

Algorithms for Longest Common Extensions

Philip Bille Inte Li Gørtz
phbi@imm.dtu.dk ilg@imm.dtu.dk

Jesper Kristensen
mail@jesperkristensen.dk

Technical University of Denmark

August 1, 2011

Abstract

The *longest common extension* (LCE) problem is to preprocess a string in order to allow for a large number of LCE queries, such that the queries are efficient. The LCE value, $LCE_s(i, j)$, is the length of the longest common prefix of the pair of suffixes starting at index i and j in the string s . The LCE problem can be solved in linear space with constant query time and a preprocessing of sorting complexity. There are two known approaches achieving these bounds, which use nearest common ancestors and range minimum queries respectively. However, in practice a much simpler approach with linear query time, no extra space and no preprocessing achieves significantly better average case performance. We show a new algorithm, FINGERPRINT_k which for a parameter k , $1 \leq k \leq \lceil \log n \rceil$, on a string of length n and alphabet size σ , gives $O(k \cdot n^{1/k})$ query time using $O(k \cdot n)$ space and $O(k \cdot n + \text{sort}(n, \sigma))$ preprocessing time. Though this solution is asymptotically strictly worse than the asymptotically best previously known algorithms, it outperforms them in practice in average case and is almost as fast as the simple linear time algorithm. On worst case input, this new algorithm is significantly faster in practice compared to the simple linear time algorithm. We also look at cache performance of the new algorithm, and we show that optimizing for cache only improves practical query time for $k = 2$.

1 Introduction

The *longest common extension* (LCE) problem is to preprocess a string in order to allow for a large number of LCE queries, such that the queries are efficient. The LCE value, $LCE_s(i, j)$, is the length of the longest common prefix of the pair of suffixes starting at index i and j in the string s . The LCE problem can be used in many algorithms for solving other algorithmic problems, e.g. the Landau-Vishkin algorithm for approximate string searching [6]. Solutions

with linear space and constant query time exists for the problem [4, 5]. These theoretically good solutions are however not the best in practice, since they have large constant factors for both time and space usage. A much simpler solution exists with average case constant time and no space required other than storing the input string. It has significantly better practical performance for average case input as well as for average case queries on some real world strings, when compared to the asymptotically best known algorithms [1]. However this algorithm has linear worst case query time, and is thus only ideal when worst case performance is irrelevant. In situations where we need both average case and worst case performance to be good, none of the existing solutions are ideal. Such a situation could be use of approximate string searching in a firewall. The firewall should not allow an attacker to significantly degrade its performance by sending it carefully crafted packages. At the same time it must scan legitimate data fast.

In this paper we give a new algorithm, which is a generalization of the simple linear time algorithm. The new algorithm is almost as simple, yet it has significantly better worst case query time. This algorithm also achieves significantly better average case practical query time compared to the asymptotically best known algorithms. The algorithm 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 $\log n$, on a string of length n .

1.1 Previous Results

Ilie et al. [1] gave an algorithm, DIRECTCOMP, for solving the LCE problem, which uses no preprocessing and $O(|LCE(i, j)|)$ query time. 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 . 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. 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], which gives DIRECTCOMP average case query time $O(1)$. The LCE problem can be solved 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, SUFFIXNCA, uses constant time nearest common ancestor (NCA) queries [4] 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 . I.e. $LCE_s(i, j) = D[NCA_T(L_i, L_j)]$. The other method, LCPRMQ, uses constant time range minimum queries (RMQ) [5] on a longest common prefix (LCP) array. The LCP array contains the length of the longest common prefixes of each pair of neighbor suffixes

in the suffix array (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 $SA^{-1}[i] < SA^{-1}[j]$. 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.

1.2 Our Results

We present a new LCE algorithm, FINGERPRINT_k, based on multiple levels of string fingerprinting. The algorithm has a parameter k in the range $1 \leq k \leq \lceil \log n \rceil$, which describes the number of levels used¹.

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 $O(\text{sort}(n, \sigma) + k \cdot n)$ preprocessing time.*

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

Corollary 1. FINGERPRINT₁ is equivalent to DIRECTCOMP with $O(n)$ space and $O(n)$ query time.

Corollary 2. FINGERPRINT₂ 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 $O(\text{sort}(n, \sigma) + k \cdot n)$ by using the SA and LCP arrays.

Table 1 shows an overview of asymptotic bounds of the different LCE algorithms.

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.

¹All logarithms are base two.

Algorithm	Space	Query time	Preprocessing
SUFFIXNCA	$O(n)$	$O(1)$	$O(\text{sort}(n, \sigma))$
LCPRMQ	$O(n)$	$O(1)$	$O(\text{sort}(n, \sigma))$
DIRECTCOMP	$O(1)$	$O(n)$	$O(1)$
FINGERPRINT _k *	$O(k \cdot n)$	$O(k \cdot n^{1/k})$	$O(\text{sort}(n, \sigma) + k \cdot n)$
$k = \lceil \log n \rceil$ *	$O(n \log n)$	$O(\log n)$	$O(n \log n)$

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. Rows marked with * show the new algorithm we present.

average case query time tradeoff between these two extremes. 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 $k = 6$.

For $k = 2$ we can improve practical FINGERPRINT_k query time even further by optimizing it for cache efficiency. However for $k > 2$ the query time performance degrades, as the added overhead outweighs the improved cache efficiency.

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.

1.3 Overview

In Section 3 we present the FINGERPRINT_k algorithm and analyze its theoretical complexity, and in Section 4 we present our findings from tests of a practical implementation of FINGERPRINT_k, which we compare against the existing algorithms.

2 Preliminaries

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 $\text{suffix}_i = s[i..n]$. The time it takes to sort n numbers within an universe of size σ is written as $\text{sort}(n, \sigma)$.

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 suffix_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 $O(\text{sort}(n, \sigma))$ time [2].

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 $\text{suffix}_{SA[1]}$, the lexicographically largest suffix is $\text{suffix}_{SA[n]}$, and the lexicographically i 'th smallest suffix is $\text{suffix}_{SA[i]}$. The *inverse suffix array* (SA^{-1}) describes where a given suffix is in the lexicographical order. Suffix suffix_i is the lexicographically $SA^{-1}[i]$ 'th smallest suffix.

The *longest common prefix array* (LCP array) describes the length of longest common prefixes of neighboring suffixes in SA. The length of the longest common prefix of $\text{suffix}_{SA[i-1]}$ and $\text{suffix}_{SA[i]}$ is $LCP[i]$, for $2 \leq i \leq n$. The first element $LCP[1]$ is always zero. Building the SA, SA^{-1} and LCP arrays takes $O(\text{sort}(n, \sigma))$ time [2].

The *nearest common ancestor* (NCA) of two nodes u and v is the node of greatest depth, which is an ancestor of both u and v . The ancestors of a node u includes u itself. 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].

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

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 .

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.

3.1 Data Structure

Given a string s , the fingerprint $F_t[i]$ is a natural number identifying the substring $s[i \dots i + t - 1]$ among all t -length substrings of s . We assign fingerprints such that for any i, j and t , $F_t[i] = F_t[j]$ if and only if $s[i \dots i + t - 1] = s[j \dots j + t - 1]$. 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.

$H_2[i+v]$	1	2	3	4	5	6	7	8	9	1	2	3	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$H_1[i+v]$	1	2	3	4	1	2	5	6	5	1	2	3	4	1	2	5	6	5	6	3	4	6	5	6	7	8	9
$H_0[i+v]$	a	b	b	a	a	b	b	a	b	a	b	b	a	a	b	b	a	b	a	b	a	a	b	a	b	a	\$
$i+v$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

$H_2[j+v]$	1	2	3	4	5	6	7	8	9	1	2	3	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$H_1[j+v]$	1	2	3	4	1	2	5	6	5	1	2	3	4	1	2	5	6	5	6	3	4	6	5	6	7	8	9
$H_0[j+v]$	a	b	b	a	a	b	b	a	b	a	b	b	a	a	b	b	a	b	a	b	a	a	b	a	b	a	\$
$j+v$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

Figure 2: FINGERPRINT_k query for LCE(3, 12) on the data structure of Figure 1. The top half shows how $H_\ell[i+v]$ moves through the data structure, and the bottom half shows $H_\ell[j+v]$.

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 $v = O(1)$. \square

Average Case Optimization We could have left out step one of the query algorithm and started with $\ell = k - 1$. 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 $O(k)$. Our experiments have shown that whenever k is small, keeping step one improves average case query time, while it does not have a measurable effect on worst case query times. When $k = \lceil \log n \rceil$, keeping step one can double the worst case query time, while it can make the average case query time 40 times faster for an input string of ten million characters. We want to optimize our LCE query time for the average case where the LCE value is small, so our results in Section 4 does not include the variant of FINGERPRINT_k with $O(k)$ average case query time.

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. Our experiments have shown that always doing one comparison at each level is the best in practice.

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 3: The first column lists all substrings of $s = \text{abbababba}$ 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 .

3.3 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 $O(\text{sort}(n, \sigma) + k \cdot n)$ 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 $s[\text{SA}[i] \dots \text{SA}[i] + t_\ell - 1]$ 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 $\text{LCE}[i] \geq t_\ell$.

An example is shown in Figure 3.

Lemma 4. *The preprocessing algorithm described above generates the data structure described in Section 3.1.*

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.

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. \square

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

Proof. We first construct the SA and LCP arrays, which takes $O(\text{sort}(n, \sigma))$ 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 $O(\text{sort}(n, \sigma) + k \cdot n)$. \square

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 the FINGERPRINT_k algorithm in C++ and compared them with optimized versions of the DIRECTCOMP and LCPRMQ algorithms. The algorithms we compared are the following:

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

$\text{FINGERPRINT}_k \langle t_{k-1}, \dots, t_1 \rangle \mathbf{ac}$ is the FINGERPRINT_k algorithm using k levels, where k is 2, 3 and $\lceil \log n \rceil$. The numbers $\langle t_{k-1}, \dots, t_1 \rangle$ describe the exact size of fingerprinted substrings at each level.

$\text{RMQ} \langle n, 1 \rangle$ is the LCPRMQ algorithm using constant time RMQ .

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.

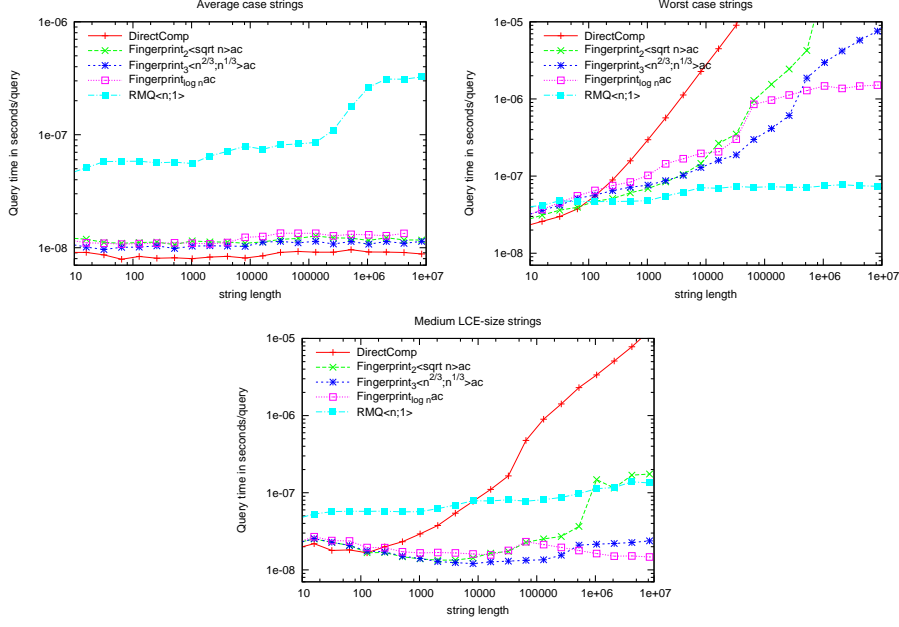


Figure 4: Comparison of our new FINGERPRINT_k algorithm for $k = 2$, $k = 3$ and $k = \lceil \log n \rceil$ versus the existing DIRECTCOMP and LCPRMQ algorithms.

Medium LCE value strings These strings have an average LCE value over all n^2 query pairs of $n/2r$, where $r = 0.73n^{0.42}$. This value of r is a power regression over $(r, n) = (5, 100)$, $(15, 1,000)$, $(35, 10,000)$ and $(100, 100,000)$, where the each of the values of n were chosen to give the best result for FINGERPRINT_k on a plot with the given constant value of r . 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.

Each measurement we make is an average of query times over a million random query pairs (i, j) . For a given string length and string type (worst case strings vs. average case strings), we use the same string and the same million query pairs on all tested algorithms.

4.3 Results

Figure 4 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 20% slower than DIRECTCOMP, and it is between than 5 and 25 times faster

File	n	σ	DC	FP ₂	FP ₃	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 2: Query times in nano seconds for DIRECTCOMP (DC), FINGERPRINT_k (FP_k) and LCPRMQ (RMQ) on the five largest files from the Canterbury corpus.

than LCPRMQ. We see the same results on some real world strings in Table 2.

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.

LCPRMQ shows 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 average case strings is 14% and 9% for the L1 and L2 caches, and for 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.

4.4 Cache Optimization

The amount of I/O 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 operations to $O(k(n^{1/k}/B + 1))$ in the cache-oblivious model. Instead of storing $F_{t_\ell}[i]$ at $H_\ell[i]$, we can store it at $H_\ell[(i-1) \bmod t_\ell] \cdot \lceil n/t_\ell \rceil + \lfloor (i-1)/t_\ell \rfloor + 1$. This will group all used fingerprints at level ℓ next to each other in memory, such that the amount of I/O at each level is reduced from $O(n^{1/k})$ to $O(n^{1/k}/B)$.

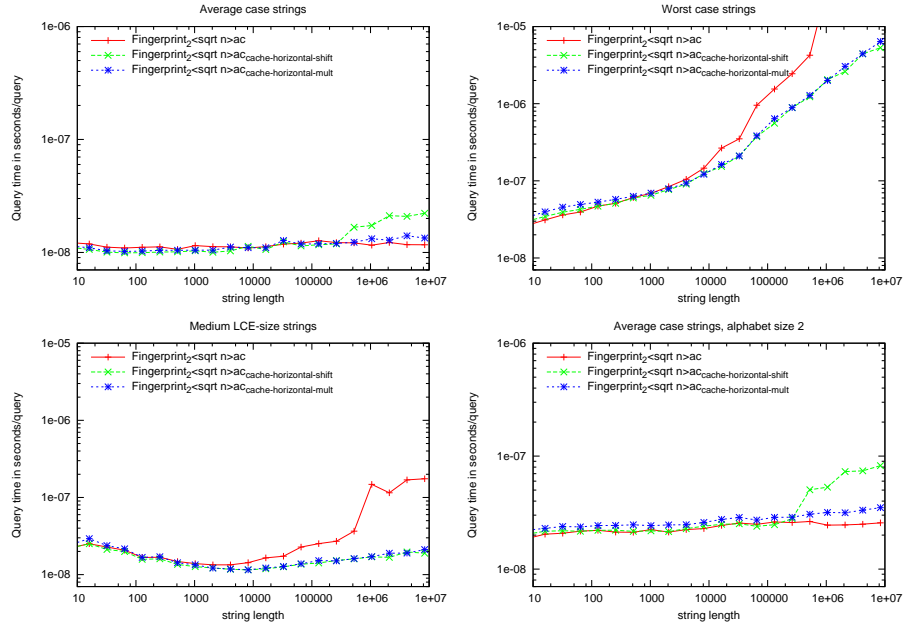


Figure 5: Query times of FINGERPRINT₂ with and without cache optimization.

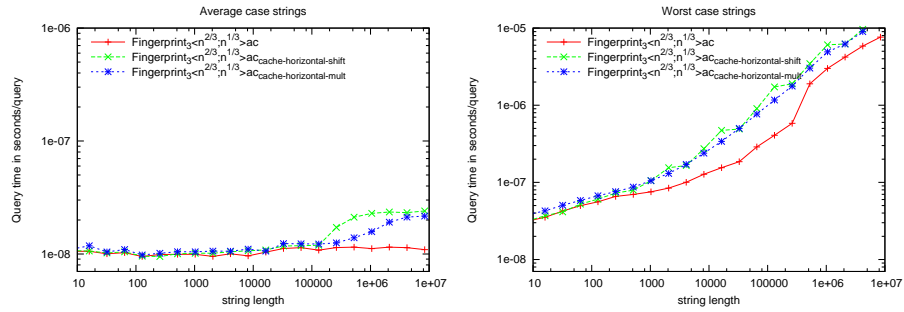


Figure 6: Query times of FINGERPRINT₃ with and without cache optimization.

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.

We have implemented two cache optimized variants. One as described above, and one where multiplication, division and modulo 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.

Figure 5 shows our measurements for FINGERPRINT₂. 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, cache optimization gives a noticeable improvement for large inputs. The cache optimized FINGERPRINT₂ variant with shift operations shows an increase in query times for large inputs, which we cannot explain.

The last plot on Figure 5 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 FINGERPRINT₂. 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 FINGERPRINT₂ has only slightly worse query time compared to the variant, which is not cache optimized.

Figure 6 shows our measurements for FINGERPRINT₃. Unlike for FINGERPRINT₂, we see a degraded performance on worst case strings compared to the not cache optimized variant. Our results thus show that cache optimization of FINGERPRINT_k only gives a performance improvement for $k = 2$.

5 Conclusions

We have presented the FINGERPRINT_k algorithm, where k is a parameter $1 \leq k \leq \lceil \log n \rceil$, which for a string s of length n and alphabet size σ , 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 $O(\text{sort}(n, \sigma) + k \cdot n)$ preprocessing time.

The FINGERPRINT_k algorithm is able to achieve a balance between practical 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.

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] 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.