# A Modification of the Landau-Vishkin Algorithm Computing Longest Common Extensions via Suffix Arrays

Rodrigo de Castro Miranda[1⋆] and Mauricio Ayala-Rincón[1]

Mestrado em Informática e Departamento de Matemática, Universidade de Brasília, Brasil
rodrigo.miranda@acm.org,ayala@mat.unb.br

**Abstract.** Approximate string matching is an essential problem in many areas related to Computer Science including biological sequence processing. The standard solution of 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. Suffix trees are used for preprocessing the sequences allowing an $O(1)$ running time computation of the longest common extensions between substrings. One of the practical drawbacks of the Landau-Vishkin algorithm is the excessive use of space inherent to the use of suffix trees. In fact, although suffix trees can be handled in linear space their construction and manipulation implies high multiplying factors to this linear behavior. We present a variation of the Landau-Vishkin algorithm which instead of suffix trees uses suffix arrays for computing the longest common extensions, thereby improving actual space usage.

**Keywords.** Approximate string matching, suffix trees, suffix arrays.

## 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 [12] 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. This algorithm may be adapted to treat other problems of biological interest such as sequence alignment. However of theoretical interest,

⋆ Corresponding author

the Landau-Vishkin algorithm has the practical drawback that the use of suffix trees implies a multiplier of over ten bytes per input character on average [11], which makes it uninteresting for the treatment of long length sequences which occur in molecular biology.

In this paper we present a variation of the Landau-Vishkin algorithm which instead of suffix trees uses suffix arrays enhanced with a table of longest common prefixes [1, 13] for computing the necessary jumps along the diagonals of the dynamic programming table. Since in contrast with the space usage of suffix trees suffix arrays use much less space [15, 10], by this proposed modification the space usage of the Landau-Vishkin algorithm is decreased obtaining a method which is closer to practical usage for long sequences which arise in molecular processing.

Initially, section 2 defines the problem and presents the dynamic programming solution. Afterward, section 3 presents the Landau-Vishkin algorithm, suffix trees and their use in the algorithm and section 4 presents suffix arrays and describes how they could be used instead of suffix trees in the Landau-Vishkin algorithm. Finally, section 5 concludes and remarks on further work.

## 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 $\Sigma$ we present a few definitions.

- $\varepsilon$ 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* of $T$ and $P$ is the longest string $L = l_1...l_k$ such that $0 \leq k \leq m$ and $l_1 \dots l_k = p_1 \dots p_k = t_1 \dots t_k$. If $k = 0$ (i.e. $P$ and $T$ do not have a common prefix) then $L = \varepsilon$.
- The *longest common extension* of $T$ and $P$ at position $(i, j)$ is the length of the longest common prefix of $T_i$ and $P_j$.

### 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 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 $T$ [5].

**Definition 2 (Approximate string matching with $k$ differences).**
*We define the approximate string matching problem with $k$ differences between a pattern $P$ and a text $T$ to be 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 $k$. The special case when $k = 0$ is the problem of finding all occurrences of $P$ in $T$.*

### 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.

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

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 [6] [5] 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 [12]. The algorithm iterates $k$ times over each diagonal of the dynamic programming table and find all matches of $P$ that end at each diagonal with at most $k$ differences. The explanation given follows the one given by Gusfield in [5].

### 3.1 Diagonals and *d*-paths

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

- A *diagonal* $d$ of the dynamic programming matrix $D$ 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 the dynamic programming table 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 *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 (insertion of $i$ spaces before $T$ for the prefix $p_1...p_i$).
  - 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 $d$ mismatches and spaces.
- A $d$-path is *farthest-reaching* on diagonal $i$ if it is a $d$-path that ends in diagonal $i$ and the index of it's ending column $c$ along diagonal $i$ is greater than or equal the index of the ending column of every other $d$-path which ends on diagonal $i$.

### 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 diagonal $i$ to cell $D_{j,i+j}$, where $j$ is the length of the longest common prefix of $t_{i+1}...t_{i+j}$ and $p_1...p_j$. $d$-paths starting from column 0 should be extended in this way too.

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

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

Secondly, after extending a $(d-1)$-path to cell $(r', s')$ on diagonal $i'$, the path is further extended $l$ cells along the diagonal $i'$ where $l$ is the length of the longest common prefix of $T_{s'}$ and $P_{r'}$. Notice that $d$-paths starting in column zero need special treatment during the initialization steps.

### 3.3 The algorithm

The Landau-Vishkin algorithm, presented in table 1, iterates over every diagonal $i$ of the dynamic programming table, 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 the dynamic programming table are matches of $P$ in $T$ with at most $k$ differences.

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 diagonal $i$ 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 diagonal $i$ is chosen from the three paths above as being the one that has the greater index of its ending column.

In its pre-processing phase the Landau-Vishkin algorithm build 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 $P$ has a corresponding leaf in $\mathcal{T}$. The tree is further pre-processed to allow for $O(1)$ lowest common ancestor queries (see section 3.5), which enables us to find the longest common extension of any two suffixes of $P$ and $T$ in constant time. The special treatment of $d$-paths starting at column zero is omitted in the algorithm 1 for simplicity.

Since the body of the algorithm's inner loop runs in constant time and the pre-process function runs in linear time, then the whole algorithm runs in time $O(kn)$, which is better than $O(mn)$ for the standard dynamic programming algorithm for large enough values of $m$ (length of the pattern).

### 3.4 Suffix trees

A suffix tree $\mathcal{T}$ for a string $T = t_1...t_n$ over an alphabet $\Sigma$ 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 $T$.

**Algorithm 1** Landau and Vishkin approximate string matching algorithm

```
1. Build a generalized suffix tree 𝒯 for P and T.
2. Pre-process 𝒯 so that we can answer LCA queries in constant time.
3. For every diagonal i of the dynamic programming table, find
   its farthest reaching 0-path with an O(1) LCA lookup between P and
   the (i+1)-th suffix of T.
4. For d = 1 to k:
   4.1 For every diagonal i of the dynamic programming table:
      4.1.1 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).
      4.1.2 Further extend it along i by a number of cells equal
            to the depth of the LCA of the corresponding suffixes
            of P and T (that is, P[r]...P[m] and T[s]..T[n]).
      4.1.3 Extend the (d−1)-paths on diagonals i+1 and i in a
            similar way, as described in the text.
      4.1.4 Choose the farthest reaching among the three extended
            paths.
5. Every path that reaches row m is a match of P in T with at most
   k mismatches and spaces.
```

A sentinel character which does not belong to $\Sigma$ (here we use \$) is concatenated to $T$ to guarantee that its suffix tree has exactly any leaves as described above. A generalized suffix tree for the strings $A$ and $B$ is the suffix tree for the string $A\#B$, where $\#$ is a second sentinel character.

The suffix tree for a string of length $n$ can be constructed in time $O(n)$ using $O(n)$ space as shown by McCreight [14] and Ukkonnen [17]. Ukkonen's algorithm has the additional property that it is *on-line*. In fact, it builds the suffix tree incrementally from the prefixes of $T$, starting with the smallest non-empty prefix and proceeding until the tree for the complete string is built. 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 $T$.

Kurtz [11] remarks that although a suffix tree can be built using linear space complexity, most implementations actually have a large constant multiplying factor. For large enough strings, such space usage can be a limiting factor in the choice of this data structure.

### 3.5 Lowest Common Ancestor

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

**Definition 3 (Ancestor node).** *Given two nodes $v$ and $w$ of a 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 $w$.*

**Definition 4 (Lowest common ancestor).**
*In a rooted tree $\mathcal{T}$, 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 longest common prefix of $T_i$ and $T_j$.

Gusfield [5] describes in detail a constant-time LCA query over a suffix tree after linear time pre-processing. It is based on a much simpler LCA algorithm for complete binary trees. The pre-processing of $\mathcal{T}$ creates an implicit mapping to a complete binary tree $\mathcal{B}$ and allows queries for lowest common ancestors of any two nodes of $\mathcal{T}$ to be answered in $O(1)$. The pre-processing also uses $O(n)$ extra space.

## 4 Modification of the Landau-Vishkin algorithm

Our proposal is to substitute the use of suffix trees on Landau and Viskin's algorithm with the use of suffix arrays for computing the longest common prefixes. The advantage of this modification is that it will enable us to use a more compact data structures than suffix trees.

### 4.1 Suffix Arrays

**Definition 5 (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 $\Sigma$ must be ordered. The sentinel character \$ is commonly used to assure the proper sorting order and it has the special property that it is either greater or smaller than any symbol of $\Sigma$.

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 much smaller than on suffix trees ($4n$ if indexes of 32-bits are used, $8n$ if 64-bits are used). A suffix array for $T$ can be built in $O(n)$ time from the suffix tree of $T$, but construction uses up too much space. An algorithm which constructs suffix arrays without an intervening suffix tree is called a *direct construction* algorithm.

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 [13]. Recently $O(n)$ worst-case direct construction algorithms have been published in [10] and [7], and [9].

An *enhanced suffix array* is a suffix array augmented with a table of longest common prefixes, also called an LCP table [1] [13] [7]. 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 longest common prefix of $Pos[i]$ and $Pos[i+1]$. The

*lcp* array can be constructed in linear time from the suffix array [8], or at the same time the suffix array is built.

Many operations that we can do with suffix trees can be done with suffix arrays with the multiplication factor of $O(\log_2 n)$. The LCP table can lower such factor to an addition of $O(\log_2 n)$ instead of a multiplication. Properly designed algorithms for suffix trees can be modified to run with the same bounds as a suffix tree on an enhanced suffix array [1].

### 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 longest common extension computation, which is done using a $O(1)$ LCA computation over a generalized suffix tree.

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

**Theorem 1.** *The longest common extension 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 longest common extension 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 longest common extension of $S_a$ and $S_b$ (i.e. $s_a ... s_{a+c-1} = s_n ... s_{b+c-1}$). We assume that the string $S$ has a sentinel character so that no suffix of $S$ is a prefix of any other suffix of $S$ but itself.

If $i = j - 1$ then $k = i$ and $lcp[i] = c$ is the longest common extension of $S_a$ and $S_b$ 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 longest common extension query to a range-minimum-query over a range in *lcp*. As it turns out, it is possible to pre-process an array of integers (such as *lcp*) in linear time so that a query for the minimum value in a given interval of the array is answered in constant-time. Kärkkäinen

and Sanders give such an algorithm in [7], as does Farach-Colton and Bender in [3] and Kim et ali. in [9].

The idea presented below follows the algorithm based on Cartesian trees given in [4] by Gabow, Bentley and Tarjan.

We will then 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 constant time by doing a constant-time lowest common ancestor query.

**Definition 6 (Cartesian Trees).** *A Cartesian tree for an sequence of real numbers $x_1 \dots x_n$ 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 $x_{m+1} \dots x_n$.*

**Proposition 1.** *The smallest number of the interval $x_i \dots x_j$ can then be found by simply finding the lowest common ancestors of nodes $i$ and $j$ on the Cartesian tree.*

*Proof.* Given the nodes $i$ and $j$, and $v$ the lowest common ancestor of $i$ and $j$, and suppose that $i < j$. The structure of the Cartesian tree as given above is such that if a node $v$ is the lowest common ancestor of nodes $i$ and $j$, then it means that $i \leq v \leq j$, because from the construction of the Cartesian tree, 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 the tree, 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 $x_i \dots x_j$.

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

We must also show how to compute the Cartesian tree in $O(n)$. It is done using the algorithm given in [4] and [3].

We build the Cartesian tree $\mathcal{C}_i$ for the array $a_1, \dots, a_i$ from the Cartesian tree $\mathcal{C}_{i-1}$ for the array $a_1, \dots, a_{i-1}$ by following the rightmost path of the tree from its leaf to the root, until we find node $k$ such that $a_i < a_k$. We then set right subtree of node $k$ to be the left subtree of node $i$, and set node $i$ to be the right subtree of node $k$. Since each node can added to the rightmost path at most once, and leave it at most once, the algorithm runs in time $O(n)$.

Thus, in order to answer longest common extension queries in $O(1)$ with $O(n)$ pre-processing, we first built a suffix array in $O(n)$ time and space for the text concatenated with the pattern (with sentinel characters), and build the LCP table for the suffix array (either together with the construction of the suffix array itself, or in time and space $O(n)$ as given in [8]). We then create a Cartesian tree $\mathcal{C}$ for the LCP table, and pre-process it 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, their longest common extension will be the result of the range-minimum-query over $lcp_i \dots lcp_j$, which is given by a $O(1)$ LCA query over the Cartesian tree $\mathcal{C}$.

### 4.3 Proposed algorithm

The proposed 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 range-minimum query over the LCP table for the suffix array, which we give as Algorithm 2.

**Algorithm 2** Modified Landau-Vishkin approximate string matching algorithm

```
1. Build a suffix array Pos for the string P#T, together with the
   lcp table
2. Build a Cartesian tree C for the lcp table
3. Process the Cartesian tree C so that we can do O(1) LCA queries
   on it.
4. For every diagonal i of the dynamic programming table, find
   its farthest reaching 0-path with an O(1) LCA lookup on the
   Cartesian Tree for the nodes corresponding to P and the i-th
   suffix of T.
5. For d = 1 to k:
   5.1 For every diagonal i of the dynamic programming table:
      5.1.1 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).
      5.1.2 Further extend it along i by a number of cells equal
            to the RMQ of the lcp array in the range corresponding
            to the relevant suffixes of P andT (that is, P[r]...P[m]
            and T[s]..T[n]). The RMQ is given by a LCA query on the
            Cartesian tree.
      5.1.3 Extend the (d-1)-paths on diagonals i+1 and i in a
            similar way, as described in the text.
      5.1.4 Choose the farthest reaching among the three extended
            paths.
6. Every path that reached row m is a match of P in T with at most
   k mismatches and spaces.
```

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.

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

*Proof.* As commented before, construction and maintenance of suffix arrays run in $O(n)$ time and space ([11]) as it do building and maintenance of Cartesian trees. Since with this pre-processing LCA queries are done in $O(1)$, the construction of the approximate matches is computed in $O(kn)$ running time and space.

However the theoretical bounds coincide with the original ones of the Landau-Vishkin algorithm, this variation is better because in practice we are now able to use 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.

Supposing we are dealing with a good implementation of suffix trees where we use about 12 bytes per character (see comments on E. Coli treatment in [11]), constructed with $\frac{3}{2}n$ nodes. The space used for the LCA pre-processing is then $12n + C\frac{3}{2}n$ bytes, where $C$ is the space used by the LCA pre-processing for each node of the suffix tree. The suffix array and the *lcp* array can be built with atotal of $8n$ bytes. If one builds the labeled cartesian tree using $12n$ bytes (a reasonable assumption) such that the labels of the cartesian tree's nodes are actual values of *lcp*, instead of indexes, we can discard the suffix array and the *lcp* array altogether. Since the cartesian tree has exactly $n$ nodes, the final amount of space used for the preprocessing is then $12n + Cn$ bytes, which is an economy of $C\frac{n}{2}$ bytes over the suffix tree-based version, and a clever implementation of the cartesian tree or a better labelling scheme would use even less space.

## 5 Concluding remarks

Baeza-Yates and Gonnet [2] and Navarro [16] commented that although theoretically very interesting, in practice the Landau-Vishkin algorithm is much slower than expected. In addition, the use of suffix trees implies a large space usage, specially when combined with the necessary pre-processing for answering LCA queries in constant-time.

We have shown that it is possible to change the Landau-Vishkin approximate string matching algorithm to use enhanced suffix arrays instead of suffix trees for its computation of longest common extensions between suffixes of the text and the pattern, while keeping the same running time and space complexity. Actual space usage is likely to be better than the standard algorithm because most implementations of suffix trees have a much larger space usage.

## 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. R. A. Baeza-Yates and G. H. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994.
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. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *6th Annual ACM Symposium on Theory of Computing (STOC)*, page 135143, 1984.

5. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
6. D. Hirschberg. A linear space algorithm for computing the maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975.
7. J. Kärkkäinen and P. Sanders. Simpler linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming*, pages 943–955, 2003.
8. 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.
9. 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.
10. P. Ko and S. Aluru. Space-efficient linear-time construction of suffix arrays. *Journal of Discrete Algorithms*, to appear.
11. S. Kurtz. Space-efficient linear-time construction of suffix arrays. *Software — Practice and Experience*, 29(13):1149–1171, 1999.
12. G. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and new serial algorithm. In $18^{th}$ *ACM STOC*, pages 220–230, 1986.
13. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. Technical Report TR 89-14, University of Arizona, 1989.
14. E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, April 1976.
15. J.I. Munro, Raman V., and Rao S. S. Space efficient suffix trees. *Journal of Discrete Algorithms*, 2001.
16. G. Navarro. A guided tour of approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
17. E. Ukkonen. On-line Construction of Suffix-Trees. *Algorithmica*, 14:249–260, 1995.