Longest Common Extensions via Fingerprinting

Philip Bille, Inge Li Gørtz, and Jesper Kristensen

Technical University of Denmark, DTU Informatics, Copenhagen, Denmark

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 $O(kn^{1/k})$ query time using $O(kn)$ space and $O(kn + sort(n, \sigma))$ preprocessing time, where $sort(n, \sigma)$ is the time it takes to sort n numbers from σ. 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 for $k = 2$, cache optimization can improve practical query time.

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, constant query time, and $O(sort(n, \sigma))$ preprocessing time exist for the problem [3, 2]. Here $sort(n, \sigma)$ is the time it takes to sort n numbers from an alphabet of size σ. For $\sigma = O(n^c)$, where c is a constant, we have $sort(n, \sigma) = O(n)$. These theoretically good solutions are however not the best in practice, since they have large constant factors for both query time and space usage. Ilie et al. [4] introduced a much simpler solution with average case constant time and no space or preprocessing required other than storing the input string. This solution 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. 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. An example could be a firewall, which needs to do approximate string searching. 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 quickly. The main goal of this paper is to design a practical algorithm that performs well in both situations, that is, achieves a good worst-case guarantee while maintaining a fast average case performance.

Previous Results Throughout the paper let s be a string of length n over an alphabet of size σ. Ilie et al. [4] gave an algorithm, DirectComp, for solving the LCE problem, which uses no preprocessing and has $O(LCE(i,j))$ query time. For a query $LCE(i,j)$, the algorithm simply 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 thus $O(n)$. However, on random strings and many real-word texts, Ilie et al. [4] showed that the average LCE is $O(1)$, where the average is over all $\sigma^n n^2$ combinations of strings and query inputs. Hence, in these scenarios DirectComp achieves $O(1)$ query time.

The LCE problem can also be solved with $O(1)$ worst case query time, using $O(n)$ space and $O(sort(n,\sigma))$ preprocessing time. Essentially, two different ways of doing this exists. One method, SuffixNca, uses constant time nearest common ancestor (NCA) queries [3] on a suffix tree. The LCE of two indexes i and j is defined as the length of the longest common prefix of the suffixes $s[i\,..\,n]$ and $s[j\,..\,n]$. In a suffix tree, the path from the root to L_i has label $s[i\,..\,n]$ (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) [2] 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 $SA^{-1}[i] < SA^{-1}[j]$. Table 1 summarizes the above theoretical bounds.

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 used[1]. The performance of the algorithm is summarized by the following theorem:

[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	$O(1)$	$O(n)$	None
Fingerprint$_k$*	$O(kn)$	$O(kn^{1/k})$	$O(sort(n, \sigma) + kn)$
$k = \lceil \log n \rceil$ *	$O(n \log n)$	$O(\log n)$	$O(n \log n)$

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.

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 $O(kn^{1/k})$ worst case query time and $O(1)$ average case query time using $O(kn)$ space and $O(sort(n, \sigma) + kn)$ preprocessing time.*

By choosing k we can obtain the following interesting tradeoffs.

Corollary 2. Fingerprint$_1$ *is equivalent to* DirectComp *with $O(n)$ space and $O(n)$ query time.*

Corollary 3. Fingerprint$_2$ *uses $O(n)$ space and $O(\sqrt{n})$ query time.*

Corollary 4. Fingerprint$_{\lceil \log n \rceil}$ *uses $O(n \log n)$ space and $O(\log n)$ query time.*

The latter result is equivalent to the classic Karp-Miller-Rosenberg fingerprint scheme [5]. To preprocess the $O(kn)$ fingerprints used by our algorithm, we can use Karp-Miller-Rosenberg [5], which takes $O(n \log n)$ time. For $k = o(\log n)$, we can speed up preprocessing to $O(sort(n, \sigma) + kn)$ by using the SA and LCP arrays.

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 $k = 6$.

For $k = 2$ we can improve practical Fingerprint$_k$ query time even further by optimizing it for cache efficiency. However for $k > 2$, this cache optimization degrades practical query time performance, 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.

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

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 [1].

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 [1].

The *nearest common ancestor* (NCA) of two nodes u and v in a tree 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 [3].

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 [2].

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.

$H_2[i]$	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
$H_1[i]$	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
$s = H_0[i]$	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	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

Fig. 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.

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 last character of the string must be a special character \$, which does not occur anywhere else in the string.

The FINGERPRINT$_k$ data structure for a string s of length n, where k is a parameter $1 \leq k \leq \lceil \log n \rceil$, 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 $0 \leq \ell \leq k-1$, $t_\ell = \Theta(n^{\ell/k})$ 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. An example is shown in Fig. 1. Since each of the k tables stores n fingerprints of constant size, we get the following.

Lemma 5. *The* FINGERPRINT$_k$ *data structure takes* $O(kn)$ *space.*

3.2 Query

The FINGERPRINT$_k$ query speeds up DIRECTCOMP by comparing fingerprints of substrings of the input string instead of individual characters. The query algorithm consists of two traversals of the hierarchy of fingerprints. In the first traversal the algorithm compares progressively larger fingerprints of substrings until a mismatch is found and in the second traversal the algorithm compares progressively smaller substrings to find the precise point of the mismatch. The combination of these two traversals ensures both a fast worst case and average case performance.

The details of the query algorithm are as follows. Given 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 $\ell = k-1$.
2. As long as $H_\ell[i+v] = H_\ell[j+v]$, increment v by t_ℓ and repeat this step.

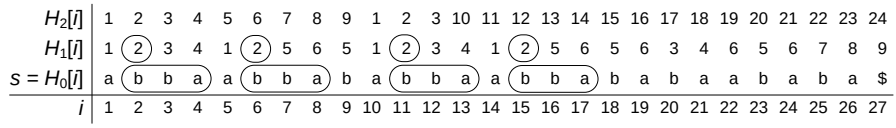

Fig. 2. FINGERPRINT$_k$ query for $LCE(3, 12)$ on the data structure of Fig. 1. The top half shows how $H_\ell[i+v]$ moves through the data structure, and the bottom half shows $H_\ell[j+v]$.

3. 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 Fig. 2.

Lemma 6. *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 7. *The worst case query time for* FINGERPRINT$_k$ *is* $O(kn^{1/k})$*, and the average case query time is* $O(1)$*.*

Proof. First we consider the worst case. 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, $O(kn^{1/k})$ query time is used.

Next we consider the average case. 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 in the number of times v is incremented, and it is thereby $O(v)$. From the proof of Lemma 6 we have $v = LCE(i, j)$. By Ilie et al. [4] the average $LCE(i, j)$ is $O(1)$ and hence the average case query time is $O(1)$. □

Average Case vs. Worst Case The first traversal in the query algorithm guarantees $O(1)$ average case performance. Without it, i.e., if the query started with

Subst.	$H_l[i]$	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

Fig. 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.

$\ell = k - 1$ and omitted step one, the average case query time would be $O(k)$. However, the worst case bound would remain $O(kn^{1/k})$. Thus, for a better practical worst-case performance we could omit the first traversal entirely. We have extensively experimented with both variants and we found that in nearly all scenarios the first traversal improved the overall performance. In the cases where performance was not improved the first traversal only degraded the performance slightly. We therefore focus exclusively on the two traversal variant in the remainder of the paper.

3.3 Preprocessing

The tables of fingerprints use $O(kn)$ space. In the case with $k = \lceil \log n \rceil$ levels, the data structure is the one generated by Karp-Miller-Rosenberg [5]. 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 $O(sort(n, \sigma) + kn)$ 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. An example is shown in Fig. 3.

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 $LCE[i] \geq t_\ell$.

Lemma 8. *The preprocessing algorithm described above generates the data structure described in Sect. 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 9. *The preprocessing algorithm described above takes $O(sort(n,\sigma)+kn)$ time.*

Proof. We first construct the SA and LCP arrays, which takes $O(sort(n,\sigma))$ time [1]. 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 LCP array. The total preprocessing time becomes $O(sort(n,\sigma) + kn)$. □

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.

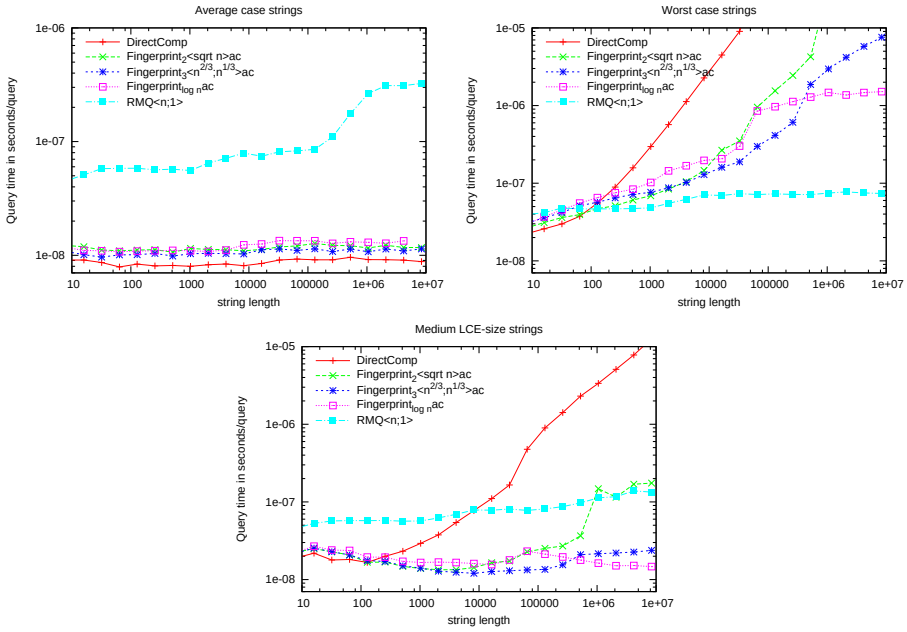

Fig. 4. Comparison of our new FINGERPRINT$_k$ algorithm for $k = 2$, $k = 3$ and $k = \lceil \log n \rceil$ versus the existing DIRECTCOMP and LCPRMQ algorithms.

Medium LCE value strings These strings have an average LCE value over all n^2 query pairs of $n/2r$, where $r = 0.73n^{0.42}$. These strings where constructed to show that there exists input strings where FINGERPRINT$_k$ is faster than both DIRECTCOMP and LCPRMQ at the same time. The strings consist of repeating substrings of r unique characters. The value of r was found experimentally.

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 we use the same string and the same million query pairs on all tested algorithms.

4.3 Results

Fig. 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

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.

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 the plot of strings with medium average LCE values, we see a case where 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 $O(kn^{1/k})$. 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 $H_\ell[((i-1) \mod t_\ell) \cdot \lceil n/t_\ell \rceil + \lfloor (i-1)/t_\ell \rfloor + 1]$. This will group all used fingerprints at level ℓ next to each other in memory, such that the amount of I/O 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

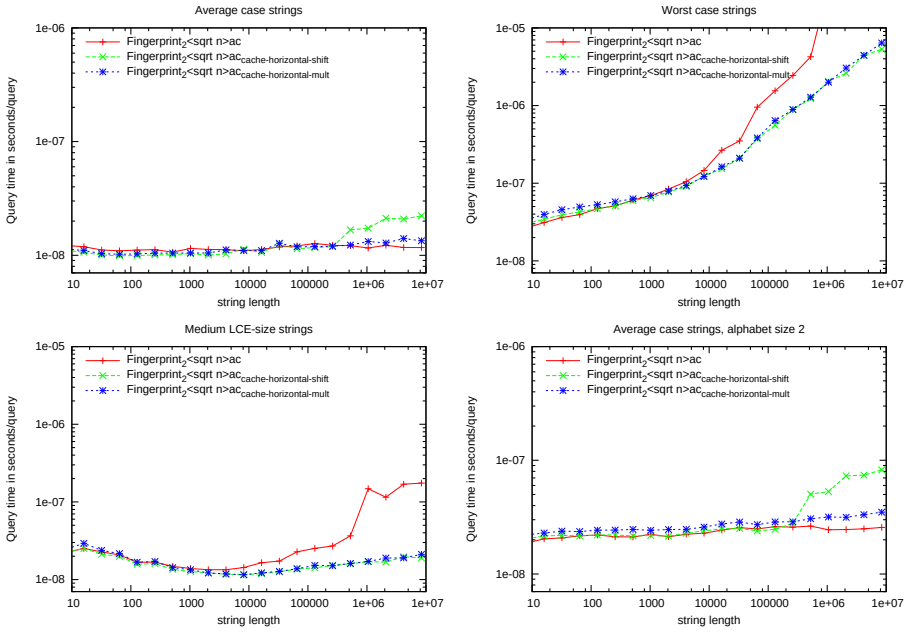

Fig. 5. Query times of FINGERPRINT$_2$ with and without cache optimization.

small. When $k = \lceil \log n \rceil$, this cache optimization gives no asymptotic 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.

Fig. 5 shows our measurements for FINGERPRINT$_2$. 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$_2$ variant with shift operations shows an increase in query times for large inputs, which we cannot explain. The last plot on Fig. 5 shows a variant of average case where the alphabet size is changed to two. This plot shows a bad case for cache optimized FINGERPRINT$_2$. 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 bad case plot we see that the cache optimized variant of FINGERPRINT$_2$ has only slightly worse query time compared to the variant, which is not cache optimized. Fig. 6 shows the measurements for FINGERPRINT$_3$. Unlike FINGERPRINT$_2$, the cache

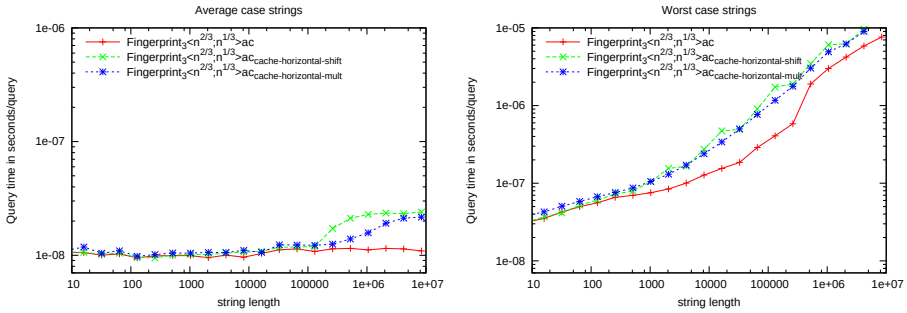

Fig. 6. Query times of FINGERPRINT$_3$ with and without cache optimization.

optimized variant is slightly slower than the unoptimized variant. Hence, our cache optimization is effective for $k = 2$ but not $k = 3$.

5 Conclusions

We have presented the FINGERPRINT$_k$ algorithm achieving the theoretical bounds of Thm. 1. We have demonstrated that the 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 shown 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. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. J. ACM 47(6), 987–1011 (2000)
2. Fischer, J., Heun, V.: Theoretical and practical improvements on the rmq-problem, with applications to LCA and LCE. In: Proc. 17th Symp. on Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 4009, pp. 36–48 (2006)
3. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
4. Ilie, L., Navarro, G., Tinta, L.: The longest common extension problem revisited and applications to approximate string searching. J. Disc. Alg. 8(4), 418 – 428 (2010)
5. Karp, R.M., Miller, R.E., Rosenberg, A.L.: Rapid identification of repeated patterns in strings, trees and arrays. In: Proc. 4th Symp. on Theory of Computing. pp. 125–136 (1972)
6. Landau, G.M., Vishkin, U.: Introducing efficient parallelism into approximate string matching and a new serial algorithm. In: Proc. 18th Symp. on Theory of Computing. pp. 220–230 (1986)