Modifications of the Landau-Vishkin Algorithm Computing Longest Common Extensions via Suffix Arrays and Efficient RMQ computations

Rodrigo César de Castro Miranda[1], Mauricio Ayala-Rincón[1], and Leon Solon[1]

Instituto de Ciências Exatas,Universidade de Brasília
rodrigo.miranda@acm.org,ayala@mat.unb.br,leonsolon@gmail.com

Abstract. Approximate string matching is an important problem in Computer Science. The standard solution for this problem is an $O(mn)$ running time and space dynamic programming algorithm for two strings of length m and n. Landau and Vishkin developed an algorithm which uses suffix trees for accelerating the computation along the dynamic programming table and reaching space and running time in $O(nk)$, where $n > m$ and k is the maximum number of admissible differences. In the Landau and Vishkin algorithm suffix trees are used for pre-processing the sequences allowing an $O(1)$ running time computation of the longest common extensions between substrings. We present two $O(n)$ space variations of the Landau and Vishkin algorithm that use range-minimum-queries (RMQ) over the longest common prefix array of an extended suffix array instead of suffix trees for computing the longest common extensions: one which computes the RMQ with a lowest common ancestor query over a Cartesian tree and other that applies the succinct RMQ computation developed by Fischer and Heun.

1 Introduction

Matching strings with errors is an important problem in Computer Science, with applications that range from word processing to text databases and biological sequence alignment. The standard algorithm for approximate string matching is a dynamic programming algorithm with $O(mn)$ running-time and space complexity, for strings of size m and n.

Landau and Vishkin [14] developed an $O(kn)$ algorithm for matching a pattern to a string of length n with at most k differences. The algorithm iterates through the diagonals of the dynamic programming table and uses a suffix tree data structure for constant-time jumps along the diagonals, bypassing character-by-character matching. Despite being of theoretical interest, this algorithm has a large space usage.

In this paper we present two variations of the Landau-Vishkin algorithm which instead of suffix trees use suffix arrays as originally proposed in [18]. Suffix arrays enhanced with a table of longest common prefixes [1,15] are used for computing the necessary jumps along the dynamic programming table's diagonals. These variations use less space, therefore resulting in a method which is closer to practical usage for matching long sequences.

Section 2 defines the problem and presents the dynamic programming solution and section 3 presents the Landau-Vishkin algorithm, suffix trees and their use in the algorithm. Afterwards, section 4 presents suffix arrays and describes how they could be used instead of suffix trees and give a first version of a modified Landau-Vishkin algorithm. Section 5 shows how the modified algorithm can be further improved by the Fischer-Heun succinct RMQ algorithm. Finally, before concluding in section 7, section 6 presents comparative performance of Landau-Vishkin algorithm and the two proposed improvements.

2 Problem Definition

In this section we will define the problem being studied and present the standard dynamic programming solution.

2.1 Preliminaries

Given the strings $T = t_1...t_n$ and $P = p_1...p_m$ of length $|T| = n$ and $|P| = m$, $m \leq n$, over an alphabet Σ we present a few definitions.

- ε is the empty string.
- P is a *substring* of T if $m \leq n$ and $p_1...p_m = t_i...t_{i+m-1}$ for some $i \geq 1$ and $i + m - 1 \leq n$. If $m < n$ we say that P is a *proper substring* of T.
- P is a *prefix* of T if $m \leq n$ and $p_i = t_i$ for $1 \leq i \leq m$. If $m < n$ then we say that P is a *proper prefix* of T.
- P is a *suffix* of T if $p_1...p_m = t_i...t_{i+m-1}$ for $i + m - 1 = n$ and $i \geq 1$. If $i > 1$ then we say that P is a *proper suffix* of T. We also say that $T_i = t_i...t_n$ where $i \geq 1$ is the *i-th suffix* of T (that is, the suffix of T that starts at position i).
- The *longest common prefix* (LCP) of T and P is the longest string $L = l_1...l_k$ such that $0 \leq k \leq m$ and $l_1 \ldots l_k = p_1 \ldots p_k = t_1 \ldots t_k$. If $k = 0$ then $L = \varepsilon$. The notation $LCP_{P,T}(i, j)$ denotes the LCP of P_i and T_j. If P and T are clear from the context, we shall use simply $LCP(i, j)$.
- The *longest common extension* (LCE) of T and P at position (i, j) is the length of the LCP of T_i and P_j.The notation $LCE_{P,T}(i, j)$ denotes the LCE of P_i and T_j. If P and T are clear from the context, we shall use simply $LCE(i, j)$.

In this article we call the string T the *text* and the string P the *pattern*.

2.2 Approximate String Matching

Definition 1 (Edit distance).
The edit distance between two strings $P = p_1...p_m$ and $T = t_1...t_n$ is the minimum amount of operations needed to transform P into T or T into P, where the allowed operations are defined as follows.

- *Substitution* when a character p_i of P is replaced with a character t_j of T.
- *Insertion* when a character p_i of P is inserted at position j of T.
- *deletion* when a character p_i is removed from P.

The sequence of operations needed to transform P into T is called the *edit transcript* of P into T.

An *alignment* of P and T is a representation of the operations applied on P and T, usually placing one string on top of the other, and filling with a dash ('-') the positions in P and T where a space was inserted so that every character or space on either string is opposite a unique character or unique space on P or T [7].

Definition 2 (Approximate string matching with k differences).
The approximate string matching problem with k differences between a pattern P and a text T is the problem of finding every pair of positions (i, j) in T where the edit distance between P and $t_i...t_j$ is at most k.

2.3 Dynamic Programming Solution

We can find the edit distance $D(i,j)$ between $p_1...p_i$ and $t_1...t_j$ from the distances $D(i-1,j-1)$ between $p_1...p_{i-1}$ and $t_1...t_{j-1}$, $D(i-1,j)$ between $p_1...p_{i-1}$ and $t_1...t_j$ and $D(i,j-1)$ between $p_1...p_i$ and $t_1...t_{j-1}$ by solving the following recurrence relation.

$$D(i,j) = \begin{cases} i+j & \text{if } j=0 \text{ or } i=0, \\ min[D(i-1,j-1)+d, D(i-1,j)+1, D(i,j-1)+1] & \text{otherwise} \\ \text{where } d=0 \text{ if } p_i = t_j \text{ or } 1 \text{ if } p_i \neq t_j & \end{cases}$$

This relation can be calculated by a straightforward $O(nm)$ dynamic programming algorithm using an $n+1 \times m+1$ dynamic programming table. A technique due to Hirschberg [7,8] can be applied to it in order to decrease the space usage to $O(n)$ at the cost of doubling the computation time.

3 The Landau-Vishkin Algorithm

Landau and Vishkin presented an $O(kn)$ algorithm for the approximate string matching problem with k-differences in [14]. The algorithm is divided in two phases: a pre-processing phase and an iteration phase. In the pre-processing phase the pattern and the text are pre-processed for a $O(1)$ LCE computation. In the iteration phase the he algorithm iterates k times over each diagonal of the dynamic programming table and finds all matches of P with at most k differences. The explanation given follows the one given by Gusfield in [7].

3.1 Diagonals and d-paths

Given the text $T = t_1...t_n$ and the pattern $P = p_1...p_m$, and the dynamic programming matrix D, we present a few definitions.

- A *diagonal* d of D consists of all $D_{i,j}$ such that $j-i=d$.
- The *main diagonal* is the diagonal 0 of D composed by the cells $D_{i,i}$ where $0 \leq i \leq m \leq n$.
- A *path* in D is a sequence of adjacent cells.
- If a cell (i,j) follows a cell $(i-1,j-1)$ in a path, it is said to be a *mismatch* if $t_j \neq p_i$, and a *match* otherwise.
- If a cell $(i+1,j)$ or a cell $(i,j+1)$ follows a cell (i,j) in a path, it is said to be a *space* .
- A *error* in a path is either a mismatch or a space.
- A *d-path* in D is a path which starts either at column 0 before row $d+1$ or at row 0 and has the following properties:
  - Paths initiating at row 0 start with zero errors and paths initiating at cell $(i,0)$ for $1 \leq i \leq d$ start with i errors.
  - From any cell (i,j), the next cell along the path (if any) can only be one of $(i+1,j+1)$, $(i,j+1)$, or $(i+1,j)$.
  - It has a total of d errors.
- A d-path is *farthest-reaching* on diagonal i if it is a d-path that ends at cell (r,c) in i and the index of its ending column c is greater than or equal the index of the ending column of every other d-path which ends on i.

3.2 d-path Construction and Extension

A d-path is constructed in D in the following way.

If $d = 0$ then a 0-path that starts on diagonal i is a path that begins at cell $D_{0,i}$ and is extended along i to cell $D_{j,i+j}$, where j is the $LCE_{P,T}(1, i+1)$.

A d-path starting at cell $(d, 0)$ for $0 < r \leq m$ is extended in the same way as a 0-path from row 0.

If $d > 0$ then a d-path is constructed from a $(d-1)$-path whose ending cell $D_{r,c}$ is on diagonal i by firstly extending it to cell $D_{r',c'}$ in diagonal i' in one of three ways:

- the path is extended one cell to the right to cell $D_{r',c'} = D_{r,c+1}$ on diagonal $i' = i+1$, meaning a space is inserted in the pattern at position r;
- the path is extended one cell down to cell $D_{r',c'} = D_{r+1,c}$ on diagonal $i' = i-1$, meaning a space is inserted in the text at position c;
- the path is extended one cell along the diagonal $i' = i$ to cell $D_{r',c'} = D_{r+1,c+1}$, meaning a mismatch between t_c and p_r.

Secondly, after extending a $(d-1)$-path to cell (r', c') on diagonal i', the path is further extended l cells along the diagonal i' where l is the $LCE_{P,T}(r', c')$.

3.3 The algorithm

The Landau-Vishkin algorithm, presented in table 1, iterates over every diagonal i of the dynamic programming table D, building d-paths that are farthest-reaching on each diagonal, beginning with all 0-paths, and then from those all 1-paths and so forth until every k-paths have been found. Those k'-paths (where $0 \leq k' \leq k$) that reach row m of D are matches of P in T with at most k differences. We have omitted the special treatment of d-paths starting at column zero in for simplicity (see algorithm 1).

Algorithm 1 Landau and Vishkin approximate string matching algorithm

```
1. Pre-process T and P so that we can answer LCE queries in O(1)
   1.1 Build a generalized suffix tree 𝒯 for P and T.
   1.2 Pre-process 𝒯 so that we can answer LCA queries in O(1).
2. For every diagonal i of the dynamic programming table, find
   its farthest reaching 0-path with an O(1) LCE lookup between P and T_{i+1}
3. For d = 1 to k:
   3.1 For every diagonal i of the dynamic programming table:
      3.1.1 Extend the farthest reaching (d − 1)-path on diagonal i − 1
            one cell to the right so that it reaches diagonal i on cell (r, s).
      3.1.2 Further extend it along i by a number of cells equal
            to the LCE of the corresponding suffixes of P and T
            3.1.2.1 The LCE is given by the depth of the LCA of the
                 corresponding suffixes of P and T on 𝒯.
      3.1.3 Extend the (d − 1)-paths on diagonals i + 1 and i in a similar way
      3.1.4 Choose the farthest reaching among the three extended paths
3. Each path that reaches row m is a match of P in T with at most k errors.
```

For each iteration we find the farthest reaching d-path on diagonal i in the following way.

- If $d = 0$ then the 0-path that starts at diagonal i is the farthest-reaching 0-path on i.

- If $d > 0$, we can find the farthest reaching d-path from the farthest-reaching $(d-1)$-paths on diagonals $i-1$, i and $i+1$ as follows.
  - We extend the farthest reaching $(d-1)$-path on diagonal $i-1$ one cell to the right to diagonal i and then further extend the path along i as described in section 3.2. In a similar way we extend the farthest reaching $(d-1)$-path on diagonal $i+1$ one cell down, and on diagonal i one cell along the diagonal i.
  - The farthest reaching d-path on i is chosen from the three paths above as being the one that is the farthest-reaching.

In its pre-processing phase the Landau-Vishkin algorithm builds a generalized suffix tree $\mathcal{T}$ (see section 3.4) for P and T, which means that every suffix of T and every suffix of P have corresponding leaves in $\mathcal{T}$. The tree is further pre-processed to allow for $O(1)$ lowest common ancestor (LCA) queries (see section 3.5), which enables us to find the LCE of any two suffixes of P and T in $O(1)$.

Since the body of the algorithm's inner loop runs in $O(1)$ and the pre-processing function runs in $O(n)$, then the whole algorithm runs in time $O(kn)$.

3.4 Suffix Trees

A suffix tree $\mathcal{T}$ for a string $T = t_1...t_n$ over an alphabet Σ is a rooted tree that has the following properties:

- there are exactly n leaves, numbered 1 to n;
- every internal node of the suffix tree, except for the root, has at least two outgoing edges;
- every edge of $\mathcal{T}$ is labeled by a substring of T, so that any two labels of all edges that start at a node v differ at least in their first characters;
- for every leaf i of $\mathcal{T}$, the concatenation of the labels of the edges on the path from the root to i gives us the suffix T_i of T.

A sentinel character which does not belong to Σ (in this paper we use both \$ and #, $\$ \neq \#$) is usually concatenated to T to guarantee that its suffix tree has exactly n leaves. A generalized suffix tree for T and P is the suffix tree for $T\#P$.

The suffix tree for a string of length n can be constructed in time $O(n)$ using $O(n)$ space as shown by McCreight [17] and Ukkonnen [20].Given the suffix tree $\mathcal{T}$ for the string $T = t_1...t_n$, querying if a pattern $P = p_1...p_m$ matches a substring of T takes at most $O(m)$ comparisons, independent of the length n of T.

3.5 Lowest Common Ancestor

The Landau-Vishkin algorithm uses a lowest common ancestor computation for finding the longest common extension of suffixes of the pattern and the text.

Definition 3 (Ancestor node and lowest common ancestor). *Given two nodes v and w of a rooted tree $\mathcal{T}$, we say that a node v is an* ancestor *of a node w if v is on the unique path from the root to w. A node v is an ancestor of itself. We say that v is a* proper ancestor *of w if v is not w. The* **Lowest Common Ancestor** *(LCA) of two nodes x and y is the deepest node in $\mathcal{T}$ that is an ancestor of both x and y.*

In a suffix tree $\mathcal{T}$ for the string T, given any two leaves i and j of $\mathcal{T}$, corresponding to the suffixes T_i and T_j of T, the LCA of i and j gives us the $LCP_{T,T}(i,j)$ and its depth gives us the $LCE_{T,T}(i,j)$.

Gusfield [7] describes in detail a $O(1)$ LCA query over a suffix tree after $O(n)$ pre-processing. The pre-processing also uses $O(N)$ extra space, where N is the number of nodes in the tree.

4 Modification of the Landau-Vishkin Algorithm

Instead suffix trees on Landau and Viskin's algorithm one can use of suffix arrays for computing the LCE as originally proposed in [18]. Such modification will enable us to use more compact data structures than even optimized implementations of suffix trees.

4.1 Suffix Arrays

Definition 4 (Suffix array). *A suffix array Pos for a string T is an array which gives us a lexicographically ordered sequence of suffixes of T.*

For the construction of the suffix array Pos, the alphabet Σ must be ordered. The sentinel character $ is commonly used to assure the proper sorting order.

Since it is an array of indexes of positions in T, a suffix array uses space $O(n)$, as the suffix tree, but the multiplying factor of n for the actual size is smaller than on suffix trees ($4n$ if indexes of 32-bits are used, $8n$ if 64-bits are used).

An $O(n)$ space and $O(n)$ expected running time and $O(n \log_2 n)$ worst-case running time direct suffix array construction algorithm is presented in [15]. Recently $O(n)$ worst-case direct construction algorithms have been published in [12], [9], and [11].

An *enhanced suffix array* is a suffix array augmented with a table of longest common prefixes, also called an LCP table [1] [15] [9]. Given the suffix array Pos for the string $T = t_1...t_n$, the LCP table is the array lcp of n elements where $lcp[i]$ is the length of the LCP of $Pos[i]$ and $Pos[i+1]$. The lcp array can be constructed in linear time from Pos [10].

4.2 Longest Common Extension Computation on a Suffix Array

What enables the Landau-Vishkin algorithm to achieve its $O(kn)$ space and running time bounds is the constant time LCE computation, which is done using a $O(1)$ LCA query over a suffix tree.

Given an enhanced suffix array Pos for the string $P\#T\$$, we can pre-process its corresponding lcp array and answer LCE queries in constant time. The key to such an operation is the following theorem.

Theorem 1. *The LCE between two suffixes S_a and S_b of S can be obtained from the lcp array in the following way.*

Let i be the rank of S_a among the suffixes of S (that is, $Pos[i] = a$). Let j be the rank of S_b among the suffixes of S. Without loss of generality, we assume that $i < j$. Then the $LCE(a, b)$ of S_a and S_b is $lcp(i, j) = min_{i \leq k < j} lcp[k]$.

Proof. Let $S_a = s_a...s_{a+c}...s_n$ and $S_b = s_b...s_{b+c}...s_n$, and let c be the $LCE(a, b)$. We assume that S has a sentinel character.

If $i = j - 1$ then $k = i$ and $lcp[i] = c$ is the $LCE(a, b)$ and we are done.

If $i < j - 1$ then select k such $lcp[k]$ is the minimum value in the interval $[i, j]$ of the lcp array. We then have two possible cases:

- If $c < lcp[k]$ we have a contradiction because $s_a \dots s_{a+lcp[k]-1} = s_b \dots s_{b+lcp[k]-1}$ by the definition of the LCP table, and the fact that the entries of lcp correspond to sorted suffixes of S.
- if $c > lcp[k]$, let $j = Pos[k]$, so that S_j is the suffix associated with position k. S_k is such that $s_j \dots s_{j+lcp[k]-1} = s_a \dots s_{a+lcp[k]-1}$ and $s_j \dots s_{j+lcp[k]-1} = s_b \dots s_{b+lcp[k]-1}$, but since $s_a \dots s_{a+c-1} = s_b \dots s_{b+c-1}$ we have that the lcp array should be wrongly sorted which is a contradiction.

Therefore we have $c = lcp[k]$

Thus we have reduced our LCE query to a *range minimum query* (RMQ) over a range in *lcp*. We can pre-process an array of integers (such as *lcp*) in linear time so that a RMQ in a given interval of the array is answered in $O(1)$. Kärkkäinen and Sanders give such an algorithm in [9], as does Farach-Colton and Bender in [3] and Kim et ali. in [11]. Fischer and Heun developed a direct-construction RMQ algorithm.

We will first present an algorithm based on Cartesian trees given in [6] by Gabow, Bentley and Tarjan.

We will build in $O(n)$ a Cartesian tree for the *lcp* array, which will enable us to query the minimum value of any range in *lcp* in $O(1)$ by doing a $O(1)$ LCA query.

Definition 5 (Cartesian Trees).
A Cartesian tree $\mathcal{C}$*for a sequence of real numbers* $x_1 \ldots x_n$ *is a binary tree with nodes labeled by those numbers, such that the root is labeled by* m *where* $x_m = \min x_i \mid 1 \leq i \leq n$*, the left subtree is the Cartesian tree for* $x_1 ... x_{m-1}$ *and its right subtree is the Cartesian tree for* $x_{m+1} \ldots x_n$*.*

Proposition 1. *The smallest number of the interval* $x_i \ldots x_j$ *can then be found by simply finding the* LCA *of nodes* i *and* j *on* $\mathcal{C}$*.*

Proof. Given the nodes i and j, and v the LCA of i and j, and suppose that $i < j$. The structure of the Cartesian $\mathcal{C}$ tree as given above is such that if a node v is the LCA of nodes i and j, then it means that $i \leq v \leq j$, because from the construction of $\mathcal{C}$, every other ancestor of v is either to the right of i and j, or to the left of them. Furthermore, from the construction of $\mathcal{C}$, the node v is the node such that x_v is the smallest value from its subtree, and thus finding $v = LCA(i, j)$ we also find the smallest value x_v in the range $x_i \ldots x_j$.

As we have remarked in section 3.5, given a tree with n nodes we can pre-process it in $O(n)$ time and space so that we can answer LCA queries over it in $O(1)$.

In order to compute the Cartesian tree in $O(n)$. It is done using the algorithm given in [6] and [3].

Thus, in order to answer LCE queries in $O(1)$ with $O(n)$ pre-processing, we do as follows.

- Build a suffix array Pos in $O(n)$ time and space for the text concatenated with the pattern (with sentinel characters), and build the LCP table for Pos in $O(n)$ as given in [10]).
- Create a Cartesian tree $\mathcal{C}$ for the LCP table
- Pre-process $\mathcal{C}$ in $O(n)$ so that we can query the LCA of any two nodes of $\mathcal{C}$ in $O(1)$.

Given the suffixes i and j of the concatenated string, $LCE(i, j)$ will be the result of the RMQ over $lcp_i \ldots lcp_j$, which is given by a $O(1)$ LCA query over $\mathcal{C}$.

4.3 Modified Algorithm

The modified algorithm is then the same Landau-Vishkin algorithm, substituting the suffix tree for a suffix array, and the LCA query over the suffix tree for a RMQ over the LCP table for the suffix array. For our first version, we use a LCA query over Cartesian tree for the RMQ and thus the only changes to the original algorithm are step 1 (LCE pre-processing) and 3.1.2 (LCE calculation) which we present as algorithms 2 and 3.

Although it also uses a LCA query over a tree, it is a smaller tree, with exactly n nodes, and which can be implemented using less space than a suffix tree.

Algorithm 2 Modified Landau-Vishkin LCE Pre-processing

```
1. Pre-process T and P so that we can answer LCE queries in O(1)
   1.1 Build a suffix array Pos for the string P#T, together with the lcp table
   1.2. Build a Cartesian tree C for the lcp table
   1.3. Process the Cartesian tree C so that we can do O(1) LCA queries
      on it.
```

Algorithm 3 Modified Landau-Vishkin LCE Calculation

```
      3.1.2 Further extend it along i by a number of cells equal
            to the LCE of the corresponding suffixes of P and T
            3.1.2.1 The LCE is given by the RMQ of the lcp array in the
                 range corresponding to the relevant suffixes of P and T.
                 The RMQ is given by a LCA query on the Cartesian tree.
```

Theorem 2 (Running time and space complexity). *The modified Landau-Vishkin approximate string matching algorithm runs in time $O(nk)$.*

Proof. As commented before, construction and maintenance of suffix arrays run in $O(n)$ time and space ([13]) and it holds for building and maintenance of Cartesian trees. Since with $O(n)$ pre-processing LCA queries are done in $O(1)$, the construction of the approximate matches is computed in $O(kn)$ running time.

Although the theoretical bounds coincide with the original ones of the Landau-Vishkin algorithm, this variation uses less space during pre-processing than with suffix trees. Suffix arrays are a more compact data structure, and the LCA pre-processing for the Cartesian tree also uses less space than the same pre-processing over suffix trees, as it has exactly n nodes, while with suffix trees it ranges from $n+1$ to $2n-1$ nodes. In [18] we have shown it to be on the average $6n$ bytes.

5 Further improvements: computing LCE via RMQ Fischer-Heun's method

In [5] Fischer and Heun developed a new representation for solving the RMQ problem for an array $N = [N_0, N_1, \ldots, N_n]$ of n integers in time $O(1)$ after $O(n)$ pre-processing.

The main idea of Fischer-Heun's method is to split a RMQ query in five $O(1)$ sub-queries using intermediate tables for storing all RMQ query results in a pre-processing phase. The integer array is split into blocks and in superblocks. It is divided into a set of n/s' superblocks, $s' = \log^{2+\epsilon} n$, where $\epsilon > 0$ is a constant. Each superblock B'_i represents an array range of size s', from $N[(i-1)s']$ to $N[is'-1]$. The array is also split into n/s blocks, $s = \log n/(2+\delta)$, where $\delta > 0$ is a constant. Each block B_i represents a range of the array N with size s, from element $N[(i-1)s]$ to $N[is-1]$. The division of the array into superblocks and into blocks is presented in figure 1.

Each $RMQ(l, r)$ query, thus, is split into five other queries of three different types, as show in figure 1: a superblock-query, a query that surpasses a superblock size limit; two block queries, one at the left and one at the right of the superblocks of the previous query; and two in-block-queries, one at the left and one at the right of the blocks of the previous queries.

The array preprocessing is done for long queries and short queries. Long queries are superblock and block queries. The RMQs of long queries are preprocessed using dynamic programming as in Sadakane's method [19]. Superblock queries are stored into

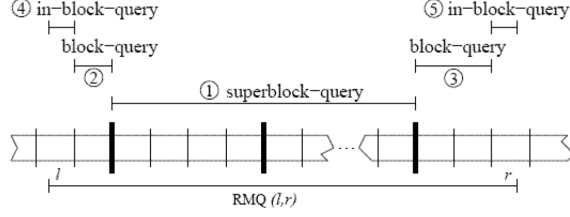

Fig. 1. Division of the array into blocks and superblocks. RMQ(l,r) query decomposed into other five queries [5]

a $n/s' \times \log(n/s')$ table, M', and block queries are stored into a $n/s \times \log(s/s')$ table, M.

Short queries, on the other hand, are inner-block queries and they are preprocessed using the *Four Russians Method* as shown in [4] combined with the Alstrup et al method [2]. An important observation is that an for array of size s there are at most a number of C_s query possibilities, C_s being the sth Catalan number. Thus, instead of storing all RMQs of each block (for inner block queries), no more than C_s RMQs need to be stored in a table P.

To complete the Fischer-Heun's RMQ representation we must index the P table. Not all N blocks RMQs are stored in P, so we need an array T which stores all types of blocks in order to group them into equivalence sets (groups). In this way, several blocks would be part of the same group in case they share the same RMQ or the same RMQ position.

Therefore a $RMQ(l, r)$ query five time $O(1)$ queries into three tables and one array: one M' query; two M queries; and two P queries. The space usage of Fischer-Heun's RMQ representation, more specifically tables M', M, P and array T, is as follows:
- M occupies $n/\log^{2+\epsilon} n \times \log(n/log^{2+\epsilon}n) \cdot \log n$ bits, $= o(n)$ bits;
- M' uses $(2+\delta)n/\log n \times \log((2+\delta)\log^{1+\epsilon} n) = o(n)$ bits;
- T is of size $n/s = (2+\delta)n/\log n$, each position of the array occupies at most $4^s/s^{3/2}$. Thus, T occupies $\frac{n}{s} \cdot \log(O(4^s/s^{3/2})) = \frac{n}{s}(2s - O(\log s)) = 2n - O(n/\log n \log\log n) = 2n - o(n)$ bits;
- P occupies $O(\frac{4^s}{s^{3/2}} s \cdot s) = O(n^{2/(2+\delta)}\sqrt{\log n}) = o(n/\log n)$ bits.

The total space used by Fischer-Heun's RMQ representation is $M' + M + T + P = 2 + O(n)$ bits, which is much better than the $8n$ plus LCA preprocessing for the Cartesian tree's method.

The modifications to Landau-Vishkin's pre-processing and LCE computationalgorithm to calculate LCE via suffix arrays using Fischer-Heun's RMQ representation are shown shown in algorithms 4 and 5.

Algorithm 4 Second Modified Landau-Vishkin LCE Pre-processing (F-H's RMQ)

1. Pre-process T and P so that we can answer LCE queries in $O(1)$
   1.1 Build a suffix array Pos for the string $P\#T$, together with the lcp table
   1.2 Preprocess lcp array and precompute all RMQs according to
      Fischer-Heun's method in order to answer LCE in time $O(1)$

Algorithm 5 Second Modified Landau-Vishkin LCE Calculation (F-H's RMQ)

```
3.1.2 Further extend it along i by a number of cells equal to the LCE
      of the corresponding suffixes of P and T
      3.1.2.1 The LCE is given by a RMQ lookup on Fischer-Heun's succinct
              data structures.
```

Theorem 3 (Running time and space complexity of Fischer-Heun RMQ method). *The modified Landau-Vishkin approximate string matching algorithm using the Fischer-Heun RMQ method for LCE runs in time $O(nk)$.*

Proof. As commented before, construction and maintenance of suffix arrays run in $O(n)$ time and space ([13]) and it holds for building of the succinct RMQ structures [5]. Since with this pre-processing RMQ queries are done with 5 $O(1)$ queries, the construction of the approximate matches is computed in $O(kn)$ running time.

6 Comparison of the three implementations

In order to obtain some measured data of the three implementations we did some simple experiments. The experiments were run on a DELL PowerEdge 1800 computer with 2 GB of RAM and a 3.0 GHz Intel Xeon processor with 2MB L2 cache and a 800Mhz FSB. The operating system was Ubuntu Linux 7.10 for 32-bit processors, running the kernel 2.6.22-14. The compiler was GNU GCC version 4.1.3. The suffix tree implementation was that of Kurtz [13] for the software Mummer 3.0, and the suffix arrays were constructed from the algorithm and implementation due to Manzini and Ferragina [16]. Fischer-Heun RMQ code was the C++ code from Fischer himself. The rest of the implementation was developed by the authors with some *LCA* code from the *strmat* library.

6.1 Random data

First we ran the program with random data, with $|\Sigma| = 4$ and $|\Sigma| = 20$, from 20M characters and up, in intervals of 5M characters until each algorithm started using more memory than was available and running times became prohibitive.

Space usage for random data is detailed on table 1, where. , n is the length of the text, **Mem** the program's peak memory usage and **Bpc** the average bytes per character. Running time in seconds for the same data is detailed on table 2. Whenever a '-' appears on the table it means that the program was not successfull for such ammount of data.

Table 1. Space usage for for random data, $m = 1000$, $k = 20$

	Suffix Tree				**Cart. RMQ**		**Fischer RMQ**	
	$\|\Sigma\| = 4$		$\|\Sigma\| = 20$					
n(MB)	Mem (MB)	Bpc	Mem (MB)	Bpc	Mem (MB)	Bpc	Mem (MB)	Bpc
20	1830	91	1594	79	1360	67	975	48
25	2287	91	1985	79	1700	67	1219	48
30	-	-	-	-	2040	67	1463	48
35	-	-	-	-	2380	67	1707	48
40	-	-	-	-	-	-	1951	48

Table 2. Running time (s) for random data $m = 1000$, $k = 20$

	$\|\Sigma\| = 4$			$\|\Sigma\| = 20$		
n(MB)	Suffix Tree	Cart. RMQ	Fischer RMQ	Suffix Tree	Cart. RMQ	Fischer RMQ
20	163	185	248	122	80	86
25	305	237	315	157	102	109
30	-	424	382	-	243	132
35	-	668	449	-	412	155
40	-	-	521	-	-	180

From the table 1 the memory usage of the Fischer-Heun RMQ-based algorithm is much smaller, enabling it to process a larger ammount of data without incurring in performance problems due to virtual memory paging. Furthermore, from table 2 we note that the running time of all three algorithms is similar when the data fits whole in memory, with a small performance advantage for the suffix tree based algorithms over the suffix array based algorithms for small alphabets, and a slight advantage for the Cartesian Tree-based method over the Fischer-Heun RMQ method.

6.2 Real data

In order to observe the behaviour of the three methods with real data we matched a 100-character pattern against the full Uniprot/Swissprot database coded as text file (one protein sequence per line), which we split it into different files which contained each as much lines as each algorithm could handle at once. The results are in table 3.

Table 3. Matching a sequence to Uniprot/Swissprot - $|\Sigma| = 20$, $m = 100$, $k = 10$

Algorithm	n(MB)	Files	Mem (MB)	Bpc	T (s)
Fischer RMQ	130	4	6319	49	296
Cart. RMQ	130	5	8810	68	590
Suffix Tree	130	6	11511	89	590

The memory usage patterns were within expected parameters. We also verified that the performance of all algorithms are similar if the whole data can be fit in RAM, however the more space-economical approach of the Fischer RMQ method enables one to process a lot more data in a single run, breaking the actual processing into less steps at a time.

7 Concluding Remarks

We have found the Landau and Vishkin algorithm it to be a reasonable testbed for alternate pattern matching techniques, enabling us to exchange data structures and find similarities and trade-offs, specially as the datasets grow to very large sizes.

We have shown that it is possible to change the a suffix-tree based string matching algorithm to use suffix arrays instead for its computation of longest common extensions between suffixes of the text and the pattern, while keeping the same running time and

space complexity. Furthermore the change in the data structures resulted in a large economy of space usage, about 27% compared to the Cartesian tree-based RMQ and 45% if compared to the suffix-tree version. This economy of space enabled us to process strings that would require some manipulation with the other methods.

References

1. M. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *9th International Symposium on String Processing and Information Retrieval*, pages 31–43, 2002.
2. S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *SPAA '02: Proc. of the* 14th *annual ACM symposium on Parallel algorithms and architectures*, pages 258–264, New York, NY, USA, 2002. ACM.
3. M. Bender and M. Farach-Colton. The LCA Problem Revisited. In *LATIN 2000*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer Verlag, 2000.
4. J. Fischer and V. Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In *CPM*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer Verlag, 2006.
5. J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer Verlag, 2007.
6. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In 16th *ACM STOC*, pages 135–143, 1984.
7. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
8. D. Hirschberg. A linear space algorithm for computing the maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975.
9. J. Kärkkäinen and P. Sanders. Simpler linear work suffix array construction. In *Int. Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer Verlag, 2003.
10. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In 12th *Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer Verlag, 2001.
11. D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer Verlag, 2003.
12. P. Ko and S. Aluru. Space-Efficient Linear Time Construction of Suffix Arrays. *Journal of Discrete Algorithms*, 3:143–156, 2005.
13. S. Kurtz. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.*, 29(13):1149–1171, 1999.
14. G. Landau and U. Vishkin. Introducing Efficient Parallelism into Approximate String Matching and a new Serial Algorithm. In 18th *ACM STOC*, pages 220–230, 1986.
15. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. Technical Report TR 89-14, University of Arizona, 1989.
16. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 23(40):33–50, 2004.
17. E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, April 1976.
18. R. de C. Miranda and M. Ayala-Rincón. A Modification of the Landau-Vishkin Algorithm Computing Longest Common Extensions via Suffix Arrays. In *BSB*, volume 3594 of *Lecture Notes in Computer Science*, pages 210–213. Springer Verlag, 2005.
19. K. Sadakane. Space-efficient data structures for flexible text retrieval systems. In *Algorithms and Computation*, volume 2518 of *Lecture Notes in Computer Science*, pages 14–24, 2002.
20. E. Ukkonen. On-line Construction of Suffix-Trees. *Algorithmica*, 14:249–260, 1995.