Master Thesis

on

Algorithms for Longest Common Extensions

Jesper Kristensen
s062397
DTU Informatics
Technical University of Denmark

August 1, 2011

Contents

1 Introduction

The Longest Common Extension problem can be used in many algorithms for solving other algorithmic problems, e.g. the Landau-Vishkin algorithm for approximate string searching [8]. Optimal solutions exists for the problem, with constant query time and linear space. These theoretically optimal solutions are however not the best in practice, since they have large constant factors for both time and space usage. In average cases, a much simpler solution with worst case linear time, average case constant time, and no preprocessing has significantly better practical performance. This algorithm is ideal when only average case performance is relevant. In situations where we need both average case and worst case performance to be good, none of the existing solutions are ideal.

We have found two new algorithms. They both have better than linear worst case query times, and achieves significantly better average case practical query times compared to the theoretically best algorithms. The best of these two uses string fingerprinting to achieve a time/space-tradeoff with $O(k \cdot n^{1/k})$ worst case query time, constant average case query time, and $O(kn)$ space for k between one and $\log n$.

In Section 3 we theoretically describe and analyze the LCE algorithms introduced here in Section 1, and in Section 4 we analyze results from practical C++ implementations of these algorithms.

We also look at how to apply the LCE problem to Ziv-Lempel compressed strings, such that we can answer LCE queries fast while only using space relative to the compressed size of the string. In Section 5 we describe LZ compression, and how we can convert the simple linear time LCE algorithm to work on compressed strings. It uses space linear to the size of the compressed string, and it has worst case query time linear to the size of the uncompressed string, and average case query time $O(\log \log N)$. In Section 6 we look at how we can use the tree structure of LZ compression to get better query times on compressed strings.

1.1 The LCE Problem

Given a string s of length n and a pair of indexes $1 \leq i \leq n$ and $1 \leq j \leq n$, the Longest Common Extension of i and j on s, $LCE_s(i, j)$, is the length of the longest common prefix of $suff_i$ and $suff_j$. If for example $s = abbababba$, $i = 4$ and $j = 6$, then $suff_4 = ababba$ and $suff_6 = abba$. The longest common prefix of these two suffixes is ab, which has length 2, and therefore we have $LCE_s(4, 6) = 2$.

The LCE problem is to preprocess a string in order to allow for a large number of LCE queries on different pairs (i, j), such that the queries are efficient.

Given a string length n and an alphabet size σ, we define average case query time as the average of query times over all $\sigma^n \cdot n^2$ combinations of strings and query inputs.

1.2 Existing Results

One way of solving the LCE problem without any preprocessing uses $O(|LCE(i,j)|)$ query time. We call this algorithm DirectComp. For a query $LCE(i,j)$, the algorithm compares $s[i]$ to $s[j]$, then $s[i+1]$ to $s[j+1]$ and so on, until the two characters differ, or the end of the string is reached. The worst case query time is $O(n)$ on a string of length n, and the average case query time is $O(1)$, because the average case LCE value over all possible inputs of a given string of length n and alphabet size σ is $O(1/(\sigma-1)) = O(1)$ [1].

The LCE problem can be solved optimally with $O(1)$ worst case query time, using $O(n)$ space and $O(sort(n,\sigma))$ preprocessing time. Two different ways of doing this exists.

One method, which we call SuffixNca, uses constant time Nearest Common Ancestor queries on a suffix tree. The LCE of two indexes i and j is defined as the length of the longest common prefix of $suff_i$ and $suff_j$. In a suffix tree, the path from the root to L_i has label $suff_i$ (likewise for j), and no two child edge labels of the same node will have the same first character. The longest common prefix of the two suffixes will therefore be the path label from the root to the nearest common ancestor of L_i and L_j. This gives $LCE_s(i,j) = D[NCA_{\mathcal{T}}(L_i, L_j)]$.

The other method, which we call LcpRmq, uses constant time Range Minimum Queries on a Longest Common Prefix array. The LCP array contains the length of the longest common prefixes of each pair of neighbor suffixes in SA. The length of the longest common prefix of two arbitrary suffixes in SA can be found as the minimum of all LCP values of neighbor suffixes between the two desired suffixes, because SA lists the suffixes in lexicographical ordering. I.e. $LCE(i,j) = LCP[RMQ_{LCP}(SA^{-1}[i]+1, SA^{-1}[j])]$, where $SA^{-1}[i] < SA^{-1}[j]$.

1.2.1 Existing Practical Results

Ilie et al. [1] have looked at a number of real world texts as well as texts of randomly generated characters, and found that all the texts they examined each has an average LCE of at most one, over all n^2 input pairs. Therefore it is interesting to have LCE algorithms, which perform good on average when the LCE value is small.

Both SuffixNca and LcpRmq have the same asymptotic space and times. In practice, LcpRmq is the best of the two [1]. The constant factor for average case query time of DirectComp is much smaller than the constant factor for query time of LcpRmq, thus DirectComp is better in practice on average case inputs than the theoretically best LcpRmq.

1.3 Our Fingerprinting Algorithm

We present a new LCE algorithm based on string fingerprinting. We call our algorithm Fingerprint$_k$, where k is a parameter $1 \leq k \leq \lceil \log n \rceil$, which describes the number of levels used[1].

[1] All logarithms are base two.

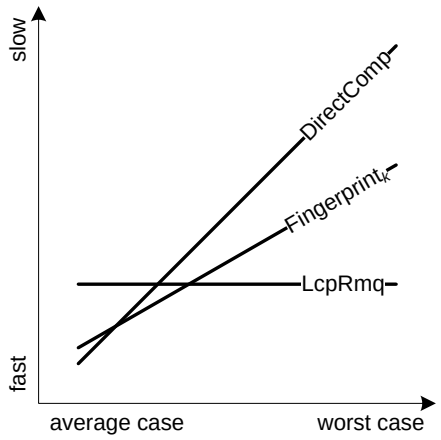


Figure 1: Graphical comparison of query times for different LCE algorithms.

Theorem 1. *For a string s of length n and alphabet size σ, the* FINGERPRINT$_k$ *algorithm, where k is a parameter $1 \leq k \leq \lceil \log n \rceil$, can solve the LCE problem in $O(k \cdot n^{1/k})$ worst case query time and $O(1)$ average case query time using $O(k \cdot n)$ space and $O(sort(n, \sigma) + k \cdot n)$ preprocessing time.*

The exact worst case performance of our solution depends on the amount of space you are willing to use.

Corollary 1. FINGERPRINT$_1$ *is equivalent to* DIRECTCOMP *with $O(n)$ space and $O(n)$ query time.*

Corollary 2. FINGERPRINT$_2$ *uses $O(n)$ space and $O(\sqrt{n})$ query time. It has two levels, where one uses a table of fingerprints and the other uses the original string.*

Corollary 3. FINGERPRINT$_{\lceil \log n \rceil}$ *uses $O(n \cdot \log n)$ space and $O(\log n)$ query time. The data structure is equivalent to the one generated by Karp-Miller-Rosenberg [3], and a query only needs to do one comparison at each level.*

To preprocess the $O(k \cdot n)$ fingerprints used by our algorithm, we can use Karp-Miller-Rosenberg [3], which takes $O(n \log n)$ time. For $k = o(\log n)$, we can speed up preprocessing to $O(sort(n, \sigma) + k \cdot n)$ by using the Suffix Array and Longest Common Prefix array.

In practice, existing state of the art solutions are either good in worst case, while poor in average case (LCPRMQ), or good in average case while poor in worst case (DIRECTCOMP). Our FINGERPRINT$_k$ solution targets a worst case vs. average case query time tradeoff between these two extremes. Figure 1 illustrates the relative query time performances of these solutions. Our solution is almost as fast as DIRECTCOMP on an average case input, and it is significantly

faster than DirectComp on a worst case input. Compared to LcpRmq, our solution has a significantly better performance on an average case input, but its worst case performance is not as good as that of LcpRmq. The space usage for LcpRmq and Fingerprint$_k$ are approximately the same when $k = 6$.

Our algorithm is fairly simple. Though it is slightly more complicated than DirectComp, it does not use any of the advanced algorithmic techniques required by LcpRmq and SuffixNca.

1.4 Our Table Lookup Algorithm

We also present another new LCE algorithm, which we call DirectComp-Lookup.

Theorem 2. *For a string s of length n, the* DirectCompLookup *algorithm can solve the LCE problem in $O(t)$ worst case query time and $O(1)$ average case query time using $O(n^2/t)$ space and $O(n^2)$ preprocessing time, where t is a parameter $1 \leq t \leq n$.*

The algorithm stores LCE values for some of the n^2 possible query inputs on a given string. It uses DirectComp to move towards one of these stored values, and stops when it reaches one. The DirectCompLookup algorithm is very simple, but the tradeoff between worst case query time and space is not very favorable, and preprocessing the data structure takes $O(n^2)$ time unless we use another LCE algorithm in the preprocessing.

2 Preliminaries

String notation. Let s be a string of length n. Then $s[i]$ is the i'th character of s, and $s[i\,..\,j]$ is a substring of s containing characters $s[i]$ to $s[j]$, both inclusive. That is, $s[1]$ is the first character of s, $s[n]$ is the last character, and $s[1\,..\,n]$ is the entire string. The suffix of s starting at index i is written $\mathit{suff}_i = s[i\,..\,n]$.

Tree notation. The ancestors of a node u is u itself as well as any node, which is a parent of an ancestor of u. A path in a tree begins at node u and ends at node v, where u is an ancestor of v. The length of a path is the number of edges between nodes in the path. The depth of a node u is the length of the path from the root to u, and the height of u is the length of the longest path from u to a descendant of u. E.g. the depth of the root and the height of a leaf is both zero.

Sorting complexity. The time it takes to sort n numbers within an universe of size σ is written as $\mathit{sort}(n, \sigma)$.

Suffix tree. A suffix tree $\mathcal{T}$ encodes all suffixes of a string s of length n with alphabet σ. The tree has n leaves named L_1 to L_n, one for each suffix of s.

Each edge is labeled with a substring of s, such that for any $1 \leq i \leq n$, the concatenation of labels on edges on the path from the root to L_i gives suff_i. Any internal node must have more than one child, and the labels of two child edges must not share the same first character. The string depth $D[v]$ of a node v is the length of the string formed when concatenating the edge labels on the path from the root to v. The tree uses $O(n)$ space, and building it takes $O(\mathit{sort}(n, \sigma))$ time [2].

There are n leaves in a suffix tree, and since each internal node has at least two children, the tree has less than $2n$ nodes. Each edge uses $O(1)$ space, since the label is a substring of s and it can therefore be represented by two indexes of s. The suffix tree therefore takes $O(n)$ space.

SA and LCP. For a string s of length n with alphabet size σ, the Suffix Array, SA, is an array of length n, which encodes the lexicographical ordering of all suffixes of s. The lexicographically smallest suffix is $\mathit{suff}_{SA[1]}$, the lexicographically largest suffix is $\mathit{suff}_{SA[n]}$, and the lexicographically i'th smallest suffix is $\mathit{suff}_{SA[i]}$. The inverse Suffix Array, SA^{-1}, describes where a given suffix is in the lexicographical order. Suffix suff_i is the lexicographically $SA^{-1}[i]$'th smallest suffix.

The Longest Common Prefix Array, LCP, describes the length of longest common prefixes of neighboring suffixes in SA. $LCP[i]$ is the length of the longest common prefix of $\mathit{suff}_{SA[i-1]}$ and $\mathit{suff}_{SA[i]}$ for $2 \leq i \leq n$. The first element $LCP[1]$ is always zero.

Building the SA, $\mathrm{SA}^{-1}$and LCP arrays takes $O(\mathit{sort}(n, \sigma))$ time [2].

NCA. The common ancestors of two nodes u and v is the intersection between the ancestors of u and the ancestors of v. The Nearest Common Ancestor of two nodes u and v is the common ancestor of u and v of greatest depth. An NCA query can be answered in $O(1)$ time with $O(n)$ space and preprocessing time in a static tree with n nodes [4].

RMQ. The range minimum of i and j on an array A is the index of a minimum element in $A[i, j]$, i.e. $RMQ_A(i, j) = \arg\min_{k \in \{i,\ldots,j\}}\{A[k]\}$. A Range Minimum Query on a static array of n elements can be answered in $O(1)$ time with $O(n)$ space and preprocessing time [5].

I/O's and Cache-Oblivious Model. The I/O model describes the number of memory blocks an algorithm moves between two layers of a layered memory architecture, where the size of the internal memory layer is M words, and data is moved between internal and external memory in blocks of B words. In the cache-oblivious model, the algorithm has no knowledge of the values of M and B.

Predecessor. Given a set of n objects, where the key of each object is a natural number between one and U, the predecessor of a natural number x is

an object with maximal key, whose key is at most x. We can solve the static Predecessor problem in $O(\log \log U)$ time using $O(n)$ space [6].

Level Ancestor. The Level Ancestor at level ℓ of node u, $LA(\ell, u)$, is the ancestor of u, whose depth is ℓ. We can preprocess a static tree in $O(n)$ time and space to answer Level Ancestor queries in $O(1)$ time [7].

Heavy Path Decomposition The weight of a node u is the size of the subtree rooted at u. For each node u, a child of greatest weight is marked heavy, and all other children are marked light. The root is always marked light. An edge going from a node to a heavy child node is heavy, and an edge going from a node to a light child node is light. A path of only heavy edges is a heavy path, and the combined length of all heavy paths in a tree of n nodes is less than n. The markings of heavy and light edges and nodes in a tree is a Heavy Path Decomposition. Every simple path in a tree of size n contains at most $O(\log n)$ light edges, and we can construct a Heavy Path Decomposition in $O(n)$ time [4].

Ladder Decomposition To make a ladder decomposition of a tree, we first create a ladder of nodes on the path from the root of the tree to the leaf of greatest depth. We then remove the nodes in the ladder from the tree and repeat this operation recursively for all remaining subtrees, until all nodes are part of a ladder. Node u's ladder is the ladder which contains u at this point. Finally, we double the number of nodes in each ladder by extending it towards the root of the tree. If we reach the root before we have doubled the number of nodes, we stop at the root. Each node may thus be in multiple ladders.

For any path in a tree of n nodes, we can find $O(\log n)$ ladders, which contain all the nodes in the path. If a node u has height h, u's ladder contains all the ancestors of u whose distance to u is at most $h + 1$. The combined size of the ladders is at most $2n$. [7]

3 LCE Algorithms

This section describes existing and new algorithms for solving the LCE problem. It contains a theoretical analysis of their asymptotic query times and space requirements, shown in summary in Table 1. Section 4 describes the results from experiments on practical implementations of these algorithms.

3.1 Existing Algorithms

In this section we look at the existing LCE algorithms described in Section 1.2 with a bit more details.

Algorithm	Space	Query time	Preprocessing
SuffixNca	$O(n)$	$O(1)$	$O(sort(n, \sigma))$
LcpRmq	$O(n)$	$O(1)$	$O(sort(n, \sigma))$
DirectComp	$O(1)$	$O(n)$	$O(1)$
DirectCompLookup, $1 \leq t \leq n$	$O(n^2/t)$	$O(t)$	$O(n^2)$
$t = 1$	$O(n^2)$	$O(1)$	$O(n^2)$
Fingerprint$_k$, $1 \leq k \leq \lceil \log n \rceil$	$O(k \cdot n)$	$O(k \cdot n^{1/k})$	$O(sort(n, \sigma) + k \cdot n)$
$k = \lceil \log n \rceil$	$O(n \log n)$	$O(\log n)$	$O(n \log n)$

Table 1: LCE algorithms with their space requirements, worst case query times and preprocessing times. Average case query times are $O(1)$ for all shown algorithms.

3.1.1 DirectComp

DirectComp solves the LCE problem without any preprocessing in $O(|LCE(i, j)|)$ query time. For a query $LCE(i, j)$, the algorithm compares $s[i + v]$ to $s[j + v]$ in a loop, for v starting at zero and incrementing, until the two characters differ, or the end of the string is reached, at which point the LCE value is v. DirectComp does not need to check if the end of the string is reached, i.e. $i + v > n$ or $j + v > n$. Instead we can preprocess the string by adding a special character \$ to the end of the string, which does not occur anywhere else in the string. When the algorithm compares $s[n] =$ \$ to any other character, it will stop, as the characters differ. This can give better practical performance compared to checking $i + v > n$ and $j + v > n$ in each iteration of the algorithm. The query time is $O(|LCE(i, j)|)$, which in worst case is $O(n)$ on a string of length n, and in average case is $O(1)$.

3.1.2 SuffixNca: NCA on a Suffix Tree

LCE of two indexes i and j is defined as the length of the longest common prefix of $suff_i$ and $suff_j$. In a suffix tree, the path from the root to L_i has label $suff_i$ (likewise for j). The path label on the path from the root to the nearest common ancestor of L_i and L_j is therefore a common prefix of the two suffixes. Since two child edges of the same node never have the same first character on their edge labels, the path label on the path from the root to $NCA_{\mathcal{T}}(L_i, L_j)$ is not only a common prefix, but the longest common prefix of $suff_i$ and $suff_j$. We can therefore find the LCE value as $LCE_s(i, j) = D[NCA_{\mathcal{T}}(L_i, L_j)]$, where $\mathcal{T}$ is the suffix tree for s, and $D[v]$ is the label depth of node v in the tree.

For a string of length n with alphabet size σ, we can preprocess SuffixNca in $O(sort(n, \sigma))$ time by building the suffix tree, preprocessing it for NCA and building the array of label depths. The query time is $O(1)$, since looking up L_i and L_j, doing an NCA query, and looking up the label depth in D all takes constant time.

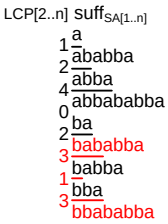


Figure 2: For $s = abbababba$, $LCE(2, 3)$ is the longest common prefix of $bababba$ and $bbababba$. There are two other suffixes between these in the lexicographical ordering, and the longest common prefixes between each pair of neighbors are 3, 1 and 3. Therefore $LCE(2, 3) = \min\ \{3, 1, 3\} = 1$.

3.1.3 LcpRmq: RMQs on a LCP Array

The LCP array contains the length of the longest common prefixes of each pair of neighbor suffixes in SA. The length of the longest common prefix of two arbitrary suffixes in SA can be found as the minimum of all LCP values of neighbor suffixes between the two desired suffixes, because SA lists the suffixes in lexicographical ordering. I.e. $LCE(i, j) = LCP[RMQ_{LCP}(SA^{-1}[i] + 1, SA^{-1}[j])]$, where $SA^{-1}[i] < SA^{-1}[j]$. An example is shown in Figure 2. Preprocessing time is $O(sort(n, \sigma))$, space requirement is $O(n)$ and query time is $O(1)$.

Different algorithms exists for solving the RMQ problem, and we can solve it in optimal $O(1)$ time and $O(n)$ space. But like for the LCE problem, the asymptotically best RMQ solution may not be the best RMQ solution in practice. We examine the following RMQ solutions, where RMQ$<p, q>$ has $O(p)$ space and preprocessing time, and $O(q)$ query time:

RMQ$<1, n>$ walks through the array directly in $O(n)$ time and does not use any preprocessing (except for transforming LCE into RMQ).

RMQ$<n, 1>$ preprocesses the array in $O(n)$ space in order to do queries in $O(1)$ time. This is a twolevel data structure as shown in Figure 3. The array is split into blocks of size $O(\log n)$. The top level is an array of length $O(n/\log n)$, where each element contains the minimum value of one block. The top level uses a RMQ$<n \log n, 1>$-solution, and the lower level precomputes all answers for each block, which is done in $O(n)$ space, since there are only a limited number of different block types of size $O(\log n)$.

RMQ$<n, \log n>$ is the same as RMQ$<n, 1>$ except that the lower level is replaced with a direct search through a $O(\log n)$ length block.

3.1.4 Combining DirectComp and LcpRmq

Ilie et al. [1] suggests combining DirectComp and RMQ$<1, n>$, by using RMQ$<1, n>$ whenever the two suffixes we want to compare are close to each

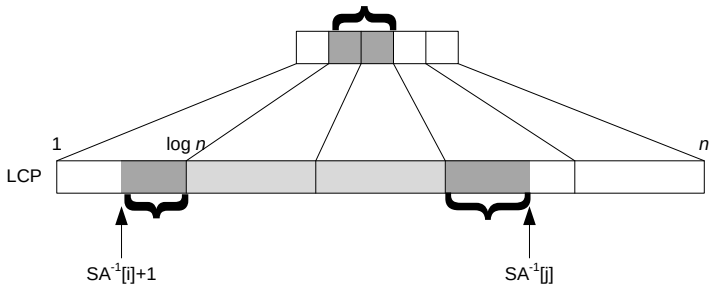


Figure 3: Twolevel Range Minimum Query. The range minimum is calculated as the minimum of one query in the top level and two queries in the lower level.

other in SA, and otherwise use DirectComp. However that still gives $O(n)$ worst case query time. For example in the string $s = aaaaa...$, the query $LCE(1, n/2)$ would require $\Omega(n)$ time for both DirectComp and RMQ<1, n>.

We can instead combine DirectComp and RMQ<n, 1>, which we call DirectComp+LcpRmq. DirectComp+LcpRmq first uses DirectComp until a specific cutoff and then switches over to RMQ<n, 1> instead. This could have both the good average case query time of DirectComp and the good worst case query time of LcpRmq. However it still keeps the large space requirements of LcpRmq, and it is not simple to implement like DirectComp.

3.2 The DirectCompLookup Algorithm

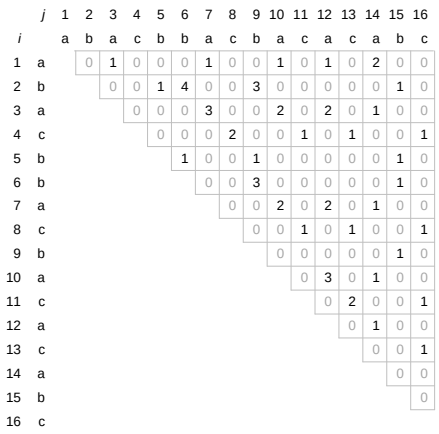
The input to the LCE problem is two numbers $1 \leq i < j \leq n$ [2], so there are $n \cdot (n-1)/2 = O(n^2)$ different input possibilities in total. Because of this limited space of possible query inputs, we can tabulate a number of LCE values, and use those to speed up queries.

3.2.1 DirectLookup

We can store results for all $O(n^2)$ possible queries in $O(n^2)$ space, and a query will then be a single table lookup in $O(1)$ time. We call this algorithm DirectLookup. A result space table for DirectLookup is shown in Figure 4, and Figure 5 shows how DirectComp moves through that table.

We can preprocess the table in in $O(n^2)$ time, since each value can be calculated in constant time like follows, if $LCE(i+1, j+1)$ is always calculated

[2] We can assume that $i < j$ without loss of generality, since $LCE(i, j) = LCE(j, i)$



	j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i		a	b	a	c	b	b	a	c	b	a	c	a	c	a	b	c
1	a	0	1	0	0	0	1	0	0	1	0	1	0	2	0	0	
2	b		0	0	1	4	0	0	3	0	0	0	0	0	1	0	
3	a			0	0	0	3	0	0	2	0	2	0	1	0	0	
4	c				0	0	0	2	0	0	1	0	1	0	0	1	
5	b					1	0	0	1	0	0	0	0	0	1	0	
6	b						0	0	3	0	0	0	0	0	1	0	
7	a							0	0	2	0	2	0	1	0	0	
8	c								0	0	1	0	1	0	0	1	
9	b									0	0	0	0	0	1	0	
10	a										0	3	0	1	0	0	
11	c											0	2	0	0	1	
12	a												0	1	0	0	
13	c													0	0	1	
14	a														0	0	
15	b															0	
16	c																

Figure 4: The result space for $LCE_s(i, j)$ for $1 \leq i < j \leq n$ where $s = abacbbacbacacabc$.

before $LCE(i, j)$ for any i and j:

$$LCE(i,j) = \begin{cases} 0 & \text{if } s[i] \neq s[j] \\ 1 & \text{if } s[i] = s[j] \wedge (i = n \vee j = n) \\ LCE(i+1, j+1) + 1 & \text{otherwise} \end{cases}$$

3.2.2 DirectComp and DirectLookup Combined

DirectComp and DirectLookup preprocesses and stores no results and all results respectively. If we only store some results, we can use DirectComp until we reach a stored result, and then retrieve this result using DirectLookup. If we store every t'th row, as shown in Figure 6, the space usage will be $O(n^2/t)$ and the worst case query time will be $O(t)$. We call this algorithm DirectCompLookup.

3.3 The Fingerprint$_k$ Algorithm

Our Fingerprint$_k$ algorithm generalizes DirectComp. It compares characters starting at positions i and j, but instead of comparing individual characters, it compares fingerprints of substrings. Given fingerprints of all substrings of length t, our algorithm can compare two t-length substrings in constant time.

3.3.1 Definition of a Fingerprint

Given a string s, the fingerprint $F_t[i]$ is a natural number identifying the substring $s[i \mathinner{..} i + t - 1]$ among all t-length substrings of s. We assign fingerprints such

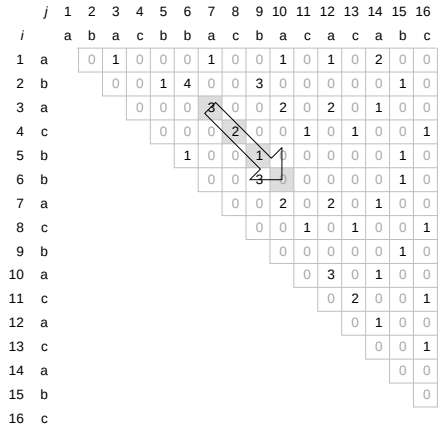


Figure 5: $LCE(3, 7)$ using DirectComp.

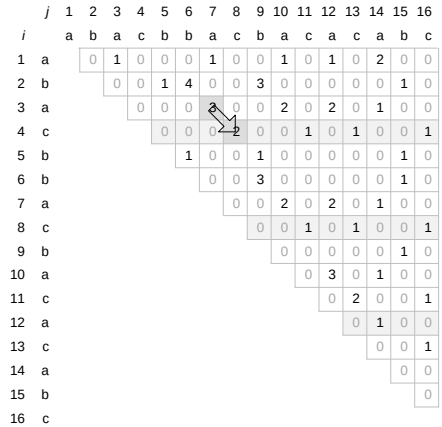


Figure 6: $LCE(3, 7)$ with distance $t = 4$ between each stored row.

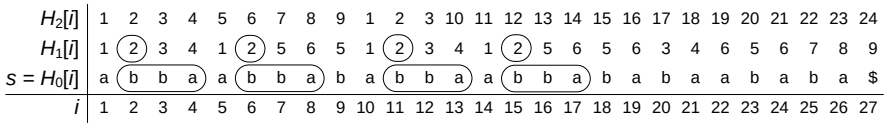


Figure 7: FINGERPRINT$_k$ data structure for $s = abbaabbababbaabbababaababa\$$, $n = 27$, $k = 3$, $t_1 = n^{1/3} = 3$ and $t_2 = n^{2/3} = 9$. All substrings *bba* are highlighted with their 3-length fingerprint 2.

that for any i, j and t, $F_t[i] = F_t[j]$ if and only if $s[i\,..\,i+t-1] = s[j\,..\,j+t-1]$. In other words, if two substrings of s have the same length, they have the same fingerprints if and only if the substrings themselves are the same.

At the end of a string when $i + t - 1 > n$, we define $F_t[i]$ by adding extra characters to the end of the string as needed.

3.3.2 Data Structure

The FINGERPRINT$_k$ data structure for a string s of length n, where k is a parameter $1 \le k \le \lceil \log n \rceil$, consists of k natural numbers t_0, ..., t_{k-1} and k tables H_0, ..., H_{k-1}, each of length n. For each ℓ where $0 \le \ell \le k-1$, $t_\ell = \Theta(n^{\ell/k})$ and table H_ℓ contains fingerprints of all t_ℓ-length substrings of s, such that $H_\ell[i] = F_{t_\ell}[i]$. We always have $t_0 = n^{0/k} = 1$, such that H_0 is the original string s. The last character of the string must be a special character \$, which does not occur anywhere else in the string. An example is shown in Figure 7.

Lemma 1. *The* FINGERPRINT$_k$ *data structure takes* $O(k \cdot n)$ *space.*

Proof. Each of the k tables stores n fingerprints of constant size. □

3.3.3 Query

To perform a LCE query using the FINGERPRINT$_k$ data structure, start with $v = 0$ and $\ell = 0$, then do the following steps:

1. As long as $H_\ell[i+v] = H_\ell[j+v]$, increment v by t_ℓ, increment ℓ by one, and repeat this step unless and $\ell = k-1$.
2. As long as $H_\ell[i+v] = H_\ell[j+v]$, increment v by t_ℓ and repeat this step.
3. Stop and return v when $\ell = 0$, otherwise decrement ℓ by one and go to step two.

An example of a query is shown in Figure 8.

Lemma 2. *The* FINGERPRINT$_k$ *query algorithm given above finds the correct LCE value.*

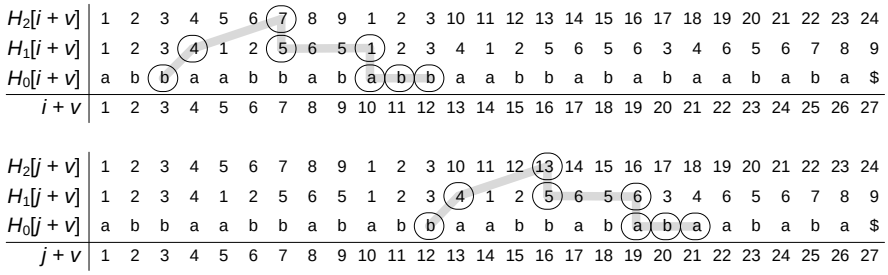


Figure 8: FINGERPRINT$_k$ query for $LCE(3, 12)$ on the data structure of Figure 7. The top half shows how $H_\ell[i + v]$ moves through the data structure, and the bottom half shows $H_\ell[j + v]$.

Proof. At each step of the algorithm $v \leq LCE(i, j)$, since the algorithm only increments v by t_ℓ when it has found two matching fingerprints, and fingerprints of two substrings of the same length are only equal if the substrings themselves are equal. When the algorithm stops, it has found two fingerprints, which are not equal, and the length of these substrings is $t_\ell = 1$, therefore $v = LCE(i, j)$.

The algorithm never reads $H_\ell[x]$, where $x > n$, because the string contains a unique character \$ at the end. This character will be at different positions in the substrings whose fingerprints are the last t_ℓ elements of H_ℓ. These t_ℓ fingerprints will therefore be unique, and the algorithm will not continue at level ℓ after reading one of them. □

Lemma 3. *The worst case query time for* FINGERPRINT$_k$ *is* $O(k \cdot n^{1/k})$*, and the average case query time is* $O(1)$*.*

Proof. Step one takes $O(k)$ time. In step two and three, the number of remaining characters left to check at level ℓ is $O(n^{(\ell+1)/k})$, since the previous level found two differing substrings of that length (at the top level $\ell = k - 1$ we have $O(n^{(\ell+1)/k}) = O(n)$). Since we can check $t_\ell = \Theta(n^{\ell/k})$ characters in constant time at level ℓ, the algorithm uses $O(n^{(\ell+1)/k})/\Theta(n^{\ell/k}) = O(n^{1/k})$ time at that level. Over all k levels, $O(k \cdot n^{1/k})$ query time is used.

At each step except step three, the algorithm increments v. Step three is executed the same number of times as step one, in which v is incremented. The query time is therefore linear to the number of times v is incremented, and it is thereby $O(v)$. The query time is thus $O(1)$ in average case, where $v = O(1)$. □

3.3.4 Average Case Optimization

We could have left out step one of the query algorithm in Section 3.3.3 and started with $\ell = k - 1$, which would change the query pattern as shown in Figure 9. This would keep the asymptotic worst case query time of $O(k \cdot n^{1/k})$, while it might improve practical worst case query time, but it would increase our average case query time to $O(k)$.

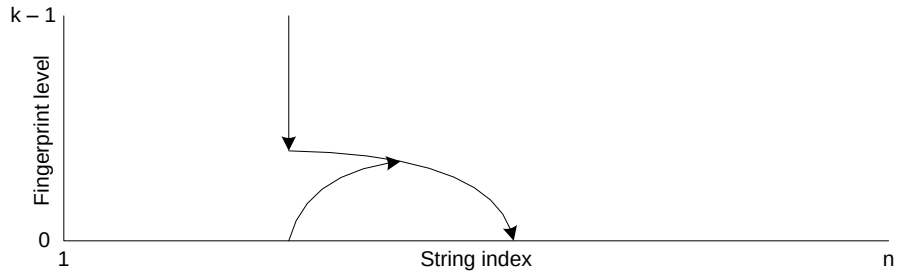


Figure 9: Two ways of querying the fingerprint levels, one starting from level $\ell = k - 1$ and one starting at level $\ell = 0$.

In step one of our query algorithm we perform one comparison at each level. We could instead do up to $O(n^{1/k})$ comparisons at each level without affecting asymptotic times.

3.3.5 Preprocessing

The tables of fingerprints use $O(k \cdot n)$ space. In the case with $k = \lceil \log n \rceil$ levels, the data structure is the one generated by Karp-Miller-Rosenberg [3]. This data structure can be constructed in $O(n \log n)$ time. With $k < \lceil \log n \rceil$ levels, KMR can be adapted, but it still uses $O(n \log n)$ preprocessing time.

We can preprocess the data structure in $O(sort(n, \sigma) + k \cdot n)$ time using the SA and LCP arrays. First create the SA and LCP arrays. Then preprocess each of the k levels using the following steps:

1. Loop through the n substrings of length t_ℓ in lexicographically sorted order by looping through the elements of SA.

2. Assign an arbitrary fingerprint to the first substring.

3. If the current substring $s[SA[i] \,..\, SA[i] + t_\ell - 1]$ is equal to the substring examined in the previous iteration of the loop, give the current substring the same fingerprint as the previous substring, otherwise give the current substring a new unused fingerprint. The two substrings are equal when $LCE[i] \geq t_\ell$.

An example is shown in Figure 10.

Lemma 4. *The preprocessing algorithm described above generates the data structure described in Section 3.3.2.*

Proof. We always assign two different fingerprints whenever two substrings are different, because whenever we see two differing substrings, we change the fingerprint to a value not previously assigned to any substring.

Subst.	$H_\ell[i]$	i
a	1	9
aba	2	4
abb	3	6
abb	3	1
ba	5	8
bab	6	3
bab	6	5
bba	8	7
bba	8	2

Figure 10: The first column lists all substrings of $s = abbababba$ with length $t_\ell = 3$. The second column lists fingerprints assigned to each substring. The third column lists the position of each substring in s.

We always assign the same fingerprint whenever two substrings are equal, because all substrings, which are equal, are grouped next to each other, when we loop through them in lexicographical order. □

Lemma 5. *The preprocessing algorithm described above takes $O(sort(n, \sigma) + k \cdot n)$ time.*

Proof. We first construct the SA and LCP arrays, which takes $O(sort(n, \sigma))$ time [2]. We then preprocess each of the k levels in $O(n)$ time, since we loop through n substrings, and comparing neighboring substrings takes constant time when we use the LCE array. The total preprocessing time becomes $O(sort(n, \sigma) + k \cdot n)$. □

3.3.6 Cache Optimization

Horizontal, intra-level optimization The amount if I/O's used by FINGERPRINT$_k$ is $O(k \cdot n^{1/k})$. However if we structure our tables of fingerprints differently, we can improve the number of I/O's to $O(k(n^{1/k}/B + 1))$ in the cache-oblivious model. Instead of storing the fingerprint of $s[i \,..\, i + t_\ell - 1]$ at $H_\ell[i]$, we can store it at $H_\ell[((i-1) \mod t_\ell) \cdot \lceil n/t_\ell \rceil + \lfloor (i-1)/t_\ell \rfloor + 1]$. This will group all used fingerprints at level ℓ next to each other in memory, such that the amount of I/O's at each level is reduced from $O(n^{1/k})$ to $O(n^{1/k}/B)$.

The size of each fingerprint table will grow from $|H_\ell| = n$ to $|H_\ell| = n + t_\ell$, because the rounding operations may introduce one-element gaps in the table after every n/t_ℓ elements. We achieve the greatest I/O improvement when k is small. When $k = \lceil \log n \rceil$, this cache optimization gives no difference in the amount of I/O's.

Vertical, inter-level optimization For larger values of k, we can attempt to optimize memory access across different levels. When we move down through the levels for $k = \lceil \log n \rceil$, we can continue in two directions after reading $H_\ell[i]$: $H_{\ell-1}[i]$ or $H_{\ell-1}[i + t_\ell]$. Neither of these positions are next to $H_\ell[i]$ in memory.

Algorithm	Space
DirectComp	0 words
DirectLookup	$2n^2$ words [a]
DirectCompLookup	$2n^2/t + n$ words
LcpRmq	ca. $5.6n$ words [b]
Fingerprint$_2$	n words
Fingerprint$_3$	$2n$ words
Fingerprint$_{\lceil \log n \rceil}$	$n \cdot \lceil \log n \rceil$ words

[a] The factor two comes from using shift instead of multiplication, which may double the size depending on n.

[b] Space use for LcpRmq is $2n + 2nb + (bs \cdot (bs+1)/2 + 1) \cdot C_{bs} + (nb+1) \cdot \lfloor \log nb \rfloor \approx 5.6n$, where $bs = \lceil \log n/4 \rceil$, $nb = (n-1)/bs+1$ and C_i is the i't Catalan Number.

Table 2: Space requirements for the LCE algorithms, where n is the number of characters, a character is one byte and a word is four bytes.

We can optimize for the case where we reach $H_{\ell-1}[i]$ by flipping the direction in which we store the fingerprints, such that all fingerprints $H_\ell[i]$ for all ℓ and a specific i are stored next to each other. This will not improve the asymptotic I/O bound, but could improve practical query times.

3.4 Space Usage

Table 2 shows the space requirement of each of our analyzed algorithms. The size of the input string of $n/4$ words is not included, and we ignore constant terms.

4 Experimental Results

In this section we show results of actual performance measurements. The measurements were done on a Windows 23-bit machine with an Intel P8600 CPU (3 MB L2, 2.4 GHz) and 4 GB RAM. The code was compiled using GCC 4.5.0 with `-O3`.

4.1 Tested Algorithms

We implemented different variants of DirectComp, LcpRmq, DirectCompLookup and Fingerprint$_k$ in C++ and compared them to each other. We have not implemented SuffixNca, since others have shown that LcpRmq is better than SuffixNca in practice [1]. The algorithms we compared are the following:

DirectComp is the simple DirectComp algorithm with no preprocessing and worst case $O(n)$ query time.

FINGERPRINT$_k$<t_{k-1}, ..., t_1>**ac** is our new FINGERPRINT$_k$ algorithm using k levels, where k is 2, 3 and $\lceil \log n \rceil$. The numbers <t_{k-1}, ..., t_1> describe the exact size of fingerprinted substrings at each level.

FINGERPRINT$_k$<t_{k-1}, ..., t_1>**wc** is a variant of our new FINGERPRINT$_k$ algorithm, which starts at level $\ell = k - 1$ instead of $\ell = 0$, as discussed in Section 3.3.4.

FINGERPRINT$_k$<t_{k-1}, ..., t_1>**ac**<u_0, ..., u_{k-2}> is a variant of our new FINGERPRINT$_k$ algorithm, which stays at level ℓ until $v \geq u_\ell$, as discussed in Section 3.3.4, instead of doing one comparison at each level in step one of the query in Section 3.3.3.

RMQ<n, 1> is the LCPRMQ algorithm using constant time RMQ.

RMQ<p, q>$_{\textbf{virtual}}$ is the LCPRMQ algorithm using different RMQ algorithms. They all contain an additional virtual method call to make implementation simpler, but their results cannot be directly compared to other algorithms, which do not have this extra virtual method call.

DIRECTCOMP<c>RMQ<n, 1> is the DIRECTCOMP+LCPRMQ algorithm, where the cutoff from DIRECTCOMP to LCPRMQ is after c iterations.

FINGERPRINT$_k$<t_{k-1}, ..., t_1>**ac**$_{\textbf{cache}-dir-op}$ is a cache optimized variant of our FINGERPRINT$_k$ algorithm, where *dir* describes which of our two cache optimized variants is used, and *op* describes which of the shift or multiplication arithmetic operations it uses.

4.2 Test Inputs and Setup

We have tested the algorithms on different kinds of strings:

Average case strings These strings have many small LCE values, such that the average LCE value over all n^2 query pairs is less than one. We use results on these strings as an indication average case query times over all input pairs (i, j) in cases where most or all LCE values are small on expected input strings. We construct these strings by choosing each character uniformly at random from an alphabet of size 10

Worst case strings These strings have many large LCE values, such that the average LCE value over all n^2 query pairs is $n/2$. We use results on these strings as an indication of worst case query times, since the query times for all tested algorithms are asymptotically at their worst when the LCE value is large. We construct these strings with an alphabet size of one.

Medium LCE value strings These strings have an average LCE value over all n^2 query pairs of $n/2r$, where $r = 0.73n^{0.42}$. We use results on these strings as an indication of query times somewhere between the average case and worst case. We construct these strings by repeating a substring of r characters, where each character is unique.

For each kind of strings, we tested the algorithms using the following pattern:

1. Generate a string of length n and generate a million random pairs (i, j), where each of i and j is chosen between 1 and n uniformly at random.
2. For each tested algorithm, preprocess the given string, and run a query for each of the million pairs. Measure the time it takes to run all the queries combined.
3. Double the value of n and repeat from step one.

4.3 LcpRmq Results

We want to compare our new algorithms against the algorithm which is the best in practice of the asymptotically best $O(1)$ algorithms. The LcpRmq algorithm is better than SuffixNca [1], so we only look at LcpRmq. We have looked at a number of RMQ algorithms to find out which works best for the LCE problem. The results are shown in Figure 11. RMQ<n, 1> is the best in practice, and we therefore use that when we test against other LCE algorithms. This result is different from that of Ilie et al. [1], where RMQ<n, $4 \log n$> is the best. This could be due to hardware or compiler differences.

The performance of the implementations shown in Figure 11 cannot be compared with those shown on other figures, because they all contain an additional, slow virtual method call, which gives them an unfair disadvantage. We have optimized the best variant of LcpRmq to avoid the virtual method call, when we compare it against other algorithms. We compare this optimized version to DirectComp in Figure 12, which shows that DirectComp is significantly better in average case, though both have constant asymptotic average case query times.

The RMQ algorithms show a significant jump in query times around $n = 1.000.000$ on the plot with average case strings, but not on the plot with worst case strings. We have run the tests in Cachegrind, and found that the number of instructions executed and the number of data reads and writes are exactly the same for both average case strings and worst case strings. The cache miss rate for RMQ<n, 1> on average case strings is 14% and 9% for the L1 and L2 caches, and on worst case strings the miss rate is 17% and 13%, which is the opposite of what could explain the jump we see in the plot.

4.3.1 DirectComp+LcpRmq Results

Figure 12 also shows DirectComp+LcpRmq, where we first try DirectComp, and if we have not found the LCE value after one iteration, we use LcpRmq instead. We see that DirectComp+LcpRmq is worse than DirectComp on average case strings. We could increase the cutoff from one iteration to something bigger, which might improve average case performance. That would however increase worst case time. None of the two plots we measure represents a good worst case for DirectComp+LcpRmq compared to the other algorithms

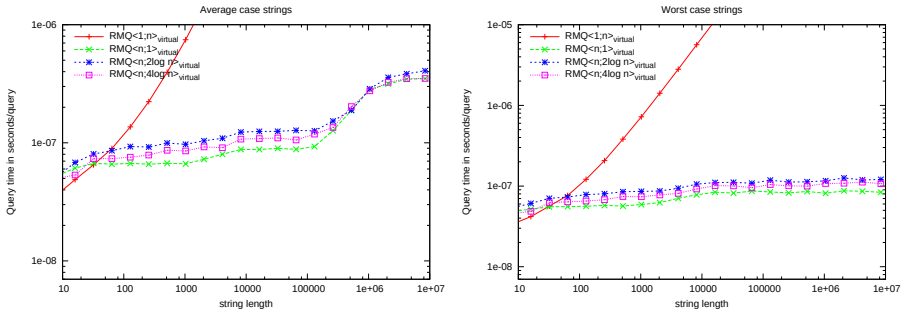


Figure 11: Query times of LcpRmq using different variants of RMQ.

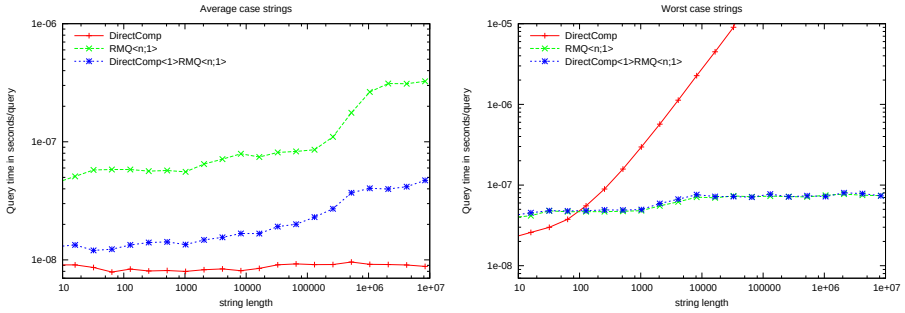


Figure 12: Query times of constant time LcpRmq, DirectComp and DirectComp+LcpRmq.

we test. The worst case for DirectComp+LcpRmq relative to the other algorithms would be when the average LCE value is slightly larger than the cutoff, such that DirectComp+LcpRmq reaches LcpRmq in most queries, while other algorithms still work on LCE values as small as possible. As described in Section 3.1.4, DirectComp+LcpRmq also uses a lot of space.

4.4 DirectCompLookup Results

Figure 13 shows DirectCompLookup with different time/space-tradeoffs together with LcpRmq. In average case, DirectCompLookup is slower than DirectComp, most likely because it needs to do an extra check to find out if it has reached a position for which a result is stored. For short strings, DirectLookup is faster than DirectCompLookup, since it does not need to check if the value it needs is stored. However, it quickly grows to be as slow as LcpRmq, while DirectCompLookup does not experience this slowdown. We explain this with that DirectLookup reads from its $O(n^2)$ table of results in every query, whereas DirectCompLookup only reads from the $O(n)$ string in

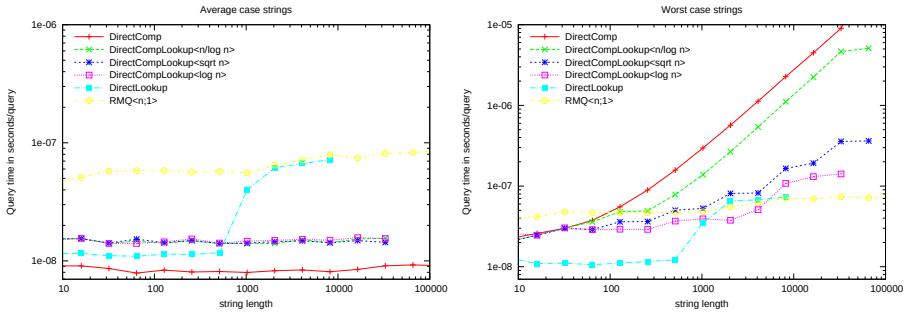


Figure 13: Query times of our new DirectCompLookup algorithm versus the existing DirectComp and LcpRmq algorithms.

File	n	σ	DC	FP_2	FP_3	$\text{FP}_{\log n}$	RMQ
book1	$0.7 \cdot 2^{20}$	82	8.1	11.4	10.6	12.0	218.0
kennedy.xls	$1.0 \cdot 2^{20}$	256	11.9	16.0	16.1	18.6	114.4
E.coli	$4.4 \cdot 2^{20}$	4	12.7	16.5	16.6	19.2	320.0
bible.txt	$3.9 \cdot 2^{20}$	63	8.5	11.3	10.5	12.6	284.0
world192.txt	$2.3 \cdot 2^{20}$	93	7.9	10.5	9.8	12.7	291.7

Table 3: Query times in nano seconds for DirectComp (DC), Fingerprint_k (FP_k) and LcpRmq (RMQ) on the five largest files from the Canterbury corpus.

average case, and thus has a significantly smaller working set.

In the worst case, the DirectCompLookup variants which use more space are faster in practice, just like in theory. If we are willing to use $O(n \log n)$ space, DirectCompLookup$<n/\log n>$ will only halve the worst case query time compared to DirectComp. This makes DirectCompLookup better than DirectComp, but still not a very good candidate when worst case query time is important.

DirectCompLookup uses $O(n^2)$ preprocessing time no matter what parameter t we choose, which is significantly slower than any of the other algorithms we have tested, and it is also the reason why we have only tested this algorithm up until $n = 100.000$. The only way we have found to improve this preprocessing time is to use one of the other algorithms we have covered to preprocess the data structure of DirectCompLookup.

4.5 Fingerprint_k Results

Figure 14 shows our experimental results on average case strings with a small average LCE value, worst case strings with a large average LCE value, and strings with a medium average LCE value.

On average case strings, our new Fingerprint_k algorithm is approximately

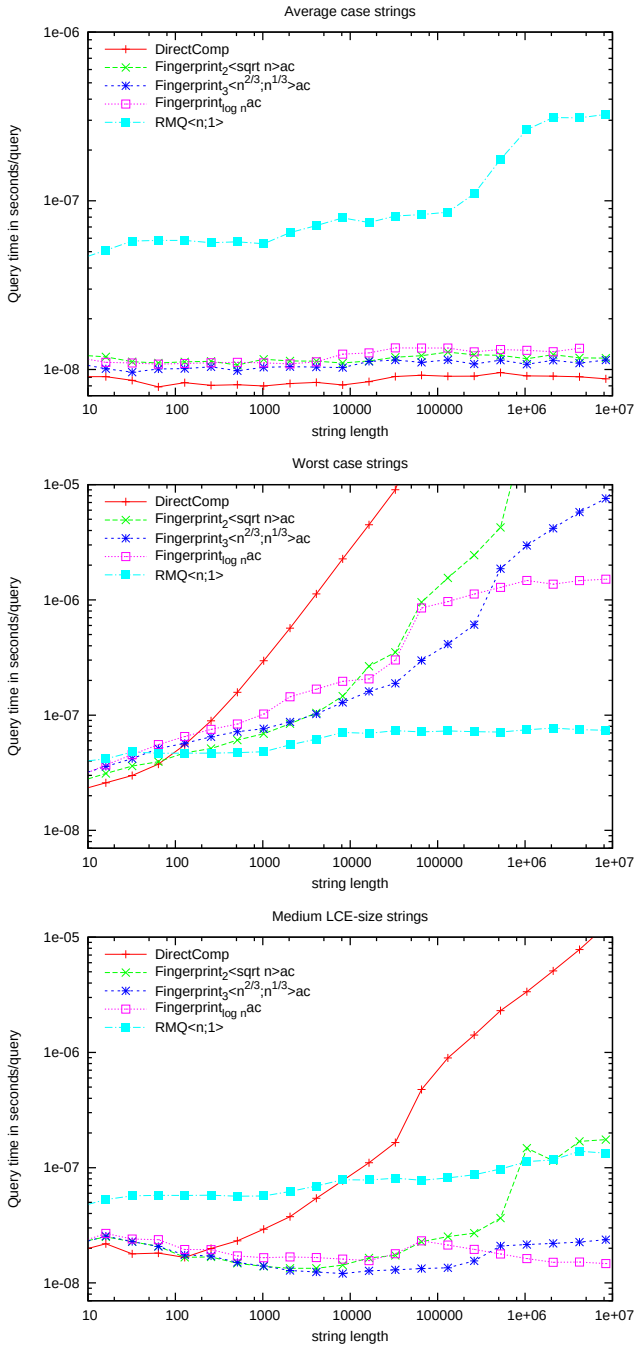


Figure 14: Comparison of our new FINGERPRINT$_k$ algorithm for $k = 2$, $k = 3$ and $k = \lceil \log n \rceil$ versus the existing DIRECTCOMP and LCPRMQ algorithms.

20% slower than DirectComp, and it is between than 5 and 25 times faster than LcpRmq. We see the same results on some real world strings in Table 3.

On worst case strings, the Fingerprint$_k$ algorithms are significantly better than DirectComp and somewhat worse than LcpRmq. Up until $n = 30.000$ the three measured Fingerprint$_k$ algorithms have nearly the same query times. Of the Fingerprint$_k$ algorithms, the $k = 2$ variant has a slight advantage for small strings of length less than around 2.000. For longer strings the $k = 3$ variant performs the best up to strings of length 250.000, at which point the $k = \lceil \log n \rceil$ variant becomes the best. This indicates that for shorter strings, using fewer levels is better, and when the input size increases, the Fingerprint$_k$ variants with better asymptotic query times have better worst case times in practice.

On strings with medium average LCE values, we see that our Fingerprint$_k$ algorithms are faster than both DirectComp and LcpRmq

We conclude that our new Fingerprint$_k$ algorithm achieves a tradeoff between worst case times and average case times, which is better than the existing best DirectComp and LcpRmq algorithms, yet it is not strictly better than the existing algorithms on all inputs. Fingerprint$_k$ is therefore a good choice in cases where both average case and worst case performance is important.

4.5.1 Optimal Value of t_ℓ

Figure 15 shows Fingerprint$_2$ for different values of t_1. On average case strings the value of t_1 makes no difference, and on worst case strings a value of t_1 between $2\sqrt{n}$ and $4\sqrt{n}$ is the best. The improvement relative to $\sqrt{n}$ is however insignificant, so we have chosen to use $t_1 = \sqrt{n}$.

Our results indicate that reading values from H_0 is faster than reading values from H_1. Differences between these two tables include that elements of $H_0 = s$ use one byte, whereas elements of H_1 use four bytes, and that elements we read from H_0 are located next to each other in memory, whereas elements we read from H_1 are spaced t_1 elements apart in memory.

4.5.2 Average Case Optimization

In Section 3.3.4 we discussed how our Fingerprint$_k$ algorithm is optimized for the average case to get $O(1)$ instead of $O(k)$ average case query time, while it keeps the same asymptotic worst case query time. But in practice the worst case query time is slightly affected.

In Figure 16 we see that on worst case strings, the query time of Fingerprint$_{\lceil \log n \rceil}$ doubles when we use the average case optimization. On average case strings, the query time of Fingerprint$_{\lceil \log n \rceil}$ is up to 40 times faster when using average case optimization. All four cases in Figure 16 show their expected $O(1)$ and $O(\log n)$ query times. In Figure 17, we see the same results for Fingerprint$_3$, as we saw for Fingerprint$_{\lceil \log n \rceil}$, but at a smaller scale. The negative effect of doing average case optimization is barely measurable on worst case strings, while the improvement on average case strings is up to 40%.

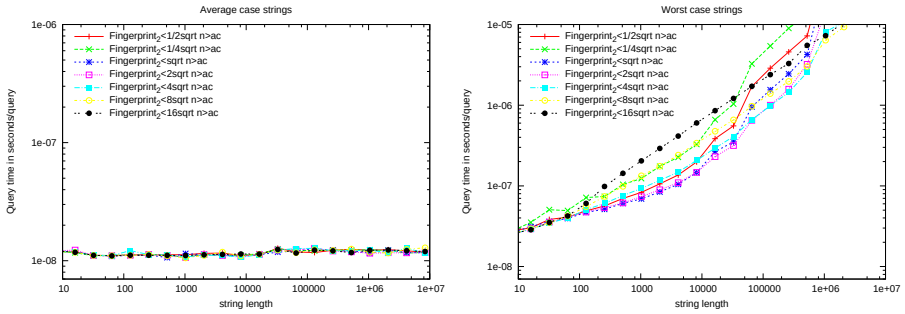


Figure 15: Query times of FINGERPRINT$_2$ for different values of t_1 versus query times for DIRECTCOMP.

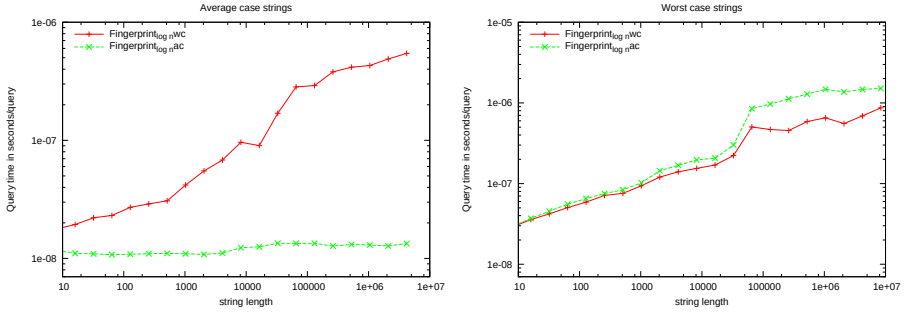


Figure 16: Query times of FINGERPRINT$_{\lceil \log n \rceil}$ with $O(1)$ versus $O(k)$ average case query time.

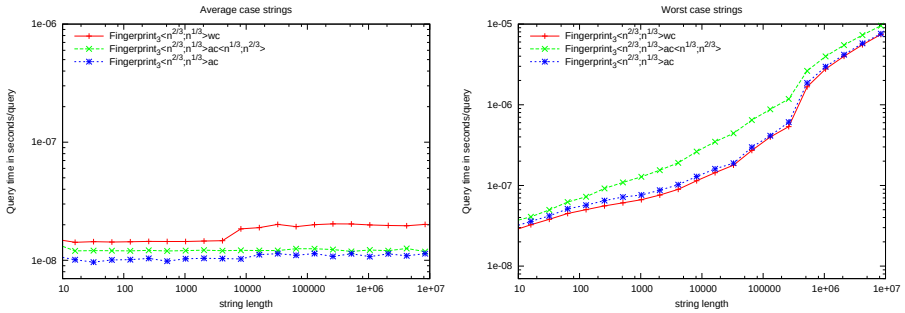


Figure 17: Query times of FINGERPRINT$_3$ with $O(1)$ versus $O(k)$ average case query time.

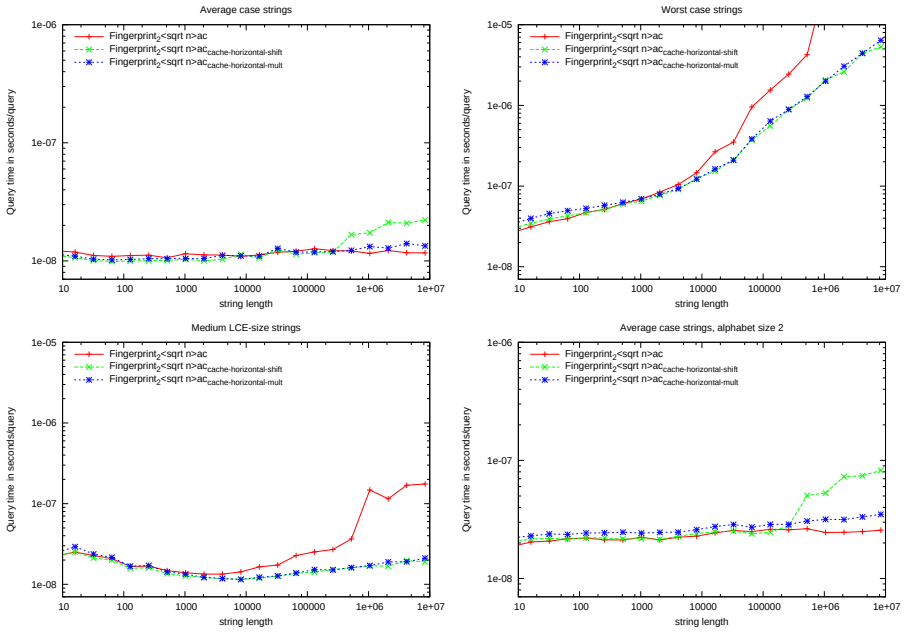


Figure 18: Query times of FINGERPRINT$_2$ with and without intra-level cache optimization.

In Figure 17 we see that doing one comparison at each level on the way up in FINGERPRINT$_3$ is better than doing $n^{1/k}$ comparisons. This is most visible on worst case strings, where the algorithm doing one comparison at each level uses less time to reach level $\ell = k - 1$, which it will reach anyway. On average case strings, the code specialized in handling only one comparison at each level is simpler.

We conclude that using average case optimization yields a query time improvement on average case strings, which is significantly greater than the query time increase on worst case strings, and that FINGERPRINT$_k$ therefore should use the average case optimization.

4.5.3 Cache Optimization

Horizontal, intra-level optimization Figure 18 shows results of the intra-level cache optimization described in Section 3.3.6. We have implemented two intra-level cache optimized variants. One as described in Section 3.3.6, and one where multiplication and division is replaced with shift operations. To use shift operations, t_ℓ and $\lceil n/t_\ell \rceil$ must both be powers of two. This may double the size of the used address space.

On average case strings the cache optimization does not change the query times, while on worst case strings and strings with medium size LCE values,

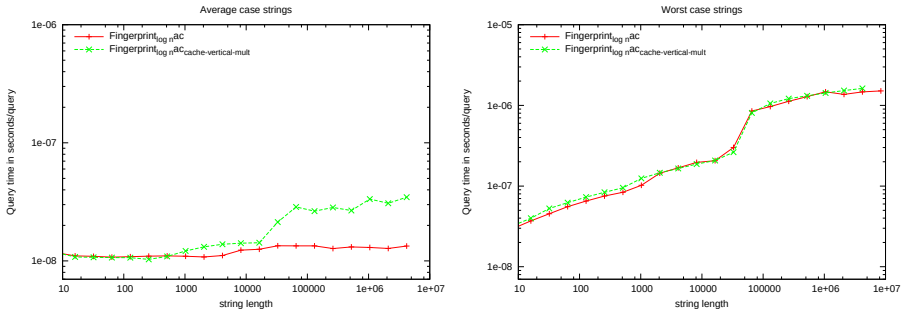


Figure 19: Query times of FINGERPRINT$_2$ with and without inter-level cache optimization.

cache optimization gives a noticeable improvement for large inputs. The cache optimized FINGERPRINT$_3$ variant with shift operations shows an increase in query times for large inputs, which we cannot explain.

The last plot on Figure 18 shows a variant of average case where the alphabet size is changed to two. This plot attempts to show the worst case for cache optimized FINGERPRINT$_3$. LCE values in this plot are large enough to ensure that H_1 is used often, which should make the extra complexity of calculating indexes into H_1 visible. At the same time the LCE values are small enough to ensure, that the cache optimization has no effect. In this plot we see that the cache optimized variant of FINGERPRINT$_3$ has only slightly worse query time compared to the variant, which is not cache optimized.

We conclude that this cache optimization does not visibly affect average case query times, while it improves worst case query times. However, due to late timing of this discovery, the cache optimized variant of FINGERPRINT$_k$ is not the main focus of our analysis.

Vertical, inter-level optimization Figure 19 shows results of our inter-level cache optimization. We see no improvement on worst case strings, and we see increased query times on average case strings. We conclude that this attempt at cache optimization does not improve practical query times, which could be expected, since it does not improve the asymptotic I/O bound either.

4.6 Conclusions on Experimental Results

Of the existing best DIRECTCOMP and LCPRMQ algorithms, we have seen that DIRECTCOMP is best on average case strings, while LCPRMQ is best on worst case strings.

We have found that our new DIRECTCOMPLOOKUP algorithm can improve the worst case query time performance of DIRECTCOMP, while it does not degrade the average case query time to the level of LCPRMQ. However it is not as

Variable	Example	Component
alphabet size	$\sigma = 10$	string
character pattern	uniformly at random	string
algorithm	FINGERPRINT$_k$	algorithm
alg. parameters	$t_1 = 2\sqrt{n}$	algorithm
string length	exponentially increasing from 1 to 100.000	test
query pattern	uniformly at random	test

Table 4: List of performance variables tested.

interesting as our new FINGERPRINT$_k$ algorithm, because it uses too much space.

Our new FINGERPRINT$_k$ algorithm is able to achieve a balance between worst case and average case query times. It has almost as good average case query times as DIRECTCOMP, its worst case query times are significantly better than those of DIRECTCOMP, and we have found cases between average and worst case where FINGERPRINT$_k$ is better than both DIRECTCOMP and LCPRMQ. FINGERPRINT$_k$ gives a good time space tradeoff, and it uses less space than LCPRMQ when k is small. The performance of FINGERPRINT$_k$ can be tweaked on many parameters, and we have found that optimizing for average case queries and for memory caches improve practical query time performance.

4.7 Test Code

To test practical performance, we have implemented the algorithms in C++. The implementation is based on that of Ilie et al. [1], which we have heavily modified. We have added a working build system, made it compile on a 32-bit Windows architecture, and generalized it to allow more algorithms and tests to be implemented. Table 4 summarizes the structure of the code and the parameters we can vary.

In our implementation, we have made the following assumptions about the LCE problem:

- A character in a string is always eight bits.
- A string always ends with a special null character, which does not occur anywhere else in the string. This does not affect our results, since we do not measure practical preprocessing time. If an algorithm has faster query times when the string ends in a special character, it could just add that character to the end of the string in its preprocessing.
- We make no assumptions on the order of i and j. In some applications of the LCE problem, for example approximate string searching [8], we know that $i < j$, and some algorithms might be able to take advantage of this knowledge to get faster query times. We do not measure this case.
- Whenever we use random characters or query pairs, we use the `rand()` function from C. This gives repeatable pseudo-random numbers, which

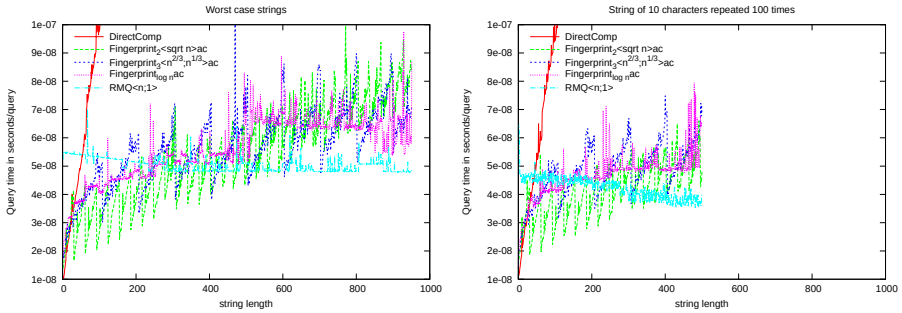


Figure 20: Query times of some of our tested algorithms as a function of the returned LCE value.

makes it easier for us to compare test runs.

4.7.1 Query time by LCE value

As an alternative way of testing our algorithms, we have looked at how long a query takes in relation to the LCE value it returns, since most of our algorithms have query times, which depend on the returned LCE value. Figure 20 shows query times for a fixed string of 1000 characters. We can see how the query times of FINGERPRINT$_2$ and FINGERPRINT$_3$ increases as the LCE value increases, and drops when the LCE value becomes long enough to make one more step at the higher level with fingerprints of longer substrings. We also see that the query time of FINGERPRINT$_{\log n}$ increases whenever the LCE value becomes long enough to reach a higher level of the data structure.

The plots in Figure 20 contains a lot of noise compared to our other plots. The number of measurements we make on these plots is many times bigger than the number of measurements we make on other plots. Therefore each measurement is an average over only 10000 queries instead of a million queries, which gives less accurate results.

We need to measure a sum of many queries to get an accurate timing, since each individual query is too fast to measure. While measuring a sequence of randomly generated queries is a very artificial test, which is unlikely to occur in practice, we think that it is even more artificial to measure a sequence of queries for which we already know that the LCE value is the same each time. On our worst case string $s = aaaaaaa...$, the queries we measure in each point in Figure 20 have the pattern $\max(i, j) = v$, which may give very different performance characteristics compared to random queries due to cache effects. We cannot meaningfully measure performance by LCE value on average case strings, as these strings do not contain large LCE values. The second plot in Figure 20 is on a string, which attempts to avoid a strong pattern like $\max(i, j) = v$ while it still has large LCE values.

Because of the amount of noise in the plots and the strong pattern of

$\max(i, j) = v$, we have not focused on this way of measuring query time performance in our analysis.

5 Ziv-Lempel Compression

Sometimes it can be useful to store a string in a compressed format and query the compressed string directly without using extra space and time to first decompress the string. In this section we explore the LCE problem on strings compressed using LZ78 compression. We look at solutions, which require space relative to the compressed size of a string, and we try to find solutions, which can answer LCE queries fast.

5.1 LZ78 definition

A string s of length N has a corresponding *compressed string* z of compressed length n, where z is a list of *compression elements*, $z = z_1, z_2, ..., z_n$. Each compression element z_x is a pair (r_x, α_x), where the reference, $\mathit{reference}_z(x) = r_x$, is an index of an earlier compression element in the compressed string or zero, i.e. $0 \leq \mathit{reference}_z(x) < x$, and the label, $\mathit{label}_z(x) = \alpha_z$, is a character from the original uncompressed string s. The *phrase* of a compression element is defined as:

$$\mathit{phrase}_z(x) = \begin{cases} \mathit{phrase}_z(\mathit{reference}_z(x)) \cdot \mathit{label}_z(x) & \text{if } \mathit{reference}_z(x) \neq 0 \\ \mathit{label}_z(x) & \text{otherwise} \end{cases}$$

where "$\cdot$" denotes string concatenation. From a compressed string $z = z_1, ..., z_n$, we can get the original string $s = \mathit{phrase}_z(1) \cdot ... \cdot \mathit{phrase}_z(n)$.

5.1.1 Compressing and Decompressing

We can easily get the decompressed string s from the compressed string z using the definitions. To get the compressed string z from the original string s we can look at the compression elements as a tree, where the reference of each compression element corresponds to the parent pointer of each node in the tree. We consume s from left to right while we build the tree:

1. Let the tree be a single special root node z_0, let $i = 1$ and $c = z_0$.
2. If c has a child u, where $\mathit{label}(u) = s[i]$, set $c = u$. Otherwise create a new node u as a child of c, set $\mathit{label}(u) = s[i]$, and set $c = z_0$.
3. Increment i by one and repeat from step two until the entire string is consumed when $i = N$.

The compressed string z is now the tree except the special root node z_0.

The size of the compressed string is at most $n = O(N)$ and at least $n = \Omega(\sqrt{N})$.

5.2 Compressed DirectComp

We can adapt DirectComp to work on a compressed string. To work directly on a compressed string, we first need to have random access to the characters of the original string.

Lemma 6. *Let s be a string of length N and let z of length n be the LZ-compression of s. For any $1 \leq i \leq N$, we can find $s[i]$ in $O(\log \log N)$ query time using $O(n)$ space.*

Proof. To find the character $s[i]$, we can first use a Predecessor query to find out which phrase $s[i]$ is in. We can then use a Level Ancestor query to find the correct character from the found phrase.

The query time for a random access query is $O(\log \log N)$, since we use $O(\log \log N)$ time on the Predecessor query and $O(1)$ time on the Level-Ancestor query. The space needed to support random access queries is $O(n)$, since both the predecessor data structure and the level ancestor data structure use $O(n)$ space. □

Theorem 3. *Let s be a string of length N and let z of length n be the LZ-compression of s. Compressed* DirectComp *can find $LCE(i, j)$ in $O(\log \log N + |LCE(i, j)|)$ query time using $O(n)$ space.*

Proof. Whenever DirectComp reads a character of the string s, we instead use the random access described in Lemma 6 to read the character from the compressed string. The predecessor query is only needed in the first comparison of the two indexes i and j in $LCE(i, j)$. In subsequent iterations of DirectComp we always need the same phrase or the phrase following the one we used used in the previous iteration of the loop, and we therefore do not need the predecessor query to find the phrase. The total query time for DirectComp on LZ-compressed strings becomes $O(\log \log N + |LCE(i, j)|)$, which is $O(N)$ in worst case and $O(\log \log N)$ on average. □

5.3 Reversed Compressed DirectComp

A simpler way of doing LCE on a compressed string is to reverse the string and then run DirectComp in the reverse direction on the reversed string, which gives the same result since reversing a string twice gives the original string. Where DirectComp compares $s[i + v]$ against $s[j + v]$ in each iteration of its loop, reversed DirectComp compares $s^R[N+1-i-v]$ against $s^R[N+1-j-v]$ in each iteration. This is simpler than using DirectComp directly on the compressed string, because Level Ancestor queries are no longer needed at each iteration, since we can use the reference of the current compression element as

follows, where the position of each of i and j is stored as a pair (x, y):

$$\textsc{PreviousPosition}(x,y) = \begin{cases} (\mathit{reference}(x), y) & \text{if } \mathit{reference}(x) \neq 0 \\ (y-1, y-1) & \text{if } y > 1 \\ \mathit{End\ of\ string} & \text{otherwise} \end{cases}$$

$$\textsc{Character}(x,y) = \mathit{label}(x)$$

$$\textsc{Position}(i) = (x,y) \text{ where}$$
$$y = \textsc{Predecessor}(i) \text{ and}$$
$$x = \textsc{Level-Ancestor}(i - \textsc{Start-Of}(y), y)$$

We then only need one Predecessor and one Level Ancestor query to initialize each of the two pairs.

5.3.1 Random Access Initialization

We can try to completely avoid Predecessor and Level Ancestor when initializing the pair (x, y). We could try to store all N pairs, but that would require $O(N)$ space and defeat the purpose of compression. We could also just walk from the end of the string to the desired position, but that would bring the average case query time up to $O(N)$. Similarly to how DirectCompLookup works, we can store some of these positions. If we store t positions equally spaced across the string s, we would get $O(n + t)$ space and $O(N/t + |LCE(i, j)|)$ query time. To keep the compressed space, we can set $t = n$, which gives $O(N/n + |LCE(i, j)|) = O(\sqrt{N} + |LCE(i, j)|)$ query time.

5.3.2 Length of the Reversed String

We can reverse the string in $O(N)$ time and $O(n_1 + n_2)$ space, where n_1 and n_2 are the compressed lengths of the original and reversed string respectively. To achieve this, we have already shown how we can read a string from right to left one character at a time in $O(1)$ space, and the compression algorithm described in Section 5.1.1 reads its input string one character at a time from left to right and uses $O(n)$ space for its tree of compression elements.

But the question is how big n_2 is relative to n_1. From knowing how much compression is possible with LZ-compression, we can see that $n_2 = O({n_1}^2)$. It might be possible to improve that bound, but for the string $z = (0, a)\ (1, b)\ (2, c)\ ...\ (n-1, z)$, we have experimentally determined that reversing z gives $n_2 = 0.89{n_1}^{1.51}$, so we cannot get an optimal bound of $n_2 = O(n_1)$. This is a disadvantage of doing LCE on the reversed string compared to doing it on the original compressed string using Level Ancestor queries.

5.4 Using Compression to Speed Up DirectComp

Our original goal of looking into compression was to see if using the techniques used in compression could improve the worst case query time of DirectComp

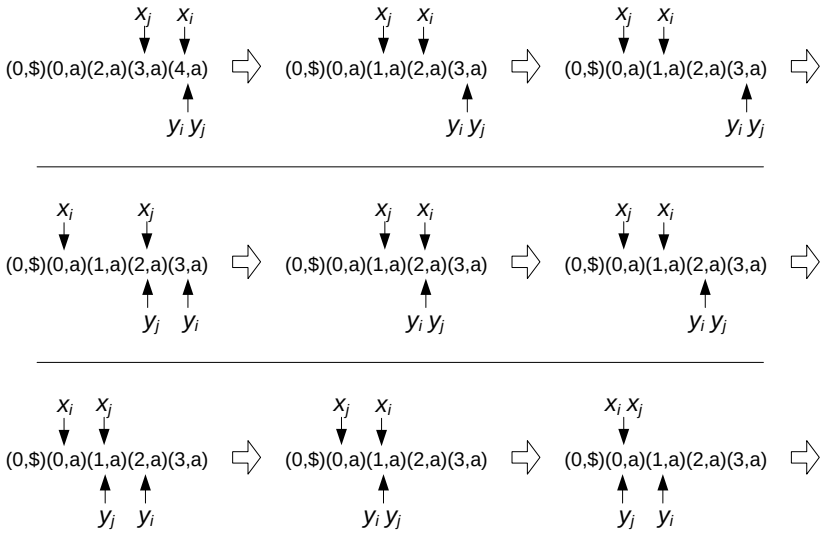


Figure 21: Reversed compressed DirectComp for $LCE(1,2)$ on the string $s = aaaaaaaaaa\$$. None of the steps can be skipped, since x_i and x_j are never the same.

to something better than $O(N)$. Our idea is that whenever you have a long LCE value, you have a repetition within the string, and repetitions would typically give good compression. If the compressed string can be searched over similarly to how DirectComp searches the uncompressed string, it might be possible to establish a better worst case query time.

We have found one case where we can make the reversed compressed DirectComp faster than DirectComp on uncompressed strings, but our finding does not improve worst case query time. Reversed compressed DirectComp starts with two positions (x_i, y_i) and (x_j, y_j) corresponding to the two positions i and j in $LCE(i,j)$. At each iteration, it then updates (x_i, y_i) and (x_j, y_j) to find the previous two positions. If $x_i = x_j$ we know that we can add $d(x_i)$ to the LCE value, where $d(x_i)$ is the depth of node x_i in the tree of compression elements. Reversed compressed DirectComp can then compare $d(x_i)$ characters in constant time.

Through this technique can improve query time in some cases, it does not improve the worst case time of $\Theta(N)$. This is because we cannot be sure to hit the condition allowing us to compare many characters in constant time, even for long LCE values. As an example, $LCE(1,2)$ on a string where all N characters are the same takes $\Omega(N)$ time. For such a string, the tree representing the compressed string is always a path, and x_i and x_j will always be either a node and its parent or the root and some other node. They will therefore never end up being the same node at the same time, as shown in Figure 21.

5.5 Compression with Other Algorithms

We can look at the other LCE algorithms we have examined in addition to DirectComp, to see if we can apply them to LZ-compressed strings. We have not found a way to apply SuffixNca or LcpRmq, as the string we want to find LCE values on still has N suffixes, and we have not found a way to relate the compression rate of the NCA structure or suffix array to the compression rate of the string.

Fingerprinting does not compress well either. We could LZ-compress each level of fingerprints just like the original string is compressed, but as the substrings fingerprinted becomes longer at each level, the compression ratio becomes worse, and at the top level with $t_\ell = \Omega(N)$, we need $\Omega(N)$ space to store H_ℓ, since all fingerprints would be unique or have very few copies.

Since we have not been able to find improvements to the $O(N)$ worst case query time in $O(n)$ space using existing algorithms, we have looked into new algorithms. The compression elements of a compressed string z forms a tree, and each phrase in z is a path in the tree. The LCE value is the length of a substring, which occurs at two positions i and j in the input string. This substring spans a number of phrases, where each phrase is a path in the tree of compression elements. If we can make fast LCE queries on such a tree, we can make fast LCE queries on highly compressable strings.

Theorem 4. *Let s be a string of length N and let z of length n be the LZ-compression of s. We can preprocess z using $O(n)$ space to answer LCE queries $v = LCE(i, j)$ in $O(\log\log N + \log n \cdot \#phrases)$ time, where $\#phrases$ is the number of phrases of z used to store $s[i\,..\,i+v-1]$ and $s[j\,..\,j+v-1]$.*

Proof. In Section 6 we describe how to perform LCE queries on a static tree of n nodes in $O(\log n)$ query time and $O(n)$ space. We can first find the positions of i and j within z in $O(\log\log N)$ time, and then we can repeat the $O(\log n)$ tree LCE query $\#phrases$ times to find the LCE value. □

5.6 Cache Performance

We have looked into the cache performance of compressed DirectComp. If we ignore the predecessor and level ancestor queries, and the string does not compress at all, i.e. the compressed string is $z = (0, a)(0, b)(0, c)...$, compressed DirectComp reads from the string in the best way possible using $O(N/B)$ I/O's in worst case. When the string has a better LZ-compression ratio, the I/O pattern becomes much worse. If $reference(x_1) = x_2 > 0$, the algorithm will read from position x_1 and then x_2, and these two may not be next to each other in memory. This can happen many times, which gives $O(N)$ I/O's in worst case.

We could lay out the tree of compression elements in memory as a B-tree, where we store groups of adjacent nodes next to each other in memory, to optimize the number of I/O's within each phrase, but each node may have more than B children, and in this case it would not improve the worst case number of I/O's, as each node visited may be in a new group.

Algorithm	Space	Query time	Preprocessing
TreeDirectComp	$O(n)$	$O(n)$	$O(n)$
TreeFingerprint$_k$, $1 \leq k \leq \lceil \log n \rceil$	$O(kn)$	$O(k \cdot n^{1/k})$	$O(n \log n)$
HeavyPathLce	$O(n)$	$O(\log n)$	$O(sort(n, \sigma))$
TreeLcpRmq	$O(n^2)$	$O(1)$	-
TreeSuffixNca	$O(n^2)$	$O(1)$	-
LadderLce	$O(n)$	$O(\log n)$	$O(sort(n, \sigma))$

Table 5: Algorithms for LCE on trees with their space requirements, query times and preprocessing times.

6 LCE on Trees

To achieve the result of Theorem 4, we need to do LCE on a tree in $O(\log n)$ query time and $O(n)$ space. In this section we analyze ways to achieve this. We first define LCE on trees, and then describe different solutions we have found to the problem. Table 5 lists the different solutions we have found.

6.1 Definitions

Trees Let T be a rooted tree with n nodes where each edge from parent node u to child node w is labeled with a character $label(w)$.

Paths A path $p = (u, w)$ in a tree is a path, which starts at node u and ends at node w, where u is an ancestor of w. We use the following notation on paths:

- The path $p = (u, w)$ has length $e = length(p) = d(w) - d(u)$, i.e. p has e edges and $e + 1$ nodes.
- The node $p[x]$ is a node in $p = (u, w)$, such that the distance between u and $p[x]$ is x. The start and end nodes of the path are $p[0] = u$ and $p[e] = w$. We can calculate $p[x]$ in constant time as $p[x] = LA(d(u) + x, w)$, where $d(u)$ is the depth of u, and LA is a Level Ancestor query.
- The subpath $p[x\,..\,y] = (p[x], p[y])$ of a path p is the path, which starts at node $p[x]$ and ends at node $p[y]$. The entire path is $p = p[0\,..\,e]$.
- The label of a path $label(p) = label(p[1]) \cdot ... \cdot label(p[e])$ is the concatenation of the labels on all edges between nodes in the path.

LCE on trees Given two paths p_i and p_j of lengths e_i and e_j in a tree T, the Longest Common Extension $v = LCE_T(p_i, p_j)$ is the length of the longest common prefix of $label(p_i)$ and $label(p_j)$.

In other words $label(p_i[0\,..\,v]) = label(p_j[0\,..\,v])$, and either $v = e_i$, $v = e_j$, or $label(p_i[v + 1]) \neq label(p_j[v + 1])$.

This definition of LCE on a tree is chosen to match the tree of LZ compression elements and the necessary LCE queries on it described in Section 5.5.

6.2 Walking the Paths with Level Ancestor

We can find $LCE(i,j)$ in $O(|LCE(i,j)|)$ query time using Level Ancestor, in the same way DIRECTCOMP finds LCE on strings: First compare $label(p_i[1])$ to $label(p_j[1])$, then compare $label(p_i[2])$ to $label(p_j[2])$ and so on until the two characters differ or either $p_i[e_i]$ or $p_j[e_j]$ is reached.

Theorem 5. TREEDIRECTCOMP *can preprocess a static tree of n nodes to answer LCE queries as defined in Section 6.1 using $O(|LCE(i,j)|)$ query time, and $O(n)$ space and preprocessing time.*

Proof. Each Level Ancestor query takes constant time, and we do $2|LCE(i,j)|+2$ of them, giving us $O(|LCE(i,j)|)$ query time. The space and preprocessing requirement of Level Ancestor is $O(n)$. □

6.3 Using Fingerprinting

We can adapt our FINGERPRINT$_k$ algorithm on strings to work on trees, which we call TREEFINGERPRINT$_k$. If we store the fingerprint of $s[i\,..\,i+t_\ell-1]$ in $H_\ell[i+t_\ell]$ instead of $H_\ell[i]$, then the FINGERPRINT$_k$ query directly adapts to TREEFINGERPRINT$_k$. If we wanted to use the same preprocessing as we use for FINGERPRINT$_k$ on strings, we would have to adapt the suffix array to apply to trees, and find a way to construct it in $sort(n,\sigma)$ time, which we have not looked at. Instead we can preprocess the fingerprints of the tree with Karp-Miller-Rosenberg [3].

Theorem 6. TREEFINGERPRINT$_k$*, where k is a parameter $1 \le k \le \lceil \log n \rceil$, can preprocess a static tree of n nodes to answer LCE queries as defined in Section 6.1 using $O(k \cdot n^{1/k})$ query time, $O(k \cdot n)$ space, and $O(n \log n)$ preprocessing time.*

6.3.1 Data Structure

We define a natural number $F_t[u]$ to be the fingerprint for node u of length t. For two nodes u and w in a tree T, their fingerprints $F_t[u]$ and $F_t[w]$ of the same length t are the same if and only if the last t characters on the path from the root of T to u is the same as the last t characters on the path from the root of T to w.

For a given parameter k, where $1 \le k \le \lceil \log n \rceil$, we define the TREEFINGERPRINT$_k$ data structure as follows: For each number ℓ between 0 and $k-1$, let $t_\ell = \Theta(n^{\ell/k})$, and let $t_0 = 1$. For each number ℓ between 0 and $k-1$ and each node u in T, whose depth is at least t_ℓ, let $H_\ell[u] = F_{t_\ell}[u]$

Lemma 7. *The* TREEFINGERPRINT$_k$ *data structure takes $O(k \cdot n)$ space.*

Proof. The data structure contains k numbers t_ℓ and k tables of fingerprints H_ℓ, and each table contains fingerprints for at most all of the n nodes in the tree. □

6.3.2 Query

The query for TreeFingerprint$_k$ is nearly the same as the query for Fingerprint$_k$ on strings, except we replace $H_\ell[i+v]$ and $H_\ell[j+v]$ with $H_\ell[p_i[v+t_\ell]]$ and $H_\ell[p_j[v+t_\ell]]$. We also decrement the level ℓ whenever we reach $v+t_\ell = e_i$ or $v+t_\ell = e_j$, since we no longer have a special character \$ to tell when we have reached the end.

In the TreeFingerprint$_k$ query, start with $v=0$ and $\ell=0$, then do the following steps:

1. As long as $v+t_\ell \le e_i \vee v+t_\ell \le e_j$ and $H_\ell[p_i[v+t_\ell]] = H_\ell[p_j[v+t_\ell]]$, increment v by t_ℓ, increment ℓ by one, and repeat this step unless $\ell = k-1$.
2. As long as $v+t_\ell \le e_i \vee v+t_\ell \le e_j$ and $H_\ell[p_i[v+t_\ell]] = H_\ell[p_j[v+t_\ell]]$, increment v by t_ℓ and repeat this step.
3. Stop and return v when $\ell = 0$, otherwise decrement ℓ by one and go to step two.

Lemma 8. *The* TreeFingerprint$_k$ *query finds the correct tree LCE value.*

Proof. The query algorithm keeps the invariant $\mathit{label}(p_i[0\,..\,v]) = \mathit{label}(p_j[0\,..\,v])$. At any given step at level ℓ, we check if $H_\ell[p_i[v+t_\ell]] = H_\ell[p_j[v+t_\ell]]$, which by the definition of our data structure is equivalent to $\mathit{label}(p_i[v\,..\,v+t_\ell]) = \mathit{label}(p_j[v\,..\,v+t_\ell])$. If the equality holds, we know that $\mathit{label}(p_i[0\,..\,v+t_\ell]) = \mathit{label}(p_j[0\,..\,v+t_\ell])$, and we maintain our invariant when we increment v by t_ℓ.

We stop when $v+1 > e_i$, $v+1 > e_j$, or $H_0[p_i[v+1]] \neq H_0[p_j[v+1]]$, at which point $v = \mathit{LCE}(p_i, p_j)$. □

Lemma 9. *The* TreeFingerprint$_k$ *query takes* $O(k \cdot n^{1/k})$ *time.*

Proof. We have changed nothing from the Fingerprint$_k$ query on strings, which negatively affects the query time, so the query time is the same. □

6.3.3 Preprocessing

We use Karp-Miller-Rosenberg [3] to preprocess the TreeFingerprint$_k$ data structure. When t_ℓ is $t_{\ell-1} < t_\ell \le 2t_{\ell-1}$, we preprocess level ℓ like this:

1. Make pairs of fingerprints $H_\ell[u] = (H_{\ell-1}[LA(d(u) - t_\ell + t_{\ell-1}, u)], H_{\ell-1}[u])$ for all nodes u in T, where $d(u) \ge t_\ell$.
2. Sort the pairs lexicographically.
3. Replace each pair with a new fingerprint, by going through them in lexicographical order, while updating the fingerprint whenever the pair changes.

Repeat the steps above for each level, starting with $\ell = 1$ and ending with $\ell = k - 1$. When $k < \lceil \log n \rceil$, the condition that $t_\ell \leq 2t_{\ell-1}$ is not satisfied. In this case we need to add extra levels in the preprocessing, which we will only store temporarily. We therefore always need to preprocess $O(\log n)$ levels, even if we only store some of them.

Lemma 10. *The KMR algorithm for preprocessing* TreeFingerprint$_k$ *gives the correct data structure.*

Proof. Two strings S and T of length a are the same if and only if we can split each of them into two smaller substrings $S = S_1 \cdot S_2$ and $T = T_1 \cdot T_2$ such that $S_1 = T_1$ and $S_2 = T_2$, where "$\cdot$" is string concatenation. Therefore we can assign fingerprints $F_a[S] = F_b[S_1] \cdot F_c[S_2]$ and $F_a[T] = F_b[T_1] \cdot F_c[T_2]$ in step one of the preprocessing algorithm. The same also holds when the two smaller substrings overlap.

The fingerprints from level $\ell - 1$ are natural numbers, less than n. Step one of the preprocessing algorithm makes each fingerprint at level ℓ a natural number less than n^2. Step two and three of the preprocessing algorithm reduces that universe of fingerprints back to n without changing the equality of the fingerprints. □

Lemma 11. *The KMR algorithm for preprocessing* TreeFingerprint$_k$ *takes* $O(n \log n)$ *time.*

Proof. Step one and three use linear time. In step two we can use radix sort, which uses linear time, because our universe of elements to sort is n^2. The total preprocessing time becomes $O(n \log n)$ for preprocessing $\log n$ levels in $O(n)$ time each. □

6.4 Using Constant Time String LCE in Heavy Paths

To answer LCE queries on a tree, we can create a heavy path decomposition of the tree and create a string of the labels of all the heavy paths concatenated. We can then use constant time string LCE on this string. We call this algorithm HeavyPathLce.

Theorem 7. HeavyPathLce *can preprocess a static tree of n nodes to answer LCE queries as defined in Section 6.1 using $O(\log n)$ query time, $O(n)$ space, and $O(sort(n, \sigma))$ preprocessing time.*

6.4.1 Data Structure

The HeavyPathLce data structure consists of the following:

- An input tree T with a heavy path decomposition.
- A string s, which is a concatenation of strings of all heavy paths of T, where the string of a heavy path is the label of the light edge leading to the heavy path concatenated with the label of the heavy path. Figure 22 shows an example of s.

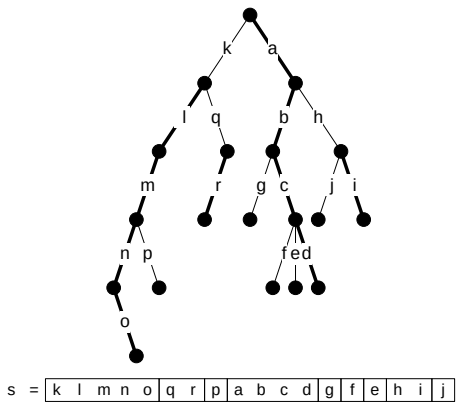


Figure 22: A heavy path decomposition of a tree and a corresponding HeavyPathLce string s.

- For a node u, $end(u)$ points to the node of greatest depth in u's heavy path, $d(u)$ is the depth of u, and $index(u)$ is the index in s, which stores $label(u)$.
- Data structures for constant time NCA on T, constant time LA on T, and constant time LCE on s.

Lemma 12. *The* HeavyPathLce *data structure takes $O(n)$ space.*

Proof. The string s contains $n-1$ characters, one for each edge in T. The space usage for NCA, LA, LCE, and the extra information about each node (*end*, *index*, and d) is all $O(n)$. □

Lemma 13. *We can preprocess the* HeavyPathLce *data structure in $O(sort(n,\sigma))$ time.*

Proof. Preprocessing LCE on s using SuffixNca or LcpRmq takes $O(sort(n,\sigma))$ time. If we walk through the tree and for each leaf append the heavy path string ending in that leaf to s, we can construct s in $O(n)$ time. The rest of the data structures can be preprocessed in $O(n)$ time. □

6.4.2 Query

In a HeavyPathLce query $LCE_T(p_i, p_j)$ we start with $v = 0$ and do the following repeatedly:

1. Stop when we reach either $v = e_i$ or $v = e_j$, and return v.
2. Find $m_i = length(p_i[v], NCA_T(end(p_i[v+1]), p_i[e_i]))$ and $i = index(p_i[v+1])$.
3. Find $m_j = length(p_j[v], NCA_T(end(p_j[v+1]), p_j[e_j]))$ and $j = index(p_j[v+1])$.

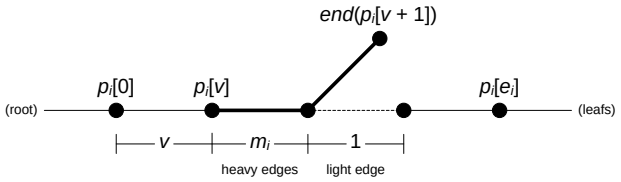


Figure 23: How we find m_i and m_j in a HeavyPathLce query.

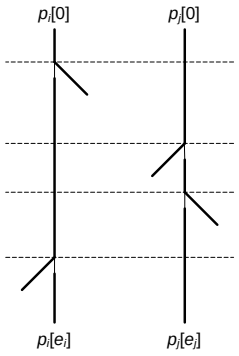


Figure 24: A HeavyPathLce query with at most five iterations. Heavy edges on p_i and p_j are marked bold.

4. Find $v' = \min\{LCE_s(i,j), m_i, m_j\}$.
5. Increment v by v'.
6. Stop if $v' < \min\{m_i, m_j\}$ and return v.

Lemma 14. *The* HeavyPathLce *query finds the correct LCE value.*

Proof. In step two, m_i is the number of edges on a single heavy path at the beginning of $p_i[v\,..\,e_i]$, like illustrated in Figure 23. The labels of these edges are stored consecutively in s, such that $\mathit{label}(p_i[v\,..\,v+m_i]) = s[i\,..\,i+m_i-1]$. The same holds for m_j in step three.

The algorithm keeps the invariant $\mathit{label}(p_i[0\,..\,v]) = \mathit{label}(p_j[0\,..\,v])$. At each iteration we find v', which is the longest common prefix of $\mathit{label}(p_i[v\,..\,v+m_i])$ and $\mathit{label}(p_j[v\,..\,v+m_j])$. This gives $\mathit{label}(p_i[0\,..\,v+v']) = \mathit{label}(p_j[0\,..\,v+v'])$, and we therefore keep the invariant when we add v' to v in step five.

If $v' < \min\{m_i, m_j\}$ in step six, we also know that $\mathit{label}(p_i[v+1]) \neq \mathit{label}(p_j[v+1])$, and therefore that $v = LCE(p_i, p_j)$.

□

Lemma 15. *The* HeavyPathLce *query takes* $O(\log n)$ *time.*

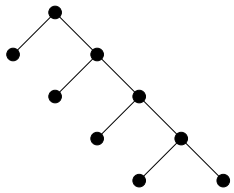


Figure 25: A tree with $\Omega(n^2)$ ways to choose a leaf and an ancestor of that leaf.

Proof. Since there is at most $O(\log n)$ light edges in any simple path in T, the two paths p_i and p_j contains at most $O(\log n)$ light edges in total. Since and each iteration of the query takes constant time, and we reach reach a new light edge of either p_i or p_j in each iteration, as shown in Figure 24, the query time is $O(\log n)$. □

6.5 Directly Mapping SuffixNca and LcpRmq

We have looked at applying SuffixNca and LcpRmq directly to trees like we did with DirectComp and Fingerprint$_k$, such that we can get $O(1)$ query time.

Theorem 8. TreeLcpRmq *and* TreeSuffixNca *can preprocess a static tree of n nodes to answer LCE queries as defined in Section 6.1 using $O(1)$ query time and $O(n^2)$ space.*

Proof. LcpRmq relies on accessing two suffixes in SA, i and j. In TreeLcpRmq we change the two suffixes to paths $p_i = (u_i, w_i)$ and $p_j = (u_j, w_j)$, and each of these can be chosen in $O(n^2)$ ways. We can restrict w_i and w_j to be leafs and still be able to answer the original LCE query in $O(1)$ time, but even with this modification, a tree like the one shown in Figure 25 still has $\Omega(n^2)$ ways to choose each path. The SA and LCP arrays in TreeLcpRmq would therefore be of length $O(n^2)$. When we look at TreeSuffixNca, the same argument holds for the necessary number of leafs in the suffix tree. □

Because of the quadratic space requirement, this result is not interesting when we use LCE on a tree to answer LCE queries on a compressed string in Section 5.5.

6.6 Looking at Level Ancestor

The constant time solution for the Level Ancestor problem by Bendera et al. [7] combines two $O(\log n)$ time solutions to get constant query times in linear space. We have looked at this algorithm to see if we can use similar techniques to achieve constant time LCE on trees in linear space. The two solutions used by the Level Ancestors algorithm are Jump Pointers and Ladder Decomposition. For the query $v = LA(\ell, u)$, first use jump pointers to jump from u half way to v

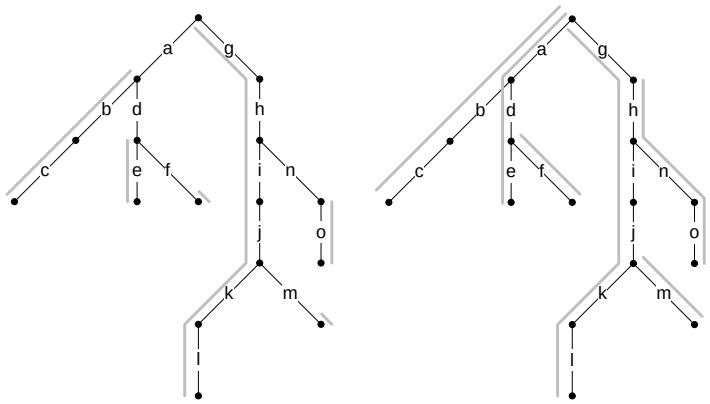


Figure 26: Ladders of a tree before and after their length is doubled. A LadderLce string s for this ladder decomposition is $s = abc|ade|df|ghijkl|ghno|jm$, where the splits between ladders are marked with a vertical line.

in constant time, and then use ladder decomposition to jump from the half way node to v.

LadderLce, which we describe in the next section, finds the LCE value of the first half of p_i and p_j in constant time, similar to how the ladder decomposition is used to jump from the half way node to v in the Level Ancestor algorithm. We have not found a constant time solution to find the LCE value of the second half of these paths, in the same way jump pointers is used to jump from node u to the half way node in the Level Ancestor algorithm. We have thus not found a constant time linear space solution for LCE on trees.

6.7 Using Constant Time String LCE in a Ladder Decomposition

We can use the ladder decomposition in the same way we use the heavy tree decomposition to get an algorithm for LCE on trees, which we call LadderLce, and which has the same properties as HeavyPathLce.

Theorem 9. LadderLce *can preprocess a static tree of n nodes to answer LCE queries as defined in Section 6.1 using $O(\log n)$ query time, $O(n)$ space, and $O(sort(n, \sigma))$ preprocessing time.*

6.7.1 Data Structure

The LadderLce data structure consists of the following:

- An input tree T with a ladder decomposition.

- A strings s, which is a concatenation of ladder labels of each ladder in T, where the label of a ladder is the concatenation of labels on each edge leading to a node in the ladder, ordered ascending by the depth of the nodes. Figure 26 shows an example of s.
- For a node u, $\mathit{index}(u)$ is an index in s within the ladder label of u's ladder, where $\mathit{label}(u)$ is stored.
- Data structures for constant time LA on T and constant time LCE on s.

Lemma 16. *The* LadderLce *data structure takes $O(n)$ space.*

Proof. The string s contains at most $2n$ characters. The rest of the data structures take $O(n)$ space. □

Lemma 17. *We can preprocess the* LadderLce *data structure in $O(\mathit{sort}(n, \sigma))$ time.*

Proof. Constructing s from the ladders takes $O(n)$ time, and preprocessing it for constant time LCE takes $O(\mathit{sort}(n, \sigma))$ time. The rest of the data structures have $O(n)$ preprocessing times. □

6.7.2 Query

The LadderLce query starts with $v = 0$ and repeats the following:

1. Stop when $v = e_i$ or $v = e_j$ and return v.
2. Find $m_i = \lceil \mathit{length}(p_i[v \,..\, e_i])/2 \rceil$ and $i = \mathit{index}(p_i[v + m_i]) - m_i + 1$.
3. Find $m_j = \lceil \mathit{length}(p_j[v \,..\, e_j])/2 \rceil$ and $j = \mathit{index}(p_j[v + m_j]) - m_j + 1$.
4. Find $v' = \min\{LCE_s(i, j), m_i, m_j\}$
5. Increment v by v'.
6. Stop if $v' < \min\{m_i, m_j\}$ and return v.

Lemma 18. *The* LadderLce *query finds the correct LCE value.*

Proof. In step two, $\mathit{length}(p_i[v \,..\, v + m_i]) + 1 \leq \mathit{length}(p_i[v + m_i \,..\, e_i]) \leq h(p_i[v + m_i])$, where h is the height of a node. For a node u of height h, the ladder decomposition guarantees that u's ancestor with distance $h + 1$ to u is in u's ladder. Therefore, $p_i[v + 1]$ is in $p_i[v + m_i]$'s ladder, and $s[i \,..\, i + m_i - 1] = \mathit{label}(p_i[v \,..\, v + m_i])$. Similarly in step three, $s[j \,..\, j + m_i - 1] = \mathit{label}(p_j[v \,..\, v + m_j])$.

By the same argument we use for HeavyPathLce, the algorithm maintains the invariant $\mathit{label}(p_i[0 \,..\, v]) = \mathit{label}(p_j[0 \,..\, v])$, and gives the correct LCE value. □

Lemma 19. *The* LadderLce *query takes $O(\log n)$ time.*

Proof. Each iteration takes constant time, and in each iteration we halve either $length(d_i[v\,..\,e_i])$ or $length(d_j[v\,..\,e_j])$. □

LadderLce is very similar to HeavyPathLce, however it has an extra property, which HeavyPathLce does not have: If $LCE(p_i, p_j)$ is less than half the length of p_i and less than half the length of p_j, the LCE query takes constant time, since it only needs to look at one ladder.

References

[1] Lucian Ilie, Gonzalo Navarro, and Liviu Tinta. The longest common extension problem revisited and applications to approximate string searching. Journal of Discrete Algorithms, Volume 8, Issue 4, December 2010, pages 418-428.

[2] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. J. ACM Vol. 47, No. 6, November 2000, pages 987-1011.

[3] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. 1972. Rapid identification of repeated patterns in strings, trees and arrays. In Proceedings of the fourth annual ACM symposium on Theory of computing (STOC '72). ACM, New York, NY, USA, 125-136.

[4] D. Harel, R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. SIAM J. Comput., 1984.

[5] Johannes Fischer, and Volker Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM'06), Lecture Notes in Computer Science 4009, 36-48, Springer-Verlag, 2006.

[6] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space Θ(N). Information Processing Letters, Volume 17, Issue 2, 24 August 1983, pages 81-84.

[7] Michael A. Bendera, Martín Farach-Colton. The Level Ancestor Problem simplified. Theoretical Computer Science 321, 2004, pages 5-12.

[8] Gad M. Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. 18th ACM STOC, pages 220–230, 1986.