ and $k_2\tau = i+\alpha$ and we are done. Otherwise, similarly we can then convert $\phi(T[k_1\tau \ldots k_2\tau - 1])$ into $\phi(T[k_1\tau - 1 \ldots k_2\tau - 2])$ in $O(1)$ time by inspecting $T[k_1\tau - 1]$ and $T[k_2\tau - 1]$. By repeating this final step we obtain $T[i \ldots i+\alpha-1]$ in $O(\tau)$ time.

We now describe how to perform a query by using fingerprints to compare substrings. We define $\phi_k^\ell = \phi(T[k\tau \ldots (k+2^\ell)\tau - 1])$ which we can compute in $O(1)$ time for any k, ℓ by Lemma 3.

First consider the problem of answering a query of the form $\text{LCE}(i\tau, j\tau)$. Find the largest ℓ such that $\phi_i^\ell = \phi_j^\ell$. When the correct ℓ is found convert the query into a new query $\text{LCE}((i+2^\ell)\tau, (j+2^\ell)\tau)$ and repeat. If no such ℓ exists, explicitly compare $T[i\tau \ldots (i+1)\tau - 1]$ and $T[j\tau \ldots (j+1)\tau - 1]$ one character at a time until a mismatch is found. Since no false negatives can occur when comparing fingerprints, such a mismatch exists. Let ℓ_0 be the value of ℓ obtained for the initial query, $\text{LCE}(i\tau, j\tau)$, and ℓ_q the value obtained during the q-th recursion. For the initial query, we search for ℓ_0 in increasing order, starting with $\ell_0 = 0$. After recursing, we search for ℓ_q in descending order, starting with ℓ_{q-1}. By the maximality of ℓ_{q-1}, we find the correct ℓ_q. Summing over all recursions we have $O(\ell_0)$ total searching time and $O(\tau)$ time scanning T. The desired query time follows from observing that by the maximality of ℓ_0, we have that $O(\tau + \ell_0) = O(\tau + \log(|\text{LCE}|/\tau))$.

Now consider the problem of answering a query of the form $\text{LCE}(i\tau, j\tau + \gamma)$ where $0 < \gamma < \tau$. By Lemma 3 we can obtain the fingerprint of any substring in $O(\tau)$ time. This allows us to use a similar approach to the first case. We find the largest ℓ such that $\phi(T[j\tau + \gamma \ldots (j+2^\ell)\tau + \gamma - 1]) = \phi_i^\ell$ and convert the current query into a new query, $\text{LCE}((i+2^\ell)\tau, (j+2^\ell)\tau + \gamma)$. As we have to apply Lemma 3 before every comparison, we obtain a total complexity of $O(\tau \log(|\text{LCE}|/\tau))$.

The general case can be reduced to one of the first two cases by scanning $O(\tau)$ characters in T. By Lemma 2, all fingerprint comparisons are correct with high probability and the result follows.

4 The Las-Vegas Data Structure

We now show Theorem 1(ii). The important observation is that when we compare the fingerprints of two strings during a query in Section 3, one of them is of the form $T[j\tau \ldots j\tau + \tau \cdot 2^\ell - 1]$ for some ℓ, j. Consequently, to ensure all queries are correctly computed, it suffices that ϕ is τ*-good*:

Definition 1. *A fingerprinting function, ϕ is τ-good on T iff*

$$\begin{aligned} &\phi(T[j\tau \ldots j\tau + \tau \cdot 2^\ell - 1]) = \phi(T[i \ldots i + \tau \cdot 2^\ell - 1]) \\ &\textit{iff}\ T[j\tau \ldots j\tau + \tau \cdot 2^\ell - 1] \;=\; T[i \ldots i + \tau \cdot 2^\ell - 1] \quad \textit{for all } (i,j,\ell). \qquad (2)\end{aligned}$$

In this section we give an algorithm which decides whether a given ϕ is τ-good on string T. The algorithm uses $O(n/\tau)$ space and takes $O(n \log n)$ time with high probability. By Lemma 2, a uniformly chosen ϕ is τ-good with high probability and therefore (by repetition) we can generate such a ϕ in the same time/space bounds. For brevity we assume that n and τ are powers-of-two.

4.1 The Algorithm

We begin by giving a brief overview of Algorithm 1. For each value of ℓ in ascending order (the outermost loop), Algorithm 1 checks (2) for all i, j. For some outermost loop iteration ℓ, the algorithm inserts the fingerprint of each block-aligned substring into a dynamic perfect dictionary, D_ℓ (lines 3-9). A substring is block-aligned if it is of the form, $T[j\tau \ldots (j + 2^\ell)\tau - 1]$ for some j (and block-unaligned otherwise). If more than one block-aligned substring has the same fingerprint, we insert only the left-most as a representative. For the first iteration, $\ell = 0$ we also build an Aho-Corasick automaton [1], denoted AC, with a pattern dictionary containing every block-aligned substring.

The second stage (lines 12-21) uses a sliding window technique, checking each time we slide whether the fingerprint of the current $(2^\ell\tau)$-length substring occurs in the dynamic dictionary, D_ℓ. If so we check whether the corresponding substrings match (if not a collision has been revealed and we abort). For $\ell > 0$, we use the fact that (2) holds for all i, j with $\ell - 1$ (otherwise, Algorithm 1 would have already aborted) to perform the check in constant time (line 18). I.e. if there is a collision it will be revealed by comparing the fingerprints of the left-half ($L'_i \neq L_k$) or right-half ($R'_i \neq R_k$) of the underlying strings. For $\ell = 0$, the check is performed using the AC automaton (lines 20-21). We achieve this by feeding T one character at a time into the AC. By inspecting the state of the AC we can decide whether the current τ-length substring of T matches any block-aligned substring.

Correctness We first consider all points at which Algorithm 1 may abort. First observe that if line 21 causes an abort then (2) is violated for $(i, k, 0)$. Second, if line 18 causes an abort either $L'_i \neq L_k$ or $R'_i \neq R_k$. By the definition of ϕ, in either case, this implies that $T[i \ldots i + \tau \cdot 2^\ell - 1] \neq T[k\tau \ldots k\tau + 2^\ell\tau - 1]$. By line 16, we have that $f'_i = f_k$ and therefore (2) is violated for (i, k, ℓ). Thus, Algorithm 1 does not abort if ϕ is τ-good.

It remains to show that Algorithm 1 always aborts if ϕ is not τ-good. Consider the total ordering on triples (i, j, ℓ) obtained by stably sorting (non-decreasing) by ℓ then j then i. E.g. $(1, 3, 1) < (3, 2, 3) < (2, 5, 3) < (4, 5, 3)$. Let (i^*, j^*, ℓ^*) be the (unique) smallest triple under this order which violates (2). We first argue that $(f_{j^*}, L_{j^*}, R_{j^*}, j^*)$ will be inserted into D_{ℓ^*} (and AC if $\ell^* = 0$). For a contradiction assume that when Algorithm 1 checks for f_{j^*} in D_{ℓ^*} (line 7, with $j = j^*, \ell = \ell^*$) we find that some $f_k = f_{j^*}$ already exists in D_{ℓ^*}, implying that $k < j^*$. If $T[j^*\tau \ldots j^*\tau + \tau 2^\ell - 1] \neq T[k\tau \ldots k\tau + \tau 2^\ell - 1]$ then $(j^*\tau, k, \ell^*)$ violates (2). Otherwise, (i^*, k, ℓ^*) violates (2). In either case this contradicts the minimality of (i^*, j^*, ℓ^*) under the given order.

Algorithm 1 Verifying a fingerprinting function, ϕ on string T

```
1: // $AC$ is an Aho-Corasick automaton and each $D_\ell$ is a dynamic dictionary
2: **for** $\ell = 0 \ldots \log_2(n/\tau)$ **do**
3:     // Insert all distinct block-aligned substring fingerprints into $D_\ell$
4:     **for** $j = 1 \ldots n/\tau - 2^\ell$ **do**
5:         $f_j \leftarrow \phi(T[j\tau \ldots (j+2^\ell)\tau - 1])$
6:         $L_j \leftarrow \phi(T[j\tau \ldots (j+2^{\ell-1})\tau - 1])$, $R_j \leftarrow \phi(T[(j+2^{\ell-1})\tau \ldots (j+2^\ell)\tau - 1])$
7:         **if** $\nexists (f_k, L_k, R_k, k) \in D_\ell$ such that $f_j = f_k$ **then**
8:             Insert $(f_j, L_j, R_j, j)$ into $D_\ell$ indexed by $f_j$
9:             **if** $\ell = 0$ **then** Insert $T[j\tau \ldots (j+1)\tau - 1]$ into $AC$ dictionary
10:    // Check for collisions between any block-aligned and unaligned substrings
11:    **if** $\ell = 0$ **then** Feed $T[1 \ldots \tau - 1]$ into $AC$
12:    **for** $i = 1 \ldots n - \tau \cdot 2^\ell + 1$ **do**
13:        $f'_i \leftarrow \phi(T[i \ldots i + \tau \cdot 2^\ell - 1])$
14:        $L'_i \leftarrow \phi(T[i \ldots i + \tau \cdot 2^{\ell-1} - 1])$, $R'_i \leftarrow \phi(T[(i+2^{\ell-1})\tau \ldots i + \tau \cdot 2^\ell - 1])$
15:        **if** $\ell = 0$ **then** Feed $T[i+\tau-1]$ into $AC$ // $AC$ now points at $T[i \ldots i+\tau-1]$
16:        **if** $\exists (f_k, L_k, R_k, k) \in D_\ell$ such that $f'_i = f_k$ **then**
17:            **if** $\ell > 0$ **then**
18:                **if** $(L'_i \neq L_k$ or $R'_i \neq R_k)$ **then abort**
19:            **else**
20:                Compare $T[i \ldots i+\tau-1]$ to $T[k\tau \ldots (k+1)\tau - 1]$ by inspecting $AC$
21:                **if** $T[i \ldots i+\tau-1] \neq T[k\tau \ldots (k+1)\tau - 1]$ **then abort**
```

We now consider iteration $i = i^*$ of the second inner loop (when $\ell = \ell^*$). We have shown that $(f_{j^*}, L_{j^*}, R_{j^*}, j^*) \in D_{\ell^*}$ and we have that $f'_{i^*} = f_{j^*}$ (so $k = j^*$) but the underlying strings are not equal. If $\ell = 0$ then we also have that $T[j^*\tau \ldots (j^*+1)\tau - 1]$ is in the AC dictionary. Therefore inspecting the current AC state, will cause an abort (lines 20-21). If $\ell > 0$ then as (i^*, j^*, ℓ^*) is minimal, either $L'_{i^*} \neq L_{j^*}$ or $R'_{i^*} \neq R_{j^*}$ which again causes an abort (line 18), concluding the correctness.

Time-Space Complexity We begin by upper bounding the space used and the time taken to performs all dictionary operations on D_ℓ for any ℓ. First observe that there are at most $O(n/\tau)$ insertions (line 8) and at most $O(n)$ look-up operations (lines 7,16). We choose the dictionary data structure employed based on the relationship between n and τ. If $\tau > \sqrt{n}$ then we use the deterministic dynamic dictionary of Ružić [27]. Using the correct choice of constants, this dictionary supports look-ups and insert operations in $O(1)$ and $O(\sqrt{n})$ time respectively (and linear space). As there are only $O(n/\tau) = O(\sqrt{n})$ inserts, the total time taken is $O(n)$ and the space used is $O(n/\tau)$. If $\tau \leq \sqrt{n}$ we use the Las-Vegas dynamic dictionary of Dietzfelbinger and Meyer auf der Heide [7]. If $\Theta(\sqrt{n}) = O(n/\tau)$ space is used for D_ℓ, as we perform $O(n)$ operations, every operation takes $O(1)$ time with high probability. In either case, over all ℓ we take $O(n \log n)$ total time processing dictionary operations.

The operations performed on AC fall conceptually into three categories, each totalling $O(n \log n)$ time. First we insert $O(n/\tau)$ τ-length substrings into the AC

dictionary (line 9). Second, we feed T into the automaton (line 11,15) and third, we inspect the AC state at most n times (line 20). We store AC in the standard compacted form with edge labels stored as pointers into T. This means that the space used is $O(n/\tau)$, the number of strings in the AC dictionary.

Finally we bound the time spent constructing fingerprints. We first consider computing f'_i (line 13) for $i > 1$. We can compute f'_i in $O(1)$ time from f'_{i-1}, $T[i-1]$ and $T[i + \tau \cdot 2^\ell]$. This follows immediately from the definition of ϕ. We can compute L'_i and R'_i analogously. Over all i, ℓ, this gives $O(n \log n)$ time. Similarly we can compute f_j from f_{j-1}, $T[(j-1)\tau \dots j\tau - 1]$ and $T[(j-1+2^\ell)\tau \dots (j+2^\ell)-1]$ in $O(\tau)$ time. Again this is analogous for L'_i and R'_i. Summing over all j, ℓ this gives $O(n \log n)$ time again. Finally observe that the algorithm only needs to store the current and previous values for each fingerprint so this does not dominate the space usage.

5 Longest Common Extensions on Two Strings

We now show how to efficiently reduce LCE queries between two strings to LCE queries on a single string. We generalize our notation as follows. Let P and T be strings of lengths n and m, respectively. Define $\mathrm{LCE}_{P,T}(i,j)$ to be the length of the longest common prefix of the substrings of P and T starting at i and j, respectively. For a single string P, we define $\mathrm{LCE}_P(i,j)$ as usual. We can always trivially solve the LCE problem for P and T by solving it for the string obtained by concatenating P and T. We show the following improved result.

Theorem 6. *Let P and T be strings of lengths m and n, respectively. Given a solution to the LCE problem on P using $s(m)$ space and $q(m)$ time and a parameter τ, $1 \leq \tau \leq n$, we can solve the LCE problem on P and T using $O(\frac{n}{\tau} + s(m))$ space and $O(\tau + q(m))$ time.*

For instance, plugging in Theorem 1(i) in Theorem 6 we obtain a solution using $O(\frac{n}{\tau} + \frac{m}{\sqrt{\tau}})$ space and $O(\tau)$ time. Compared with the direct solution on the concatenated string that uses $O(\frac{n+m}{\sqrt{\tau}})$ we save substantial space when $m \ll n$.

5.1 The Data Structure

The basic idea for our data structure is inspired by a trick for reducing constant factors in the space for the LCE data structures [9, Ch. 9.1.2]. We show how to extend it to obtain asymptotic improvements. Let P and T be strings of lengths m and n, respectively. Our data structure for LCE queries on P and T consists of the following information.

- A data structure that supports LCE queries for P using $s(m)$ space and $q(m)$ query time.
- An array A of length $\lfloor \frac{n}{\tau} \rfloor$ such that $A[i]$ is the maximum LCE value between any suffix of P and the suffix of T starting at position $i \cdot \tau$, that is, $A[i] = \max_{j=1\dots m} \mathrm{LCE}_{P,T}(j, i\tau)$.

– An array B of length $\lfloor \frac{n}{\tau} \rfloor$ such that $B[i]$ is the index in P of a suffix that maximizes the LCE value, that is, $B[i] = \arg\max_{j=1\ldots m} \mathrm{LCE}_{P,T}(j, i\tau)$.

Arrays A and B use $O(n/\tau)$ space and hence the total space is $O(n/\tau + s(m))$. We answer an $\mathrm{LCE}_{P,T}$ query as follows. Suppose that $\mathrm{LCE}_{P,T}(i, j) < \tau$. In that case we can determine the value of $\mathrm{LCE}_{P,T}(i, j)$ in $O(\tau)$ time by explicitly comparing the characters from position i in P and j in T until we encounter the mismatch. If $\mathrm{LCE}_{P,T}(i, j) \geq \tau$, we explicitly compare $k < \tau$ characters until $j + k \equiv 0 \pmod{\tau}$. When this occurs we can lookup a suffix of P, which the suffix $j + k$ of T follows at least as long as it follows the suffix $i + k$ of P. This allows us to reduce the remaining part of the $\mathrm{LCE}_{P,T}$ query to an LCE_P query between these two suffixes of P as follows.

$$\mathrm{LCE}_{P,T}(i, j) = k + \min\left(A\left[\frac{j+k}{\tau}\right], \mathrm{LCE}_P\left(i + k,\ B\left[\frac{j+k}{\tau}\right] \right) \right) .$$

We need to take the minimum of the two values, since, as shown by Example 3, it can happen that the LCE value between the two suffixes of P is greater than that between suffix $i+k$ of P and suffix $j+k$ of T. In total, we use $O(\tau + q(m))$ time to answer a query. This concludes the proof of Theorem 6.

Example 3. Consider the query $\mathrm{LCE}_{P,T}(2, 13)$ on the string P from Example 2 and

	1	2	3	4	$\underline{5}$	6	7	8	9	$\underline{10}$	11	12	13	14	$\underline{15}$	16	17	18	19	$\underline{20}$	21	22
$T =$	c	a	c	d	e	a	b	a	a	c	a	a	b	c	a	a	b	c	d	c	a	e

The underlined positions in T indicate the positions divisible by 5. As shown below, we can use the array $A = [0, 6, 4, 2]$ and $B = [16, 3, 11, 10]$.

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
i	iv	$P =$	d	b	c	a	a	b	c	a	b	c	a	a	b	c	a	c	$A[i]$	$B[i]$
1	5																	$\cancel{e}$	0	16
2	10				c	a	a	b	c	a	$\cancel{a}$								6	3
3	15												a	a	b	c	$\cancel{d}$		4	11
4	20											c	a	$\cancel{e}$					2	10

To answer the query $\mathrm{LCE}_{P,T}(2, 13)$ we make $k = 2$ character comparisons and find that

$$\begin{aligned} \mathrm{LCE}_{P,T}(2, 13) &= 2 + \min\left(A\left[\frac{13+2}{5}\right], \mathrm{LCE}_P\left(2 + 2, B\left[\frac{13+2}{5}\right] \right) \right) \\ &= 2 + \min(4, 5) = 6 . \end{aligned}$$

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18, 1975.

2. J. Allouche, M. Baake, J. Cassaigne, and D. Damanik. Palindrome complexity. *Theoret. Comput. Sci.*, 292(1):9–31, 2003.
3. A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. Algorithms*, 50(2):257–275, 2004.
4. D. Breslauer and Z. Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14(4):355–366, 1995.
5. C. J. Colbourn and A. C. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1-2):9–12, 2000.
6. R. Cole and R. Hariharan. Approximate String Matching: A Simpler Faster Algorithm. *SIAM J. Comput.*, 31(6):1761–1782, 2002.
7. M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. 17th ICALP, LNCS*, volume 443, pages 6–19, 1990.
8. J. Fischer and V. Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In *Proc. 17th CPM, LNCS*, volume 4009, pages 36–48, 2006.
9. D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge, 1997.
10. D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69:525–546, 2004.
11. L. Gąsieniec, R. Kolpakov, and I. Potapov. Space efficient search for maximal repetitions. *Theoret. Comput. Sci.*, 339(1):35 – 48, 2005.
12. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
13. L. Ilie, G. Navarro, and L. Tinta. The longest common extension problem revisited and applications to approximate string searching. *J. of Discrete Algorithms*, 8:418–428, 2010.
14. J. Jeuring. The Derivation of On-Line Algorithms, with an Application to Finding Palindromes. *Algorithmica*, 11(2):146–184, 1994.
15. J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.
16. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
17. R. Kolpakov and G. Kucherov. Searching for gapped palindromes. In *Proc. 19th CPM, LNCS*, volume 5029, pages 18–30, 2008.
18. G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27(2):557–582, 1998.
19. G. M. Landau and J. P. Schmidt. An Algorithm for Approximate Tandem Repeats. *J. Comput. Biol.*, 8(1):1–18, 2001.
20. G. M. Landau and U. Vishkin. Fast Parallel and Serial Approximate String Matching. *J. Algorithms*, 10:157–169, 1989.
21. L. Lu, H. Jia, P. Dröge, and J. Li. The human genome-wide distribution of DNA palindromes. *Funct. Integr. Genomics*, 7(3):221–7, 2007.
22. M. G. Main and R. J. Lorentz. An O (n log n) algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.
23. G. Manacher. A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. *J. ACM*, 22(3):346–351, 1975.
24. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoret. Comput. Sci.*, 410(8-10):900–913, 2009.

25. E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
26. S. J. Puglisi and A. Turpin. Space-time tradeoffs for Longest-Common-Prefix array computation. In *Proc. 19th ISAAC, LNCS*, volume 5369, pages 124–135, 2008.
27. M. Ružić. Uniform deterministic dictionaries. *ACM Trans. Algorithms*, 4:1–23, 2008.