The Longest Common Extension Problem Revisited and Applications to Approximate String Searching*

LUCIAN ILIE[1†‡] GONZALO NAVARRO[2§]
LIVIU TINTA[1]
[1]Department of Computer Science, University of Western Ontario
N6A 5B7, London, Ontario, CANADA

[2]Department of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, CHILE

November 5, 2009

Abstract

The *Longest Common Extension* (LCE) problem considers a string s and computes, for each of a number of pairs (i, j), the longest substring of s that starts at both i and j. It appears as a subproblem in many fundamental string problems and can be solved by linear-time preprocessing of the string that allows (worst-case) constant-time computation for each pair. The two known approaches use powerful algorithms: either constant-time computation of the Lowest Common Ancestor in trees or constant-time computation of Range Minimum Queries in arrays. We show here that, from practical point of view, such complicated approaches are not needed. We give two very simple algorithms for this problem that require no preprocessing. The first is 5 times faster than the best previous algorithms on the average whereas the second is faster on virtually all inputs. As an application, we modify the Landau-Vishkin algorithm for approximate matching to use our simplest LCE algorithm. The obtained algorithm is 13 to 20 times faster than the original. We compare it with the more widely used Ukkonen's cutoff algorithm and show that it behaves better for a significant range of error thresholds.

---

[*] A preliminary version of this paper has been presented at SPIRE'09; see [9].

[†] Research supported in part by NSERC.

[‡] Corresponding author; e-mail: `ilie@csd.uwo.ca`

[§] Research supported in part by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile

1 Introduction

The *longest common extension (LCE)* problem takes as input a string s and many pairs (i, j) and computes, for each pair (i, j), the longest substring of s that occurs both starting at position i and at j in s. That is, the longest common prefix of the suffixes of s that start at positions i and j, respectively. Sometimes the problem receives two strings as input, s and t, and is required to compute, for each pair (i, j), the longest common prefix of the ith suffix of s and jth suffix of t. This reduces to the previous problem by considering the string $s\$t$, where \$ is a letter that does not appear in s and t.

The LCE problem appears as a subproblem in many fundamental string problems, such as k-mismatch problem and k-difference global alignment [15, 21, 16], computation of (exact or approximate) tandem repeats [17, 7, 14], or computing palindromes and matching with wild cards [6]. Very efficient algorithms are obtained and it is not clear how to solve those problems without employing LCE solutions.

The LCE problem can be optimally solved by linear-time preprocessing of the string s so that the answer for each pair (i, j) can be computed in constant time. Two powerful algorithms are employed to achieve this bound. The first is the constant-time computation of the Lowest Common Ancestor in trees (with linear-time preprocessing) [8, 23, 3, 2]. When applied to the suffix tree [6] of the string s, it easily yields the solution for the LCE problem. The second uses constant-time computation of Range Minimum Queries (RMQ) in arrays (with linear-time preprocessing) [3, 2, 20, 5]. Applied to the LCP array of s (that is part of the suffix array data structure of s, see Section 2), this gives again a solution of the LCE problem. The RMQ-based solution is more efficient in practice [5].

In this paper we look at the LCE problem from a practical point of view. Our aim is to provide simple and efficient algorithms. As it is often the case, the best worst-case algorithms need not be the fastest in practice. Indeed, already [5] considered a simplified algorithm that resolves each (i, j) pair in $\mathcal{O}(\log n)$ time (with linear-time preprocessing). This algorithm performs the best in practice.

Our starting point is the observation that, on the average, the LCE values are very small. We give the precise limit of this average, for a given alphabet size, when the string length goes to infinity. An important consequence is that the algorithm that directly compares the suffixes starting at positions i and j is optimal on the average and significantly faster in practice, on the average, than all previous ones. It needs only the string s; no preprocessing.

The fastest algorithm to date computes RMQ on the longest common prefix array (see Section 2). The distance between the positions corresponding to the given i and j in the suffix array is inversely proportional to the size of their LCE. For the vast majority of pairs, our algorithm described above is the fastest. When they are very close, there is another algorithm — direct computation of range minimum — that is the best. Combining the two and using the superior speed of the cache memory produces an algorithm that, while still very simple (no preprocessing required; uses only the existing LCP array), is the fastest on virtually all inputs.

Next we test the behavior of our algorithm in real applications. The approximate string searching algorithm of Landau-Vishkin [16] is using heavily LCE computations. When the current best LCE algorithm is replaced by our simplest one, the obtained algorithm runs 13 to 20 times faster in practice. We compare the obtained algorithm with the more widely used Ukkonen's cutoff algorithm [25] and show that it is faster for a significant range of error thresholds.

The paper is organized as follows. Section 2 contains the basic definitions including the LCE problem with its current solutions. The average LCE is precisely computed in Section 3 and a linear-time algorithm computing the average LCE for a given file is given in Section 4. Our fastest-on-average algorithm is given in Section 5 where extensive comparison with previous fastest algorithms is provided. The approach on the (practical) worst case starts in Section 6 with several approximations on the maximum LCE. The combined algorithm that is the fastest in practice is given in Section 7, together with the corresponding experiments. Section 8 contains the application to approximate string searching. We briefly recall the idea of Landau and Vishkin and then present experimental comparison results. The comparison with Ukkonen's algorithm is presented in Section 9. We summarize our achievements in the Conclusions section.

2 Basic definitions

Let A be an alphabet with $\mathrm{card}(A) = \ell \geq 2$. Let $s \in A^*$ be a string of length $|s| = n$. For any $1 \leq i \leq n$, the ith letter of s is $s[i]$ and $s[i\mathinner{..}j] = s[i]s[i+1]\cdots s[j]$. In this notation $s = s[1\mathinner{..}n]$. Let also suf_i denote the suffix $s[i\mathinner{..}n]$ of s. For $1 \leq i \neq j \leq n$, the length of the longest common prefix of the strings suf_i and suf_j is called the *longest common extension* of the two suffixes, denoted by $\mathsf{LCE}_s[i,j]$. When s is understood, it will be omitted.

Assuming a total order on the alphabet A, the *suffix array* of s, [18], denoted SA, gives the suffixes of s sorted ascendingly in lexicographical order, that is, $\mathsf{suf}_{\mathsf{SA}[1]} < \mathsf{suf}_{\mathsf{SA}[2]} < \cdots < \mathsf{suf}_{\mathsf{SA}[n]}$. The suffix array of the string $\mathsf{abbababba}$ is shown in the second column of Fig. 1. The suffix array is often used in combination with another array, the longest common prefix (LCP) array that gives the length of the longest common prefix between consecutive suffixes of SA, that is, $\mathsf{LCP}[i] = \mathsf{LCE}[\mathsf{SA}[i-1], \mathsf{SA}[i]]$; see the fourth column of Fig. 1 for an example. By definition, $\mathsf{LCP}[1] = 0$.

The suffix array of a string of length n over an integer alphabet can be computed in $\mathcal{O}(n)$ time by any of the algorithms in [10, 12, 13]. The longest common prefix array can be computed also in $\mathcal{O}(n)$ time by the algorithm of [11].

The *LCE problem* is: given a string s and a set of pairs (i, j), compute $\mathsf{LCE}(i, j)$ for each pair. It can be solved by preprocessing the string s in linear time so that each $\mathsf{LCE}(i, j)$ is computed in constant time. The first solution uses constant-time computation of the Lowest Common Ancestor [8, 23, 3, 2] applied to the suffix tree; see an example in Figure 1. The second, more effi-

i	SA[i]	suff$_{\mathsf{SA}[i]}$	LCP[i]
1	9	a	0
2	4	ababba	1
3	6	abba	2
4	1	abbababba	4
5	8	ba	0
6	3	bababba	2
7	5	babba	3
8	7	bba	1
9	2	bbababba	3

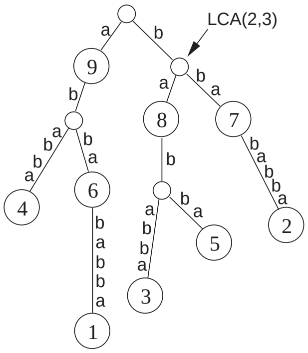

Figure 1: The SA and LCP arrays (left) and the suffix tree (right) for the string abbababba. We have $\mathsf{LCE}(2,3) = \mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[3]+1, \mathsf{SA}^{-1}[2]) = \mathsf{RMQ}_{\mathsf{LCP}}(7,9) = 1$; this is also the depth $|\mathsf{b}|$ of the node $\mathsf{LCA}(3,2)$ in the suffix tree.

cient, uses constant-time computation of Range Minimum Queries (RMQ) in arrays [3, 2, 20, 5] applied to the LCP array. In general, we have $\mathsf{LCE}(i,j) = \mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{SA}^{-1}[i]+1, \mathsf{SA}^{-1}[j])$. Note the need for the inverse suffix array SA^{-1}; an example is shown in Figure 1.

We shall denote the LCE algorithm of [5] based on constant-time RMQ computation by $\mathrm{RMQ}_{\mathrm{const}}$. The practically most efficient algorithm of [5] computes each $\mathsf{LCE}(i,j)$ in (suboptimal) $\mathcal{O}(\log n)$ time; it will be denoted by $\mathrm{RMQ}_{\log}$.

3 Average LCE

The starting point of our approach is the observation that most LCE values are very small. The main result of this section estimates the average value of the LCE over all strings of a given length n, that is,

$$\mathsf{Avg\_LCE}(n,\ell) \quad = \quad \frac{1}{\ell^n} \sum_{s \in A^n} \Big(\frac{1}{\binom{n}{2}} \sum_{1 \le i < j \le n} \mathsf{LCE}_s(i,j) \Big)$$

Theorem 1 *(i) For any* $\ell \ge 2$, $\lim_{n \to \infty} \mathsf{Avg\_LCE}(n,\ell) = \frac{1}{\ell - 1}$.

(ii) For any $n \ge 2$ *and* $\ell \ge 2$, $\mathsf{Avg\_LCE}(n,\ell) < \frac{1}{\ell-1}$.

Proof. Reorganizing the formula for $\mathsf{Avg\_LCE}(n,\ell)$ gives

$$\mathsf{Avg\_LCE}(n,\ell) \quad = \quad \frac{2}{n(n-1)\ell^n} \sum_{k=1}^{n-1} k \sum_{1 \le i < j \le n-k+1} \mathrm{card}(\{s \mid \mathsf{LCE}_s(i,j) = k\})$$

(i) For fixed k, i, j, denote $K_{k,i,j} = \{s \mid \mathsf{LCE}_s(i,j) = k\}$. We compute the cardinality of $K_{k,i,j}$. Recall that, in any string $s \in K_{k,i,j}$, we have $s[i\,..\,i+k-1] = s[j\,..\,j+k-1]$.

(i.1) Assume first that $j \leq n-k$. If also $j-i \geq k$, then there are ℓ^k possibilities for the strings letters contained in the substrings $s[i\,..\,i+k-1]$ and $s[j\,..\,j+k-1]$. The letters right after those, $s[i+k]$ and $s[j+k]$, can be chosen in $\ell(\ell-1)$ different ways as they must be different. There are $\ell^{n-2(k+1)}$ possibilities to choose the remaining letters of s. In total we obtain $\mathrm{card}(K_{k,i,j}) = \ell^{n-k-1}(\ell-1)$.

Now, if $j-i<k$, then $s[i\,..\,i+k-1] = x^p x'$, with $|x| = j-i$, x' a prefix of x, and $p \geq 1$. The letters contained in the substrings $s[i\,..\,i+k-1]$ and $s[j\,..\,j+k-1]$ are completely determined by x which can be any string out of ℓ^{j-i} possibilities. The letter in position $j+k$ can be chosen in $\ell-1$ ways, since it has to be different from $s[i+k]$. The remaining letters can be chosen in $\ell^{n-(k+j-i+1)}$ ways. In total, $\mathrm{card}(K_{k,i,j}) = \ell^{n-k-1}(\ell-1)$.

(i.2) Assume next $j = n-k+1$. We no longer need the condition that $s[i+k] \neq s[j+k]$, as above, since $s[j+k]$ is undefined. Therefore, by a reasoning similar to the one above, $\mathrm{card}(K_{k,i,j}) = \ell^{n-k}$.

There are $\binom{n-k}{2}$ pairs (i,j) verifying (i.1) above and $n-k$ that verify (i.2). Consequently, we obtain (i) as follows:

$$\begin{aligned}
\mathsf{Avg\_LCE}(n,\ell) &= \frac{2}{n(n-1)\ell^n}\sum_{k=1}^{n-1} k\left(\binom{n-k}{2}\ell^{n-k-1}(\ell-1) + (n-k)\ell^{n-k}\right)\\
&= \frac{2}{n(n-1)\ell^n}\sum_{k=1}^{n-1}(n-k)\left(\binom{k}{2}\ell^{k-1}(\ell-1) + k\ell^k\right)\\
&= \frac{1}{n(n-1)\ell^n}\sum_{k=1}^{n-1}(n-k)\left(k(k+1)\ell^k - k(k-1)\ell^{k-1}\right)\\
&= \frac{1}{n(n-1)\ell^n}\left(n(n-1)\ell^{n-1} + \sum_{k=1}^{n-1}k(k-1)\ell^{k-1}\right)\\
&= \frac{n}{n-1}\frac{1}{\ell-1} - \frac{1}{n-1}\frac{\ell+1}{(\ell-1)^2} + \frac{1}{n(n-1)}\frac{2\ell(\ell^n-1)}{\ell^n(\ell-1)^3}\\
&\xrightarrow{n\to\infty} \frac{1}{\ell-1}\,.
\end{aligned}$$

(ii) Using the second last line of the above calculation, we obtain:

$$\begin{aligned}
\mathsf{Avg\_LCE}(n,\ell) &< \frac{n}{n-1}\frac{1}{\ell-1} - \frac{1}{n-1}\frac{\ell+1}{(\ell-1)^2} + \frac{1}{n(n-1)}\frac{2\ell}{(\ell-1)^3}\\
&= \frac{1}{\ell-1} + \frac{2(n+\ell)-2n\ell}{n(n-1)(\ell-1)^3}\\
&\leq \frac{1}{\ell-1}\,.
\end{aligned}$$

□

4 Average LCE for a fixed text in linear time

The alphabet size ℓ is reasonably large for usual texts and therefore the expected value for the average LCE is quite low. However, we assumed an independent uniform distribution of letters which does not happen in practice. We compute in this section the average LCE for the text files in the Canterbury[1], Manzini[2], and Pizza&Chili[3] corpora, as well as for some random files we generated.

For a file of length n, naively computing the average LCE would require the computation of quadratically many LCE values. We give an algorithm that computes it in linear time. For a string of length n, there are $\frac{n(n-1)}{2}$ LCE values, since $\mathsf{LCE}[i,i]$ is undefined and $\mathsf{LCE}[i,j] = \mathsf{LCE}[j,i]$. Figure 2(i) gives the LCE matrix for the string abbababba. In order to be able to compute the average in linear time, we reorder first the rows and columns according to the permutation given by the SA array; see Figure 2(ii). The matrix is symmetric before the permutation and remains so after, therefore we consider only the top half (in black). The main diagonal is undefined but the one immediately above it is the LCP array (without the first, useless, position). The element in position (i,j) in the permuted matrix contains the minimum value of $\mathsf{LCP}[i\,..\,j]$. Therefore, the upper half of the matrix can be partitioned into rectangles containing elements of the same value and which have a corner on the LCP diagonal. The sides of the rectangle containing the ith element of the LCP diagonal are equal to the distances from the ith element of LCP to the closest previous (next, resp.) smaller element (or to the end of the array, if such an element does not exist); two such rectangles are shaded in Figure 2(ii).

$$
\begin{array}{c|ccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 &   & 0 & 0 & 2 & 0 & 4 & 0 & 0 & 1 \\
2 & 0 &   & 1 & 0 & 1 & 0 & 3 & 1 & 0 \\
3 & 0 & 1 &   & 0 & 3 & 0 & 1 & 2 & 0 \\
4 & 2 & 0 & 0 &   & 0 & 2 & 0 & 0 & 1 \\
5 & 0 & 1 & 3 & 0 &   & 0 & 1 & 2 & 0 \\
6 & 4 & 0 & 0 & 2 & 0 &   & 0 & 0 & 1 \\
7 & 0 & 3 & 1 & 0 & 1 & 0 &   & 1 & 0 \\
8 & 0 & 1 & 2 & 0 & 2 & 0 & 1 &   & 0 \\
9 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 &
\end{array}
\qquad
\begin{array}{c|ccccccccc}
 & 9 & 4 & 6 & 1 & 8 & 3 & 5 & 7 & 2 \\
\hline
9 &   & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 &   & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\
6 & 1 & 2 &   & 4 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 2 & 4 &   & 0 & 0 & 0 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 &   & 2 & 2 & 1 & 1 \\
3 & 0 & 0 & 0 & 0 & 2 &   & 3 & 1 & 1 \\
5 & 0 & 0 & 0 & 0 & 2 & 3 &   & 1 & 1 \\
7 & 0 & 0 & 0 & 0 & 1 & 1 & 1 &   & 3 \\
2 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 3 &
\end{array}
$$

(i) (ii)

Figure 2: (i) The LCE matrix corresponding to the string abbababba and (ii) the same matrix where the rows and columns are permuted according to the array $\mathsf{SA} = (9,4,6,1,8,3,5,7,2)$. The longest diagonal is the array $(1,2,4,0,2,3,1,3)$, that is, LCP without the first element. Each shaded block contains elements of the same value as the one on the LCP diagonal.

Therefore, the sum of all LCE values can be computed by a single pass through the LCP array with an additional stack that enables computation of the rectangle sizes. The algorithm is shown in Figure 3. The stack contains pairs of

---

[1] `http://corpus.canterbury.ac.nz/`

[2] `http://web.unipmn.it/ manzini/lightweight/corpus/`

[3] `http://pizzachili.dcc.uchile.cl/`

the form $(\mathsf{LCP}[i+1], i)$. In order to treat all elements in the same way, we push at the beginning the pair $(0,0)$ and at the end the pair $(\mathsf{LCP}[n+1]=0, n)$. The algorithm runs in linear time because each element is pushed onto the stack and popped out of the stack only once.

```
ComputeAvgLCE($s$)
$\mathsf{LCP}[n+1] \leftarrow 0$
sum $\leftarrow 0$; $\mathscr{S} \Leftarrow \emptyset$; $\textsc{push}((0,0), \mathscr{S})$
for $i$ from $1$ to $n$ do
    if $(\mathsf{LCP}[i+1] \geq \textsc{top}(\mathscr{S})_1)$ then
        $\textsc{push}((\mathsf{LCP}[i+1], i), \mathscr{S})$
    else
        while $(\mathsf{LCP}[i+1] < \textsc{top}(\mathscr{S})_1)$ do
            $(x_1, x_2) \leftarrow \textsc{pop}(\mathscr{S})$
            sum $\leftarrow$ sum $+ x_1(i - x_2)(x_2 - \textsc{top}(\mathscr{S})_2)$
return $\frac{2}{n(n-1)}$sum
```

Figure 3: Computing the average LCE for a given text in linear time; $\textsc{top}(\mathscr{S})_i, i \in \{1,2\}$, refers to the ith element of the pair $\textsc{top}(\mathscr{S})$.

We used the ComputeAvgLCE algorithm for the files in Table 1. For small files our LCE algorithms are far better than any other ones, so we discuss only the five largest files in Canterbury corpus. (The other corpora contain only large files.) The results are shown in the fifth column of Table 1. While the LCE averages are higher than expected according to Theorem 1, they are still small (at most 1).

5 An average-case optimal algorithm for LCE

This result in Theorem 1 has an important implication for our purpose, that is, no sophisticated algorithms are necessary for computing LCEs. By Theorem 1, direct comparison of the two suffixes requires, on the average, $\frac{\ell}{\ell-1}$ comparisons. Therefore our DirectComp algorithm (see Figure 4) is optimal on the average.

```
DirectComp($s, i, j$)
$t \leftarrow 0$
while $\big((s[i+t] = s[j+t])$ and $(j+t \leq n)\big)$ do
    $t \leftarrow t+1$
return $t$
```

Figure 4: Computing LCE by direct comparison.

We tested the DirectComp algorithm on the files in Table 1, and compared it with $\mathrm{RMQ}_{\mathrm{const}}$ and $\mathrm{RMQ}_{\log}$; the results are shown in the last three columns. All tests were done on a Sun Fire V440 Server, using one UltraSPARC IIIi processor at 1593MHz, 1MB L2 Cache, 4GB RAM, running SunOS 5.10. The programs were compiled using `gcc 3.4.3` with options `-O3 -fomit-frame-pointer`. One million random (i, j) pairs were generated and all three algorithms were run on those. Each experiment was repeated three times and the average times are shown. The preprocessing times for $\mathrm{RMQ}_{\mathrm{const}}$ and $\mathrm{RMQ}_{\mathrm{log}}$ were not counted.

	File	size	alph.	Avg_LCE	max_LCE	$\mathrm{RMQ}_{\mathrm{const}}$	$\mathrm{RMQ}_{\mathrm{log}}$	DirectComp
Canterbury	book1	0.7	82	0.0736	104	1.34	1.11	0.07
	kennedy.xls	1	256	0.3946	18	1.37	1.17	0.11
	E.coli	4.4	4	0.3371	2815	1.43	1.12	0.21
	bible.txt	3.9	63	0.0915	551	1.28	1.00	0.21
	world192.txt	2.3	93	0.0693	543	1.41	1.21	0.20
Manzini	chr22.dna[1]	33	4	0.3419	1777	1.46	1.17	0.20
	etext99	100	146	0.0732	286352	1.53	1.20	0.21
	howto	38	197	0.0909	70720	1.51	1.20	0.21
	jdk13c	66	113	0.0444	37334	1.44	1.16	0.22
	rctail96	109	93	0.0692	26597	1.50	1.21	0.22
	rfc	111	120	0.2140	3445	1.50	1.21	0.21
	sprot34.dat	105	66	0.0860	7373	1.49	1.20	0.22
	w3c2	99	256	0.0341	990053	1.50	1.22	0.21
Pizza&Chili	sources	201	230	0.0497	307871	—	—	0.20
	pitches	53	133	0.0420	25178	1.63	1.28	0.20
	proteins	1129	27	0.0625	647051	—	—	0.20
	DNA	385	16	0.3500	1378596	—	—	0.21
	English	2108	239	0.0753	4735603	—	—	0.22
	XML	282	96	0.0538	1084	—	—	0.20
random	rand_100_2	100	2	1.0000	52	1.51	1.23	0.29
	rand_100_4	100	4	0.3333	26	1.52	1.22	0.27
	rand_100_20	100	20	0.0526	11	1.48	1.23	0.28
	rand_1000_2	1000	2	1.0000	55	—	—	0.31
	rand_1000_4	1000	4	0.3333	29	—	—	0.30
	rand_1000_20	1000	20	0.0526	13	—	—	0.30

Table 1: Files from Canterbury (five largest ones), Manzini, and Pizza&Chili corpora and some randomly generated with various sizes and number of letters. The first six columns contain, in order: file source, file name, size (in megabytes), alphabet size, average LCE, maximum LCE. The last three contain the average running times for solving the LCE problem using $\mathrm{RMQ}_{\mathrm{const}}$, $\mathrm{RMQ}_{\mathrm{log}}$, and DirectComp, resp., given in microseconds per input pair. DirectComp is roughly 6 times faster. Also, the first two algorithms require the SA^{-1} and LCP arrays and further preprocessing, whereas our algorithm uses only the text without any preprocessing. The first two algorithms ran out of memory for files larger than 160MB.

Our algorithm is roughly 5 times faster than $\mathrm{RMQ}_{\mathrm{log}}$, the previous fastest algorithm. Recall here that the preprocessing times of $\mathrm{RMQ}_{\mathrm{const}}$ and $\mathrm{RMQ}_{\mathrm{log}}$ have not been considered. (Our comparison between $\mathrm{RMQ}_{\mathrm{const}}$ and $\mathrm{RMQ}_{\mathrm{log}}$

gives results similar to [5].) Due to the additional space needed (for a file of size n, more than $16n$ bytes are needed, in addition to $8n$ bytes for the LCP and SA^{-1} arrays), the RMQ-based algorithms could not handle files large than 160 MB (see also Table 2).

6 Maximum LCE

As seen in the previous section, our DirectComp algorithm performs significantly better than the best ones to date on the average. However, when counting the expected number of operations performed by each algorithm, the difference should be even bigger. That is due to the lower speed of RAM compared to cache memory. Most of the time is spent on accessing the large arrays. We turn this property into our advantage by trying to do better not only on the average but also in the worst case.

In this section we prove a number of results that help us get an idea of how large the maximum LCE is expected to be as well as an estimate on how many "large" LCE values are expected. Denoting $\mathsf{max\_LCE}(s) = \max\limits_{i,j} \mathsf{lcp}_s(i,j)$, we have the following theorem:

Theorem 2 *For any $n \geq 2$ and $\ell \geq 2$, we have*

(i) For any $s \in A^n$, $\mathsf{max\_LCE}(s) > \log_\ell(n) - 2$.

(ii) There exists an $s \in A^n$ such that $\mathsf{max\_LCE}(s) < \log_\ell(n)$.

(iii) The average maximum LCE, $\mathsf{Avg\_max\_LCE}(n,\ell)$, satisfies

$$\log_\ell(n) - 2 \leq \mathsf{Avg\_max\_LCE}(n,\ell) \leq 2\log_\ell(n)\ .$$

(iv) The average number of pairs (i,j) with $\mathsf{LCE}(i,j) \geq \log_\ell(n)$ is less than $n/2$.

(v) The average number of pairs (i,j) with $\mathsf{LCE}(i,j) \geq 2\log_\ell(n)$ is less than $1/2$.

Proof. (i) Consider a string $s \in A^n$and put $k = \mathsf{max\_LCE}(s)$. That means any two substrings of length $k+1$ of s are different. Since s has $n-k$ such substrings, it must be that $n - k \leq \ell^{k+1}$. From this (i) follows.

(ii) We use de Bruijn strings [4]. For a given ℓ and k, a de Bruijn string has all strings of length k as substrings and minimum length $n = \ell^k + k - 1$. Therefore $\mathsf{max\_LCE}(s) = k - 1$ as all substrings of length k are different and (ii) follows.

(iii) The first inequality follows immediately from (i). For the second, consider a string s such that $\mathsf{max\_LCE}(s) \geq k$. This means there is a position i

---

[1] For the file `chr22.dna` the stretches of unknown bases, `NN...N`, were not considered in all LCE computations, consistent with any use of this file.

such that $s[i\,..\,i+k-1]$ appears twice in s. The number of such strings is at most $(n-k+1)\ell^{n-k}$ since the factor $s[i\,..\,i+k-1]$ is completely determined by its second occurrence even in the case when the two occurrences overlap. Therefore, bounding the $\mathsf{max\_LCE}$ of all these strings by the maximum possible value $n-1$ and the remaining ones by $k-1$, we obtain

$$\begin{aligned}\mathsf{Avg\_max\_LCE}(n,\ell) &= \frac{1}{\ell^n}\sum_{s\in A^n}\mathsf{max\_LCE}(s)\\ &\le \frac{1}{\ell^n}\Big((n-1)(n-k+1)\ell^{n-k}+(k-1)(\ell^n-(n-k+1)\ell^{n-k})\Big)\\ &= k-1+\frac{1}{\ell^k}(n-k)(n-k+1)\ .\end{aligned}$$

For $k = 2\log_\ell(n)$, this gives the second inequality.

(iv) We make a reasoning similar to the one in the proof of Theorem 1(i). Denoting the average we are looking for by $\mathsf{Avg\_LCE}_{\log}(n,\ell)$, we have

$$\begin{aligned}\mathsf{Avg\_LCE}_{\log}(n,\ell) &= \frac{1}{\ell^n}\sum_{s\in A^n}\mathrm{card}(\{(i,j)\mid \mathsf{LCE}_s(i,j)\ge\log_\ell(n)\})\\ &= \frac{1}{\ell^n}\sum_{k=\lceil\log_\ell(n)\rceil}^{n-1}\ \sum_{1\le i<j\le n-k+1}\mathrm{card}(\{s\mid \mathsf{LCE}_s(i,j)=k\})\\ &= \frac{1}{\ell^n}\sum_{k=1}^{n-\lceil\log_\ell(n)\rceil}\left(\binom{k}{2}\ell^{k-1}(\ell-1)+k\ell^k\right)\\ &= \frac{1}{2\ell^n}\sum_{k=1}^{n-\lceil\log_\ell(n)\rceil}\big(k(k+1)\ell^k-(k-1)k\ell^{k-1}\big)\\ &= \frac{1}{2\ell^n}(n-\lceil\log_\ell(n)\rceil)(n-\lceil\log_\ell(n)\rceil+1)\ell^{n-\lceil\log_\ell(n)\rceil}\\ &< \frac{n^2}{2\ell^{\lceil\log_\ell(n)\rceil}}\le\frac{n}{2}\ .\end{aligned}$$

(v) The reasoning is the same as the one for (iv) except that $\lceil\log_\ell(n)\rceil$ is replaced by $\lceil 2\log_\ell(n)\rceil$ which gives the bound $1/2$. □ □

The conclusion of this section is that most LCE values are expected to be small and therefore our DirectComp algorithm performs better for most pairs. For the remaining few, we look for a different solution in the next section. The maximum LCE can be much larger than expected (see the sixth column of Table 1) but our solution avoids the large LCE values.

7 The worst case

The RMQ-based algorithms are better for a very small fraction of the input (i,j) pairs, namely those for which the difference between $\mathsf{SA}^{-1}[i]$ and $\mathsf{SA}^{-1}[j]$ is very small, as that usually implies large $\mathsf{LCE}[i,j]$ value. But, for such cases, there is another, very simple, algorithm, already considered by [5], that performs the best. It requires no preprocessing. Instead, it computes directly the minimum of the values $\mathsf{LCP}[\mathsf{SA}^{-1}[i]+1\,..\,\mathsf{SA}^{-1}[j]]$. This algorithm, called DirectMin, is described below.

```
DirectMin($\mathsf{LCP}, i, j$)
$low \leftarrow \min(\mathsf{SA}^{-1}[i], \mathsf{SA}^{-1}[j])$
$high \leftarrow \max(\mathsf{SA}^{-1}[i], \mathsf{SA}^{-1}[j])$
$t \leftarrow \mathsf{LCP}[low + 1]$
for $k$ from $low + 2$ to $high$ do
    if $\mathsf{LCP}[k] < t$ then $t \leftarrow \mathsf{LCP}[k]$
return $t$
```

Figure 5: Direct computation of the range minimum.

Table 2 contains a summary of the memory and preprocessing requirements for each of the four algorithms: $\mathrm{RMQ}_{\mathrm{const}}$, $\mathrm{RMQ}_{\mathrm{log}}$, DirectMin, and DirectComp. The first two require the SA^{-1} array to compute the corresponding positions in the LCP array and the data structures necessary for the constant- (logarithmic-, resp.) time computation of the RMQ values. DirectMin requires SA^{-1} and LCP for the same reason but no additional space. DirectComps needs only the text.

Algorithm	$\mathrm{RMQ}_{\mathrm{const}}$	$\mathrm{RMQ}_{\mathrm{log}}$	DirectMin	DirectComp
Preprocessing	RMQ data structures, SA^{-1}, LCP		SA^{-1}, LCP	—
Memory (bytes)	$24n+$		$8n$	n

Table 2: Preprocessing and memory requirements for a file of size n; we assume an integer is represented on 4 bytes.

We tested the performance of all four algorithms discussed for the files in Table 1. We run them on pairs at a given distance, $\mathtt{step} = |\,\mathsf{SA}^{-1}[j] - \mathsf{SA}^{-1}[i]|$, in the suffix array, represented on the abscissa in logarithmic scale; the ordinate gives the time in microseconds. All pairs at a given distance have been considered for each computation. The results are given in Figures 6-8. Again, all preprocessing time have been discarded.

8 Landau-Vishkin algorithm

An important application of LCE algorithms is to approximate string search. Landau and Vishkin [16] adapted an idea of Ukkonen [24] to obtain an algorithm that searches occurrences that have no more than k differences in a text of length n in time $\mathcal{O}(kn)$. We recall briefly the idea. Consider the pattern p of length m, the text t of length n and build the well-known dynamic programming matrix for searching occurrences of p in t. The one for $p = \mathsf{codes}$, $t = \mathsf{coincidence}$ is shown in Figure 9(i).

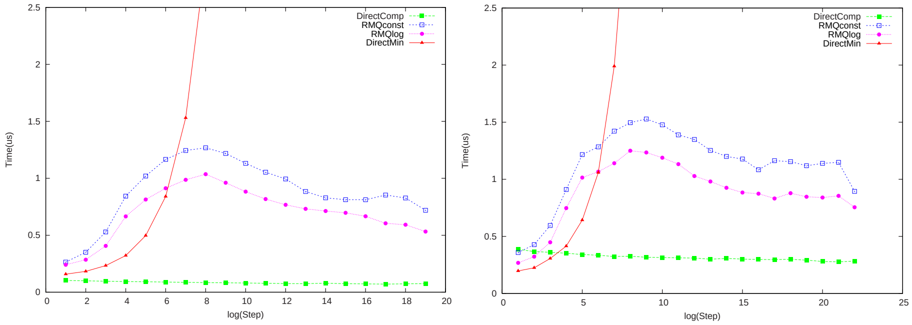

Figure 6: The files `book1` (left) and `E.coli` (right). The behavior of DirectMin, $\text{RMQ}_{\text{const}}$, and RMQ_{log} for the other three files from Canterbury corpus is similar; the curve of DirectComp (green) is in-between the two cases shown. (For files of size less than 1MB, DirectComp is the best on all inputs.)

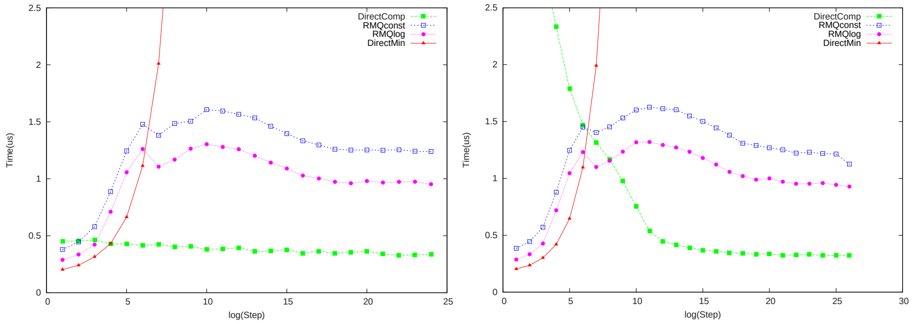

Figure 7: The files `chr22.dna` (left) and `jdk13c` (right). The behavior of DirectMin, $\text{RMQ}_{\text{const}}$, and RMQ_{log} for the other other files in Manzini corpus (and `pitches` from Pizza&Chile) is similar; the curve of DirectComp is in-between the two cases shown. The file `jdk13c` is the only one where the combination DirectComp-DirectMin is slightly slower than RMQ_{log} on a very small interval.

A d-*path* in the DP matrix is a path that starts in row 0 and specifies a total of d mismatches and indels. Diagonal i is the diagonal containing cells for which the difference between the column and row index is i.

A d-path is *farthest reaching* in diagonal i if it ends on diagonal i and its end has a higher row index than any other d-path. Any farthest reaching k-path that reaches row m specifies the end of an occurrence of p with k errors. In Figure 9(i), the diagonals 3 and 4 contain end points of 2-paths that reach the last row. They correspond to the occurrences cide and ciden of the pattern with 2 errors.

Landau-Vishkin algorithm computes the ends of all farthest reaching k-paths. To compute the end of the farthest reaching d-path in diagonal i, one considers the farthest reaching of the following three paths. First, the farthest

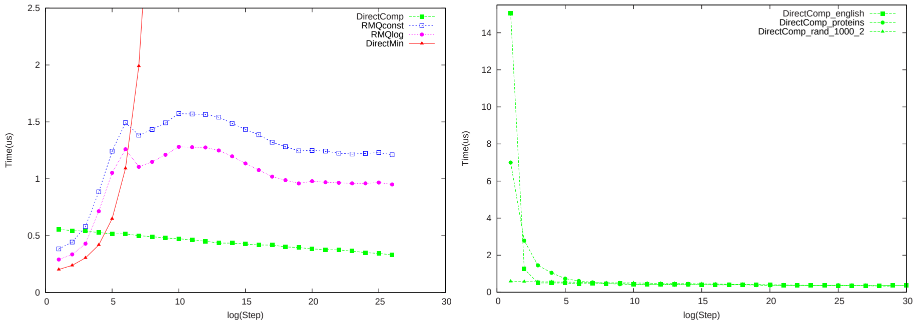

Figure 8: The file `rand_100_2` (left; the behavior on the other two random files of the same size is similar) and the three largest files, `rand_1000_2`, `English` and `proteins` (right); only DirectComp can handle those. The performance is impressive; only at distance 2 some of the times are higher; such a case would be very easily handled by DirectMin given enough space for the SA^{-1} and LCP arrays.

		c	o	i	n	c	i	d	e	n	c	e
	0	0	0	0	0	0	0	0	0	0	0	0
c	1	0	1	1	1	0	1	1	1	1	0	1
o	2	1	0	1	2	1	1	2	$\underline{2}$	2	1	1
d	3	2	1	1	$\underline{2}$	$\underline{2}$	2	1	2	3	2	$\underline{2}$
e	4	3	$\underline{2}$	$\underline{2}$	$\underline{2}$	3	3	2	1	$\underline{2}$	3	$\underline{2}$
s	5	4	3	3	3	3	4	3	$\underline{2}$	$\underline{2}$	3	3

(i)

	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
0			2	0	0	0	1	0	0	0	0	1	0	0
1		3	3	2	1	2	4	1	1	1	2	2	1	0
2	4	4	4	3	3	5	5	4	2	4	3	2	1	0

(ii)

Figure 9: (i) Dynamic programming matrix for searching the pattern codes in the text coincidence. The ends of the farthest reaching 2-paths are underlined. (ii) The rows containing the ends of the farthest reaching d-paths, $0 \leq d \leq 2$.

reaching $(d-1)$-path in diagonal $i-1$ (say it ends in row j) is extended by an insertion in t and then an LCE between positions $i+j$ in t and j in p. Similarly, the farthest reaching $(d-1)$-paths in diagonals i and $i+1$ are extended by a deletion/mismatch followed by an LCE. This way, all ends of d-paths are computed from ends of $(d-1)$-paths. Since each LCE can be computed in constant time, the whole algorithm requires time $\mathcal{O}(kn)$. These ends are computed in Figure 9(ii) for $0 \leq d \leq 2$.

In our algorithm the constant-time LCE algorithm is replaced by the newly introduced DirectComp; the original algorithm is denoted by LV whereas ours is $\mathrm{LV}_{\mathrm{DC}}$. Here, as opposed to the LCE case, preprocessing for constant time LCE is part of the algorithm and is counted. First, Table 3 gives the memory and preprocessing requirements for the two algorithms.

We compared the two algorithms on two files for various pattern and error sizes. (The same machine as in Section 5. The results are shown in Figures 10-12. For each file, half of the patterns were randomly picked from the text (and

Algorithm	LV	LV_{DC}
Preprocessing	SA, SA^{-1}, LCP, RMQ data structures,	—
Memory (bytes)	$28n+$	$5n$

Table 3: Preprocessing and memory requirements for a file of size n; we assume an integer is represented on 4 bytes.

the corresponding number of errors were randomly introduced) whereas the other half were randomly generated over the alphabet the text. Our algorithm is 13 to 20 times faster. For a fixed text, the time were affected only by the number of errors and not the size or type of pattern.

File `chr22` from Manzini, size 32MB				
pat. source	pat. length	errors	LV	LV_{DC}
rand. pick from text	10	3	206	13
	20	6	333	23
	50	20	970	74
	100	20	959	74
	1000	20	946	73
rand. gen. over alph.	10	3	201	12
	20	6	340	23
	50	20	952	73
	100	20	944	73
	1000	20	944	73

Figure 10: Comparison between the original Landau-Vishkin algorithm and ours for the file `chr22` from Manzini corpus. We used the algorithm of Manzini-Ferragina [19] for computing the suffix array, the one of Kasai et al. [11] for the longest common prefix array, and the algorithm of Fischer and Heun [5] for the RMQ-based computation of LCE.

9 Modified Landau-Vishkin versus Ukkonen's cutoff

We compared experimentally the improved Landau-Vishkin algorithm (LV_{DC}) with the more widely used Ukkonen's cutoff algorithm [25]. The latter is $O(kn)$ on average and rather practical among the classical algorithms. The former, instead, guarantees $O(kn)$ worst-case. The Landau-Vishkin algorithm has always been regarded as an impractical algorithm [22]. With the improved LCE algorithm, a competitive algorithm, LV_{DC}, is obtained. Notice that, due to Theorem 1, LV_{DC} is $O(kn)$ time on the average.

Our machine is an Intel Core2 Duo, each of the two cores containing a 3 GHz processor with 6 MB cache, and 8 GB RAM. It runs `Gnu Linux 2.6.24-24-server`.

Prefix of file `English` from Pizza&Chili, size 50MB				
pat. source	pat. length	errors	LV	$\mathrm{LV}_{\mathrm{DC}}$
rand. pick from text	10	3	344	18
	20	6	572	34
	50	20	1557	106
	100	20	1536	106
	1000	20	1546	104
rand. gen. over alph.	10	3	353	18
	20	6	562	33
	50	20	1497	105
	100	20	1497	105
	1000	20	1481	104

Figure 11: Comparison between the original Landau-Vishkin algorithm and ours for the prefix of 50MB of the file `English` from Pizza&Chili corpus.

Prefix of file `English` from Pizza&Chili, size 750MB				
pat. source	pat. length	errors	LV	$\mathrm{LV}_{\mathrm{DC}}$
text	1000	20	—	1592
rand. gen.	1000	20	—	1574

Figure 12: The times for running our program on a prefix of 750MB of the file `English` from Pizza&Chili corpus. The original Landau-Vishkin algorithm cannot run on files larger than 140MB.

The compiler is `Gnu gcc` using full optimization, and the experiments ran without any other significant process competing for the CPU. We measure user times.

We have used 100MB of different text types from Pizza&Chili: Proteins, DNA, MIDI pitches, English, C/Java source code, and XML text. Each data point is the average over 100 searches for a pattern randomly chosen from the text, which yields a standard deviation for the estimator (usually well) below 2% of the mean. Because the search times turned out to be largely independent on m, we fix $m = 50$ and give the results for increasing k values.

Figure 13 shows the results. As it can be seen, $\mathrm{LV}_{\mathrm{DC}}$ is faster than Ukkonen's for low k values, which are usually the most interesting ones for approximate string matching. At some turnover point (usually around $k = 5$–15, growing for larger alphabets) the result reverses and Ukkonen's becomes faster, yet never for much more than 10%. For low k, instead, $\mathrm{LV}_{\mathrm{DC}}$ can be up to twice as fast. This shows that the technique of computing the longest common prefix by brute force is indeed practical, and it yields to improving a widely used algorithm for approximate string matching (especially for verification of short text areas pointed out by a faster filtration algorithm).

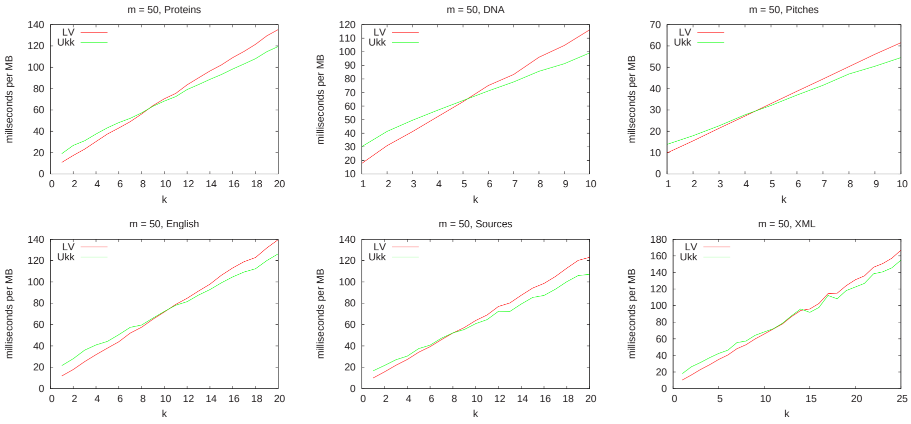

Figure 13: Time comparison between the Landau-Vishkin algorithm with the fast LCE algorithm, $\mathrm{LV}_{\mathrm{DC}}$ (LV in the figure) and the classical Ukonnen's cutoff algorithm (Ukk in the figure).

10 Conclusions

We gave very simple algorithms for the LCE problem that are the best in practice with respect to both time and space. When the pairs are randomly distributed, DirectComp should be used as it is approximately 5 times faster on the average than the current fastest algorithm. If the performance on every single input matters, then the combination DirectComp-DirectMin should be used. Only DirectComp can handle very large files and the performance on those is very good.

In order to test the efficiency of our new algorithms, we presented an application to approximate string searching. Landau-Vishkin algorithm uses heavily LCE algorithms. When those were replaced by our DirectComp, the obtained algorithm runs 13 to 20 times faster, is much simpler, and uses much less space.

Our improvement turns Landau-Vishkin's algorithm form an impractical algorithm to a practical one. We compared it with Ukkonen's cutoff algorithm and proved it to be faster for a significant range of error thresholds.

Acknowledgements

The statistics of the large files were computed on SHARCNET (`www.sharcnet.ca`).

References

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlenbusch, Replacing suffix trees with enhanced suffix arrays, *J. Discrete Algorithms* **2** (2004) 53 – 86.

[2] M.A. Bender and M. Farach-Colton, The LCA problem revisited, in *Proc. of LATIN 2000*, Lecture Notes in Comput. Sci. **1776**, Springer, 2000, 88 - 94.

[3] O. Berkman and U. Vishkin, Recursive star-tree parallel data structure, *SIAM J. Comput. 22* (1993) 221 – 242.

[4] N.G. de Bruijn, A combinatorial problem, *Nederl. Akad. Wetensch. Proc.* **49** (1946) 758 – 764.

[5] J. Fischer and V. Heun, Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE, in: M. Lewenstein and G. Valiente (Eds.), *Proc. of CPM 2006*, Lecture Notes in Comput. Sci. **4009**, Springer-Verlag Berlin Heidelberg 2006, 36 - 48.

[6] D. Gusfield, *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*, Cambridge Univ. Press, 1997.

[7] D. Gusfield and J. Stoye, Linear time algorithm for finding and representing all tandem repeats in a string, *J. Comput. Syst. Sci.* **69** (2004) 525 - 546.

[8] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* **13** (1984) 338 - 355.

[9] L. Ilie and L. Tinta, Practical algorithms for the longest common extension problem, in: J. Karlgren, J. Tarhio, and H. Hyyr (eds.), *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE'09)*, Lecture Notes in Comput. Sci. **5721**, Springer, Heidelberg, 2009, 302 – 309.

[10] J. Kärkkäinen and P. Sanders, Simple linear work suffix array construction, in *Proc. of ICALP'03*, Lecture Notes in Comput. Sci. **2719**, Springer-Verlag, Berlin, Heidelberg, 2003, 943 - 955.

[11] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, *Proc. of CPM'01*, Lecture Notes in Comput. Sci. **2089**, Springer-Verlag, Berlin, 2001, 181 - 192.

[12] D.K. Kim, J.S. Sim, H. Park, and K. Park, Constructing suffix arrays in linear time. *J. Discrete Algorithms* **3**(2-4) (2005) 126 - 142.

[13] P. Ko and S. Aluru, Space efficient linear time construction of suffix arrays, *J. Discrete Algorithms* **3**(2-4) (2005) 143 - 156.

[14] G. Landau, J.P. Schmidt, and D. Sokol, An algorithm for approximate tandem repeats, *J. Comput. Biol.* **8** (2001) 1 - 18.

[15] G. Landau and U. Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in *Proc. of STOC* , ACM Press, 1986, 220 - 230.

[16] G. Landau and U. Vishkin, Fast parallel and serial approximate string matching, *J. Algorithms* **10** (1989) 157 - 169. (Preliminary version in ACM STOC 86.)

[17] M. Main and R.J. Lorentz, An O(n log n) algorithm for finding all repetitions in a string, *J. Algorithms* **5** (1984) 422 - 432.

[18] U. Manber and G. Myers. Suffix arrays: a new method for on-line search, *SIAM J. Comput.* **22**(5) (1993) 935 – 948.

[19] G. Manzini and P. Ferragina, Engineering a lightweight suffix array construction algorithm, *Algorithmica* **40**(1) (2004) 33 – 50.

[20] R. de C. Miranda and M. Ayala-Rincon, A Modification of the Landau-Vishkin algorithm computing longest common extensions via suffix arrays, *Porc. of BSB'05*, Lecture Notes in Comput. Sci. **3594**, Springer, Berlin, 2005, 1611-3349.

[21] G. Myers, An O(nd) difference algorithm and its variations, *Algorithmica* **1** (1986) 251 - 266.

[22] G.Navarro, A guided tour to approximate string matching, *ACM Computing Surveys* **33**(1) (2001) 31 – 88.

[23] B. Schieber and U. Vishkin, On finding lowest common ancestors: Simplification and parallelization, *SIAM J. Comput.* **17** (1988) 1253 - 1262.

[24] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control* **64** (1985) 100 - 118. (Preliminary version in Proceedings of the International Conference Foundations of Computation Theory, LNCS, vol. 158, 1983).

[25] E. Ukkonen, Finding approximate patterns in strings, *J. Algorithms* **6** (1985) 132 – 137.