Algorithms for Longest Common Extensions

Jesper Kristensen
DTU Informatics
Technical University of Denmark

August 1, 2011

1 Introduction

The *longest common extension* (LCE) problem is to preprocess a string in order to allow for a large number of LCE querie, 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 with constant query time exists for the problem [TODO cite who]. These theoretically good 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 [1]. 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. Such a situation could be use of approximate string searching in a firewall, which should not allow an attacker to significantly degrade its performance by sending it carefully crafted packages, while the average case of scanning legitimate data must still be highly performant.

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

1.1 Previous Results

Ilie et al. [1] gave and 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 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. $LCE_s(i,j) = D[NCA_{\mathcal{T}}(L_i, L_j)]$. The other method, LcpRmq, uses constant time range minimum queries (RMQ) 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$_k$, where k is a parameter $1 \leq k \leq \lceil \log n \rceil$, which describes the number of levels used[1]. This algorithm is based on string fingerprinting.

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.

[1] All logarithms are base two.

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

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

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 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. 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 $k = 6$.

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

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 $\mathit{suff}_i = s[i\,..\,n]$.

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].

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 array) describes the length of longest common prefixes of neighboring suffixes in SA. The length of the longest common prefix of $\mathit{suff}_{SA[i-1]}$ and $\mathit{suff}_{SA[i]}$ 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 $O(\mathit{sort}(n, \sigma))$ time [2].

NCA. The ancestors of a node u is u itself as well as any node, which is a parent of an ancestor of u. 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. 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* (RMQ) on a static array of n elements can be answered in $O(1)$ time with $O(n)$ space and preprocessing time [5].

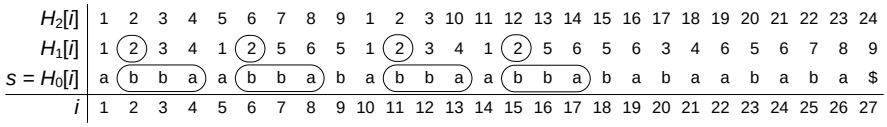

Figure 1: FINGERPRINT$_k$ data structure for $s = abbaabbababbaabbababaababa\$$, $n = 27$, $k = 3$, $t_1 = n^{1/3} = 3$ and $t_2 = n^{2/3} = 9$. All substrings bba are highlighted with their 3-length fingerprint 2.

I/O 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.

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 $s[i\,..\,i+t-1]$ 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 $s[i\,..\,i+t-1] = s[j\,..\,j+t-1]$. 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.

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 1.

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. □

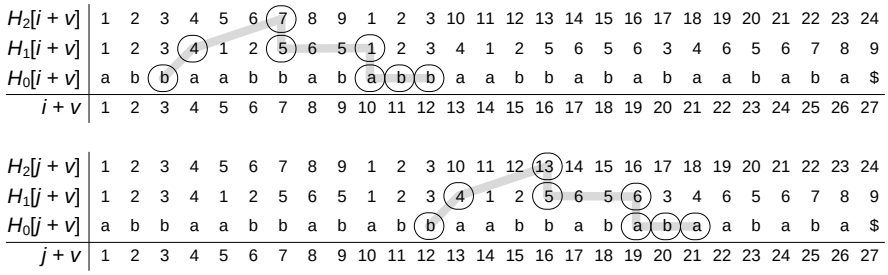

Figure 2: FINGERPRINT$_k$ query for $LCE(3, 12)$ on the data structure of Figure 1. The top half shows how $H_\ell[i + v]$ moves through the data structure, and the bottom half shows $H_\ell[j + v]$.

3.2 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 2.

Lemma 2. *The* FINGERPRINT$_k$ *query algorithm is correct.*

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 $v = O(1)$. □

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.

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 $O(sort(n, \sigma) + k \cdot n)$ 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 $s[SA[i] \mathinner{..} SA[i] + t_\ell - 1]$ 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 3.

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 $s = abbababba$ 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.

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. □

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 $O(sort(n, \sigma) + k \cdot n)$. □

3.4 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 the fingerprint of $s[i \,..\, i + t_\ell - 1]$ 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 $O(n^{1/k}/B)$.

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.

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.

FINGERPRINT$_k$**<**t_{k-1}**, ...,** t_1**>ac** is the 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.

RMQ**<**n**,** 1**>** 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.

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.

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.

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 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 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

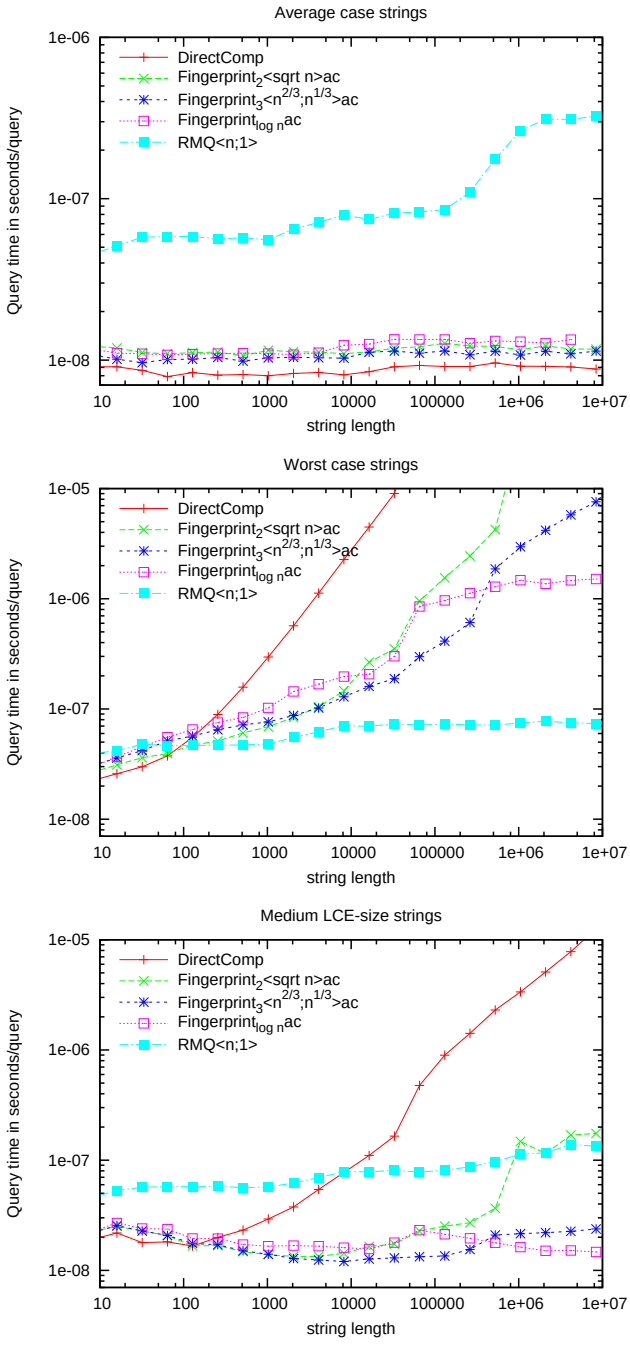

Figure 4: Comparison of our new FINGERPRINT$_k$ algorithm for $k = 2$, $k = 3$ and $k = \lceil \log n \rceil$ versus the existing DIRECTCOMP and LCPRMQ algorithms.

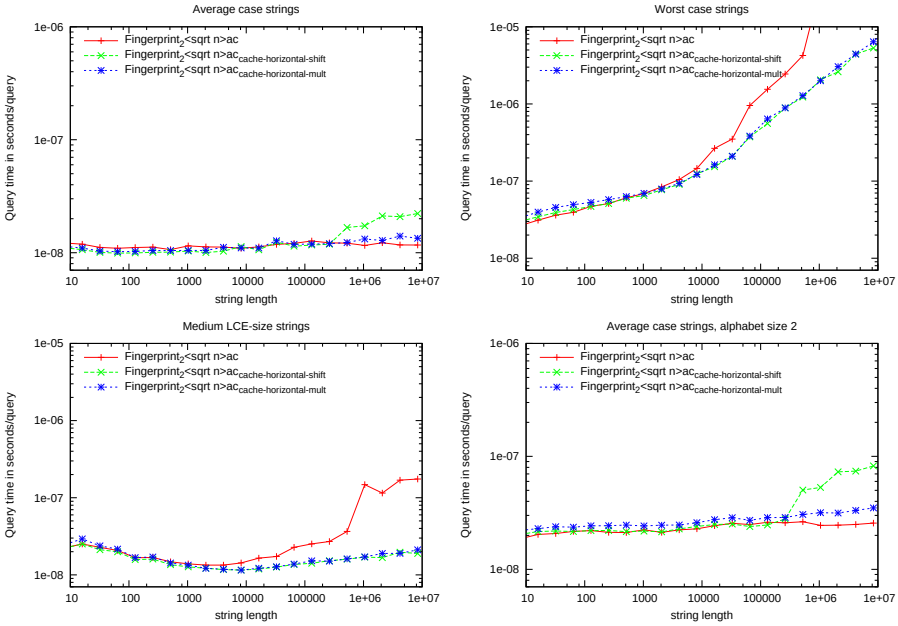

Figure 5: Query times of FINGERPRINT$_2$ with and without intra-level cache optimization.

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

Figure 5 shows results of the cache optimization described in Section 3.4. We have implemented two cache optimized variants. One as described in Section 3.4, 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, 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 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 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.

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 $O(sort(n, \sigma) + k \cdot n)$ 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.