Math. Struct. in Comp. Science (2017), *vol.* 27, *pp.* 277–295. © Cambridge University Press 2015
doi:10.1017/S0960129515000110 First published online 29 May 2015

The longest common substring problem

MAXIME CROCHEMORE[†,‡,§], COSTAS S. ILIOPOULOS[‡],
ALESSIO LANGIU[‡,¶] and FILIPPO MIGNOSI[‖]

[‡]*Department of Informatics, King's College London, London, UK*
Email: Maxime.Crochemore@kcl.ac.uk, Costas.Iliopoulos@kcl.ac.uk,
Alessio.Langiu@kcl.ac.uk
[§]*Université Paris-Est, Marne-La-Vallée, France*
[¶] *DMI, Università degli Studi di Palermo, Palermo, Italy*
[‖]*DISIM, Università dell'Aquila, L'Aquila, Italy*
Email: Filippo.Mignosi@di.univaq.it

Received 27 January 2014; revised 7 March 2015

Given a set $\mathcal{D}$ of q documents, the Longest Common Substring (LCS) problem asks, for any integer $2 \leqslant k \leqslant q$, the longest substring that appears in k documents. LCS is a well-studied problem having a wide range of applications in Bioinformatics: from microarrays to DNA sequences alignments and analysis. This problem has been solved by Hui (2000 *International Journal of Computer Science and Engineering* **15** 73–76) by using a famous constant-time solution to the Lowest Common Ancestor (LCA) problem in trees coupled with the use of suffix trees.
In this article, we present a simple method for solving the LCS problem by using suffix trees (STs) and classical union-find data structures. In turn, we show how this simple algorithm can be adapted in order to work with other space efficient data structures such as the enhanced suffix arrays (ESA) and the compressed suffix tree.

1. Introduction

One of the most powerful techniques to study living organisms is to sequence their genomes.

Bioinformatics is of course the major discipline involved in such a task and the design of algorithms capable to compare biological sequences faster and faster is of great interest for the life science community.

What Gusfield (1997) claims as 'the *first fact* of biological sequence analysis' is that: 'In biomolecular sequences (DNA, RNA or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity.' This approach has allowed the identification of critical amino acids for specific proteins allowing for example the prediction of the severity of amino acid substitutions in certain pathologies. In this respect, the possibility to have an algorithm capable to compare orthologous genes from a very large dataset of species could give some extra information with respect to the actual state of the art.

[†] This work was partly supported by EPSRC under grant EP/J017108/1.

One of the theoretical problem borrowed from the stringology field that suit this purpose is the LCS problem, which is meant to identify the substrings, i.e. a sequence of contiguous letters, that are common to two or more documents. To put it in bioinformatics terms, given a collection of q genomes, the aim of the LCS problem is to find the sequences that appear in at least k different genomes, $2 \leqslant k \leqslant q$. Those common parts probably are conserved sequences, which deserve a further biological investigation. As Gusfield (1997) says: 'the problem of finding (exactly matching) common substrings in a set of distinct strings arises as a subproblem of many heuristics developed in the biological literature to *align* a set of strings. That problem (is) called multiple alignment problem.' On the side of evolutionary studies, the k genomes having a such long common sequence are likely to have a common ancestor in the tree of life.

In this article, we present an original solution that is easier to implement and it is candidate to be faster in practice using less space than previous solutions. In Section 2, we recall some basic notions from the stringology field and we define the notation used in this paper. In Section 3, we review the classic solution that uses the ST and the LCA query. We present in Section 4 an original linear time solution that uses the classic *union-find* data structure and, in Section 5, we present some variants of this solution which are more space efficient.

2. Preliminaries

A string, also called document in this paper, is a sequence of zero or more symbols (or letters) from an alphabet Σ. A string t of length n is denoted by $t[1\,..\,n] = t_1 t_2 \ldots t_n$, where $t_i \in \Sigma$ for $1 \leqslant i \leqslant n$. The length of t is denoted by $|t| = n$. ε is the empty (zero-length) string.

A string w is a factor of t if $t = uwv$ for $u, v \in \Sigma^*$; in this case, the string w occurs at position $|u| + 1$ in t. The factor w is denoted by $t[|u| + 1\,..\,|u| + |w|]$. A prefix (or suffix) of t is a factor $t[x\,..\,y]$ such that $x = 1$ $(y = n)$, $1 \leqslant y \leqslant n$ $(1 \leqslant x \leqslant n)$. We define the ith suffix of t as the suffix starting at position i, i.e. $t[i\,..\,n]$, $1 \leqslant i \leqslant n$.

Given a text t of length n and a pattern p of length m such that $m \leqslant n$, p is said to occur in t at position i (i.e. exact match) if $p = t[i\,..\,i + m - 1]$. The position i is said to be an occurrence of p in t.

In many full-text indexing solutions, one of the fundamental data structures is the ST. Another one is the suffix array (*SA*). A complete description of the ST and the SA is beyond the scope of this paper, and can be found in any textbook on stringology (e.g. Crochemore *et al.* (2007); Gusfield (1997); Ohlebusch (2013)). Here we give a very concise definition of those data structures.

The suffix tree ST_t is a compacted trie of all the suffixes of the text t, that is, each suffix $t[i\,..\,n]$ of t, $1 \leqslant i \leqslant n$, can be read on the suffix tree ST_t in a path starting at the root and ending in the ith leaf. The number of leaves in such tree is exactly n and, since any internal explicit node is a branching node, the number of internal nodes (non-leaf) is strictly less than n. Hence, the total number of nodes is linear in the length of the text, i.e. $O(n)$. Any substring w of t can be read from the root to an internal node, either explicit or implicit (i.e. a node in the middle of a compacted path represented by a single edge). Auxiliary informations used by many construction algorithms are the suffix links. If u is the node corresponding to the path aw and v is the node corresponding to the path w, then a link from node u to node v is called suffix link. Given two nodes v and u in ST_t, the LCA of u and v is $LCA(u,v) = z$ that is the lowest explicit node traversed by both the path to u and v. The string corresponding to the path from the root to z is the LCP between the string of the path to u and the string of the path to v. Notice that ε is prefix of any string, and the root is a common ancestor of any node in the tree. Given a set $\mathcal{D} = \{t_1 \mathinner{..} t_q\}$ of documents (or strings), the generalized suffix tree $ST_{\mathcal{D}}$ is the ST of the text obtained by concatenating the documents of $\mathcal{D}$ with a symbol $\$_i$ appended at the end of each document, i.e. the generalized suffix tree $ST_{\mathcal{D}}$ is equal to the truncated suffix tree ST_t, where $t = t_1\$_1 \mathinner{..} t_i\$_i \mathinner{..} t_q\$_q$ and every path is truncated after the first \$.

The ST was introduced by Weiner (1973) together with a linear time construction algorithm. Other notably (online) linear time construction algorithms (also for integer alphabets) are (Breslauer and Italiano 2013; Farach-Colton *et al.* 2000; McCreight 1976; Ukkonen 1995). The pattern matching query time is optimal, i.e. $O(m + occ)$, where occ is the number of occurrences and m is the length of the pattern.

The recent research trend is to build compact or compressed version of the ST (Abeliuk et al. 2013; Abouelhoda et al. 2004; Fischer and Heun 2008; Fischer *et al.* 2009; Gog and Ohlebusch 2013; Grossi and Vitter 2005; Kim and Park 2005; Kim *et al.* 2008; Lin et al. 2009; Navarro and Mäkinen 2007; Ohlebusch and Gog 2009; Russo *et al.* 2011), basically using a (compressed) SA and some auxiliary structures that provide almost all the ST functionalities (for instance, top-down and bottom-up traversal, substring retrieval and pattern matching functionalities).

The suffix array SA_t of a text $t[1 \mathinner{..} n]$ is a permutation of $[1 \mathinner{..} n]$ such that $SA_t[i] = j$ if and only if, $t[j \mathinner{..} n]$ is the ith suffix of t in (ascending) lexicographic order. The SA was first introduced in Manber and Myers (1993), where a construction algorithm having $O(n \log n)$ time and an $O(m + \log n + occ)$ time pattern matching solution were presented. Later, linear time construction algorithms for the SA were presented (Kärkkäinen *et al.* 2006; Kim *et al.* 2005; Ko and Aluru 2005; Puglisi *et al.* 2007). The query time was also improved to the optimal $O(m+occ)$ in Abouelhoda et al. (2004); Fischer and Heun (2008); Fischer *et al.* (2009); Kim *et al.* (2008) with the help of another array essentially storing the lengths of LCPs between consecutive suffixes, in lexicographic order.

We remark that the query time of ST (and other similar data structures) always contains a hidden $O(\log |\Sigma|)$ factor, where Σ is the underlying alphabet. However, since in most of the cases the size of the alphabet Σ is a constant, the trend in the literature is to omit the $O(\log |\Sigma|)$ factor from the stated time complexity. In this paper, we focus on biological sequences where usually the underlying alphabet size is a very small constant (e.g. 4 for DNA/RNA sequences and 20 for protein/amino acid sequences). Finally, we note that there are several linear time ST and SA construction methods working with integer alphabet as well, e.g. Farach-Colton *et al.* (2000); Kim *et al.* (2005); Ko and Aluru (2005).

3. The classical solution

Given a set $\mathcal{D}$ of q documents, the LCS problem (also known in the literature as multiple common substring problem and longest k-common substring problem (Gusfield 1997)) asks, for any integer $2 \leqslant k \leqslant q$, for the length of the longest substring that appears in at least k documents. This problem has been solved by Hui (1992, 2000) in optimal linear time by using a generalized ST and a famous constant-time solution for the LCA query. This solution is also reported in the Gusfield's book (Gusfield 1997, Sections 7.6 and 9.7) and it is briefly reviewed in this section.

Let us suppose to have the generalized suffix tree $ST_{\mathcal{D}}$ of the set $\mathcal{D} = \{t_1 .. t_q\}$ of documents. Recall that the generalized suffix tree $ST_{\mathcal{D}}$ is defined as the suffix tree ST_t of the text $t[1..n] = t_1\$_1 .. t_i\$_i .. t_q\$_q$. Due to the uniqueness of symbols $\$_i$ and since any internal node in the ST is a branching node, there are no internal nodes in the tree corresponding to a substring containing a dollar. That is, each path in the tree containing a \$ corresponds to a leaf or to a leaf incoming edge. Hence, we can think of $ST_{\mathcal{D}}$ as a truncated tree, where any path is truncated as soon as a \$ is met. The advantage of the truncated tree is that it does not contain any artificial string produced by document concatenation. Moreover, the last symbol in any path ending on a leaf is a \$, and, then, it is easy (i.e. a constant-time operation) to know, given a leaf, what document it represents a suffix of. Alternatively, we can associate each leaf in $ST_{\mathcal{D}}$ to a document t_i, i.e. we can mark with i every leaf having $\$_i$ as the first dollar (left to right) contained in its path from the root.

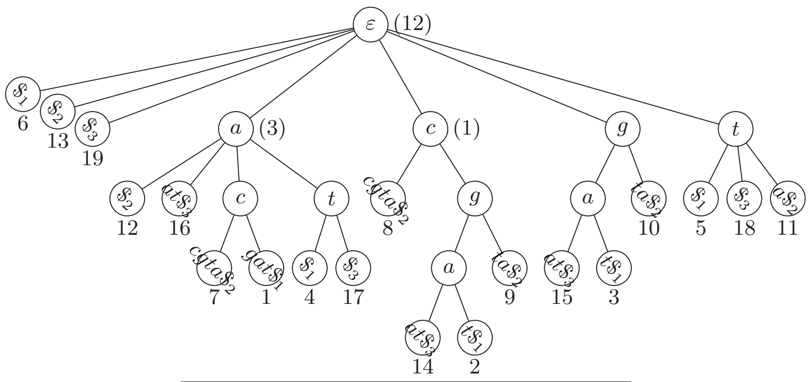
Now, let us suppose to associate to any internal node v the value $C(v)$ that is the counter of how many different documents are associated to the leaves in the subtree rooted in v. It is easy to prove that one can populate an array $\mathrm{L}[k]$, $1 \leqslant k \leqslant q$, accumulating the lengths of the longest substring common to exactly k documents while visiting such augmented tree via a depth-first traversal. (In the meantime, by using a simple book-keeping strategy, we can store a reference to one of such substrings.) A further scan of the array L will produce the correct $\mathrm{LCS}[k]$ array, $2 \leqslant k \leqslant q$, where $\mathrm{LCS}[k]$ is the length of the longest substring that appears in at least k different documents.

Now the point is how to compute the C values. This is a classic problem on trees: the Color-Set-Size (CSS) problem.

Definition Given a tree T with coloured leaves, the CSS problem asks, for each internal node v in T, for the number of different colours that are present in the sub tree rooted in v.

L. Hui presented in his paper (Hui 1992) a solution to the CSS problem where the desired C value, for each internal node, is computed as the number of leaves in the rooted subtree minus the number of the duplicated colours in such subtree.

In order to efficiently compute the number of duplicated colours in a subtree, given a tree with n leaves and q colours, one builds q lists of pointers to the leaves having the same colour as they appear at the bottom of the tree from left to right, that is, q lists of lexicographically ordered leaves of the same colour. Obviously, the sum of the length of all the lists is equal to n. Let us assume that every node in the tree is equipped with a counter d. After preparing in linear time the tree for constant-time LCA queries (Berkman and Vishkin 1993; Gonnet *et al.* 2000; Harel and Tarjan 1984; Schieber and Vishkin 1988), such queries are conducted on pairs of adjacent elements in the lists. Let u, v be two of such adjacent leaves and $\mathrm{LCA}(u, v) = z$. For each node z that is the LCA



List 1	6	1	4	2	3	5	
List 2	13	12	7	8	9	10	11
List 3	19	16	17	14	15	18	

Fig. 1. Generalized suffix tree of the set $\mathcal{D} = \{acgat, accgta, cgaat\}$ illustrated together with the $|\mathcal{D}| = 3$ lists of leaves of a same (document) colour. Numbers between parenthesis are counter d values. For instance, if we call v the node of path a, since $\text{LCA}(1,4) = \text{LCA}(12,7) = \text{LCA}(16,17) = v$, then $d(v) = 3$.

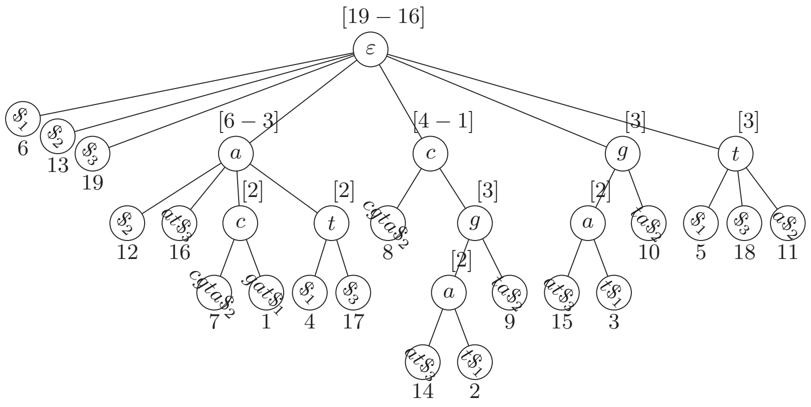


Fig. 2. Generalized suffix tree of the set $\mathcal{D} = \{acgat, accgta, cgaat\}$ illustrated. $C(v)$ value of any internal node v is reported within square brackets.

of two consecutive leaves, d of z is incremented; Figure 1 shows an example. Let us call $D(v)$ the sum of the counters $d(u)$ of all nodes u in the subtree rooted in v. $D(v)$ is the number of occurrences of duplicated colours in the subtree of v and it can be computed via a post-order traversal of the tree. In this way, the number $C(v)$ of distinct colours is computed, for each node v of the tree, as the difference $C(v) = S(v) - D(v)$ between the total number $S(v)$ of leaves in the rooted subtree and the number of duplicated colours $D(v) = \sum_u d(u)$, $u \in \{\text{subtree rooted in } v\}$. See Figure 2 for an illustration. Notice that computing $S(v)$ is a standard operation on trees.

In the next section, we present a new solution for the CSS problem that uses a well-known, fast and space efficient *union-find* data structure instead of the LCA queries used in the classical solution. Even if the time complexity does not get asymptotically improved, the *union-find* based solution has some advantages that make it be simple, fast, and easily extendible to more space efficient variants.

4. A simple linear time solution

In previous section, we have briefly summarized the classic solution for the LCS problem. It essentially reduces the LCS problem to the CSS problem on the generalized ST of a document collection. Maintaining this approach, we show a simple solution for the CSS problem that turns out to be a simple solution for the LCS problem, as well.

These two problems are strictly related. Therefore, as in the classical solution, if we build the generalized suffix tree $ST_{\mathcal{D}}$ of a set $\mathcal{D}$ of documents and we colour any leaf by assigning a different colour to each document, the LCS problem can be solved with a simple traverse of $ST_{\mathcal{D}}$ provided that $ST_{\mathcal{D}}$ has been preprocessed to solve the CSS problem. Notice that the CSS solution presented by Hui requires the preprocessing of the generalized ST in order to answer efficiently (in constant time) the LCA query.

In the last decades, many papers have appeared dealing with a set of problems called coloured range queries, see, for instance, (Gagie *et al.* 2013; Navarro 2014) and references therein. These problems of the computational geometry research field also have many applications in algorithms on strings and information retrieval fields, where many of the recent solutions make use of the SA of a text and a geometry query on it. Indeed, also the LCS problem can be addressed in such a settings, as we will show in next section, by using a compressed suffix tree (CST) on top of a compressed SA and the range minimum query. Since a complete dissertation on coloured range queries is out of the scope of this paper, we just report that the time and space complexity achieved by the solutions treated in this paper for the LCS problem are equivalent to the one achieved by using the SA, either compressed or not, and the coloured range queries.

In this section, we present a new simple solution for the CSS problem that uses a classical *union-find* data structure. The *union-find* data structures for disjoint sets have myriads of practical applications, see, for instance, (Galil and Italiano 1991). Their running time, due to a multiplicative inverse of the Ackermann function, is not linear, but, due to their simplicity and to the small constants involved, they turn out to be very fast, in practice. Moreover, from a theoretical point of view, Tarjan *et al.* (Gabow and Tarjan 1985) showed how to get rid of the multiplicative Ackermann function when the tree of the union is known in advance. In Loebl and Nesetril (1988a,b); Lucas (1990) it is shown that if the *finds* are performed in post order, at most one find for each element, then the overall time is truly linear. Moreover, when the union-find data structure in Gabow and Tarjan (1985) is used, the representative element of a set, i.e. the element returned by a find operation, it is known in advance, and it is the father between the two representatives. In the general case, the representative of a set is a generic element of the set. In what follows, we use union-find to handle sets of nodes of a tree and the union are always performed on two sets containing an element that is an ancestor of all the nodes in both sets. We can associate such ancestor to the representative of the set generated by union, with a simple book-keeping strategy with constant-time operations after any union. We assume, in the rest of the paper, that a find operation returns such ancestor node within the queried set and we hide all the necessary operations to handle the associations between the representative and the ancestor node. Notice that when the structure in Gabow and Tarjan (1985) is used, no operation has to be hidden.

Recently, many algorithms appeared that, under some special conditions, achieve *union-find* in linear time (see, for instance, (Fiorio and Gustedt 1996; Gustedt 1998)) even including a new operation *delete* (Alstrup *et al.* 2005; Ben-Amram and Yoffe 2011; Dillencourt *et al.* 1992).

4.1. *A new colorsetsize algorithm*

For the sake of clarity, we first present a simple algorithm that solves the CSS problem in a non-efficient way and we refine it later into a linear time solution.

Assume a tree T is given and any node in T is augmented by a counter C. We follow the simple idea of dynamically assigning a set of colours to each node, where the colours are the unique colours present in the subtree rooted at the node. Proceeding in a recursive way along a post-order visit of the given tree, we assign to a leaf a set containing the colour of the leaf, and we assign to an internal node the union of the sets coming from its children. At the end of any recursive step, we store the size of the current set to the colour counter C of the examined node. Algorithm 0 follows this idea.

Algorithm 0 Recursive processing the tree T to compute the C values.

```
 1: function ColorSetSize($v$)
 2:     set ← $\varnothing$
 3:     if v is a leaf then
 4:         union(set,{color($v$)})
 5:     else
 6:         for each child $u$ of $v$ do
 7:             union(set,ColorSetSize($u$))
 8:         end for
 9:     end if
10:     C($v$) ← size(set)
11:     return set
12: end function
13: ColorSetSize(root($T$))
```

Algorithm 0 is conceptually very simple and it correctly computes, for any node v in T, the number of different colours C(v) present in the subtree rooted in v. A proof of this fact is straightforward and uses a simple property on sets. Unfortunately, Algorithm 0 cannot be implemented in an efficient time, because of the union of sets that are not disjoint. We can improve it by keeping disjoint sets of sibling nodes. We maintain sets of leaves instead of set of colours and we use, in a book-keeping strategy, a global array of previous occurrence of colours to maintain sets whose colours are disjoint. We also keep track of the size of the set associated with a node by maintaining this value in the colour counter field of the node. Hence, we initialize the C counter of each node to 0, and we increment it accordingly to the cardinality of the associated set. In order to manage the set of leaves, we use a classic *union-find* data structure for disjoint sets. Instead of associating a set with a node, we include such node into a set. Hence, for any node v, there exist a set containing v and all the nodes in the subtree rooted in v. Algorithm 1 is the resulting algorithm.

Let us summarize Algorithm 1 steps in what follows. Assume that q is the number of different colours in the given tree T. An auxiliary global array P (P stands for 'previous visited leaf') of size q is used along a recursive post-order traversal of T in order to keep track of the previous occurrence of the colour $c_k, 1 \leqslant k \leqslant q$.

Algorithm 1 Recursive processing the tree T to compute the C values. *make-set*, *union* and *find* are common operations on *union-find* data structures. The operation *colour* on a leaf v returns the associate colour c_k. *find*(*null*) returns *null*. Notice that z will always be an ancestor of v.

```
1:  function ColorSetSize($v$)
2:      make-set($v$)
3:      C($v$) $\leftarrow$ 0
4:      if v is a leaf then
5:          $z \leftarrow$ find(P[color($v$)])
6:          P[color($v$)] $\leftarrow v$
7:          if $z$ is not null then
8:              C($z$) $\leftarrow$ C($z$) $-$ 1
9:          end if
10:         C($v$) $\leftarrow$ 1
11:     else
12:         for each child node $u$ of $v$ do
13:             ColorSetSize($u$)
14:             union(find($v$), find($u$))
15:             C($v$) $\leftarrow$ C($v$) + C($u$)
16:         end for
17:     end if
18: end function
19: ColorSetSize(root($T$))
```

Recursively, we associate to any node a set containing itself and the union of the sets associated to its children and we add children colour counter values to the parent counter. Once a leaf is visited, we find the root of the smallest subtree containing it and the previous visited leaf having the same colour, and we decrement its colour counter as this colour will be added again at the end of the recursion of such subtree. At the end of each recursion, the colour counter field C of the examined node is fixed, i.e. it does not change any more, and it stores the number of different colours in the rooted subtree. Figure 3 illustrates some (non-consecutive) steps of an example.

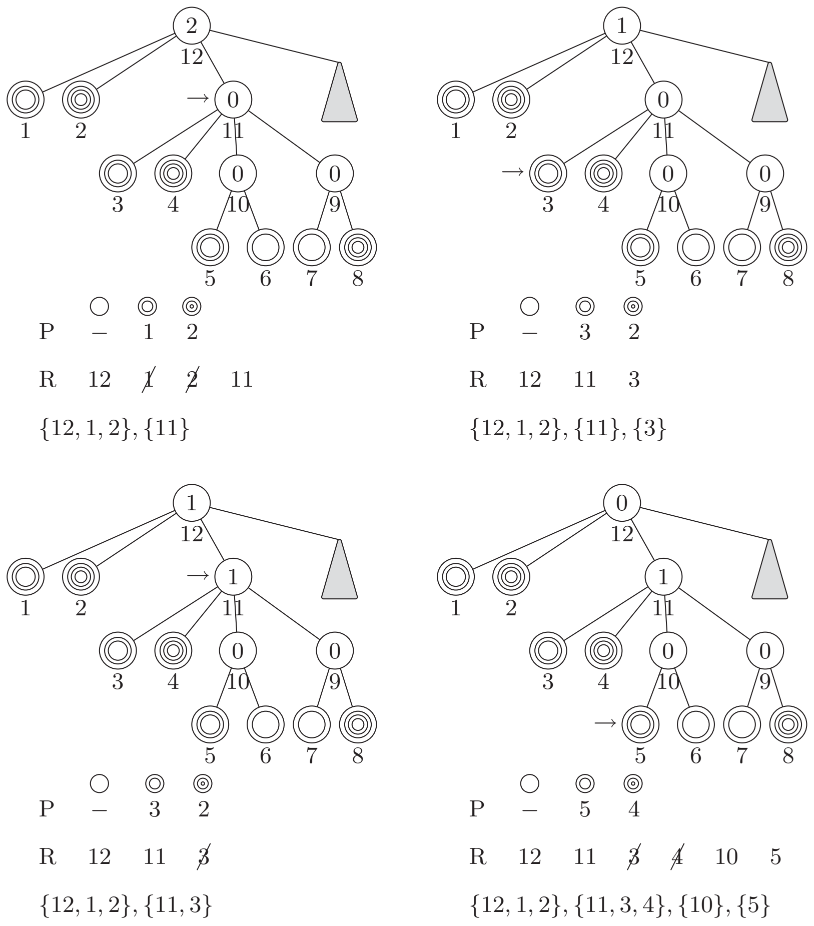


Fig. 3. Some key steps of Algorithm 1 illustrated. Algorithm 1 computes C for any node v in the given tree. Array P is reported together with the list R of recursive calls to function ColorSetSize, and the sets maintained by the *union-find* data structure. The arrow indicates the node currently considered.

Proposition 4.1. Assume a tree T is given. For each internal node v in T, ColorSetSize of v correctly computes the C value of any node in the subtree rooted in v, v included.

Proof. The property that we want to prove is the following: After a call to function ColorSetSize(v), the set containing v also contains all the nodes in the subtree T_v rooted in v. Moreover, ColorSetSize(v) correctly computes the C value of all the nodes in the subtree T_v. The proof is by induction on the number of nodes (internal nodes and leaves) in the subtree T_v rooted in v.

If the number of nodes in T_v is 1, its root v is a leaf and the base of the induction is clearly correct. Indeed, C(v), i.e. the colour counter of node v, is set to 1 on line 10. Notice that the global array P is initialized with *null* pointers. Let $u_1 \cdots u_h$, $h \geqslant 1$, be the sequence of children of v in the same order as they are considered in the **for** cycle at line

12. The inductive hypothesis holds on the tree $T_{v|u_h}$ that is the original subtree T_v without the subtree rooted in u_h, and it holds also for T_{u_h} itself. Let us focus on the execution of function ColorSetSize(v) and freeze it after the child u_{h-1} has been processed by the whole **for** cycle on lines 12–16. Up to this point, ColorSetSize(v) has executed the same operations as ColorSetSize(root($T_{v|u_h}$)) would have executed. We call C'(v) the value of C(v) at this step of the algorithm. Hence, by inductive hypothesis applied to $T_{v|u_h}$, the set that contains v it also contains the internal nodes of its subtree, and all the leaves that are present in $T_{v|u_h}$. Moreover, the C values on all the nodes of the subtree $T_{v|u_h}$ have been correctly computed and have been stored in their C fields, C'(v) included. Notice that, all the leaves in subtree $T_{v|u_h}$ are contained in the set containing v by effect of the union call at line 14.

Let us move forward on the execution of ColorSetSize(v) considering **for** cycle on line 12 for the child u_h. Let us also suppose that the number of leaves in $T_{v|u_h}$ having a colour that also appears in T_{u_h} is d, that is, the number of duplicated colours in those two subtrees. After we make the call ColorSetSize(u_h) on line 13, by inductive hypothesis applied to T_{u_h}, the set containing u_h also contains the node u_h and all the nodes in T_{u_h}. Moreover, the C values on all nodes in T_{u_h}, C(u_h) included, have been correctly computed. Furthermore, for each leaf in T_{u_h}, the colour counter of the representative of the set containing the previous occurrence of a leaf having the same colour (pointed by array P) has been decremented by effect of lines 5 and 8, and the array P is then accordingly updated at line 6. Hence, since d is the number of colours in $T_{v|u_h}$ that appear also in T_{u_h} and since all the leaves in $T_{v|u_h}$ are, at this step, contained in the same set of v, the colour counter of node v is now equal to C(v) = C'(v) − d.

Then, at line 15, the number of colours in T_{u_h}, i.e. C(u_h) $\geqslant d$, which is correct by induction hypothesis, is added to C(v) without counting duplicate colours, and the set containing v also contains the nodes in $T_{v|u_h}$, by hypothesis, together with the nodes in T_{u_h} by the union performed at line 14. This concludes the proof. □

Let us now analyse the time and space requirements of Algorithm 1. From a simple analysis of the pseudocode of Algorithm 1, it is easy to see that any recursion of Algorithm 1 executes some constant-time assignments and at most a fixed number of any of the make-set, union and find operations of a *union-find* data structure. Let us call t_m, t_u, t_f the time of make-set, union and find operations, respectively.

Theorem 4.2. Assume tree T having n nodes is given. Algorithm 1 runs in $O(n\ (t_m+t_u+t_f))$ time.

The make-set and union operations are standard constant-time operation in many common linear space *union-find* data structure, i.e. $t_m = t_u = O(1)$ (Gabow and Tarjan 1985; Galil and Italiano 1991; Loebl and Nesetril 1988a,b; Lucas 1990). There are many classical *union-find* solutions providing a find operation in $t_f = O(\log m)$, where m is the cardinality of the set containing the queried element (Galil and Italiano 1991). The time used by such structures to execute a find query amortized over the n elements is $O(\alpha(m))$, where α is a functional inverse of Ackermanns function, that is an extremely slow growing function that, for any practical number m, is less than 4. Practically speaking,

$\alpha(m)$ is usually considered just a small constant. Moreover, since there exist *union-find* implementations that are very fast in practice, such solutions are usually preferred to theoretically more efficient solutions. In Galil and Italiano (1991), six algorithms are described that make use of either *union by size* or *union by rank* as union strategy, and they use *compaction rules* to speed up the time bound (Galil and Italiano 1991, Theorem 1.1.1).

Corollary 4.3. Assume tree T having n nodes is given. By using any data structure for disjoint set in Galil and Italiano (1991) that make use of union by size or union by rank, Algorithm 1 runs in $O(n\ \alpha(n))$ time and linear space in the RAM model.

However, the *union-find* data structure presented in Gabow and Tarjan (1985) is able to answer to a find query in amortized constant time, when a find operation is followed by a union operation and the sequence of find operations is known in advance. This is the case of Algorithm 1, where the union operations strictly follow the recursive visit of the input tree.

Corollary 4.4. Assume tree T having n nodes is given. By using Tarjan's data structure (Gabow and Tarjan 1985), Algorithm 1 runs in linear time and linear space in the RAM model.

5. Reducing space

In this section, we describe how to reduce the space used by the CSS solution presented in the previous section and the total space used for the LCS problem.

A first observation leads to keep low the space used by the *union-find* data structure.

Algorithm 1 uses, at any moment, the leaves pointed by array P and no other leaves are used in order to adjust the colour counters in case of duplicates. At any time, those leaves are at most q, same as the number of encountered different colours. Moreover, the internal nodes are to be released as soon as their subtree is entirely visited. Putting these observations together, we adapt Algorithm 1 to use the delete operation provided by Alstrup *et al.* (2005) to keep low the space used by the *union-find-delete* data structure. Algorithm 2 reports the adapted pseudocode.

The delete operation is not part of the classical set of operations of the *union-find* data structures, where the element in the set are always maintained once they are created by a make-set operation. The delete operation appears only in an extension of the *union-find* data structures called *union-find-delete* data structures (Alstrup *et al.* 2005; Ben-Amram and Yoffe 2011). Recall that, in Alstrup *et al.* (2005), the delete operation is also a constant-time operation, same as make-set and union operations, while a find takes $O(\alpha(m))$ amortized time, where m is the number of elements in the set returned by the find operation, and α is a functional inverse of Ackermanns function. The space is linear in the number of elements simultaneously maintained, at any time. Algorithm 2 maintains, at any time, one internal node for any nested call to the function ColorSetSize and, at most, q leaves.

Algorithm 2 Recursive processing the tree T to compute the C values. *make-set*, *union*, *find* and *delete* are common operations on *union-find-delete* data structures.

```
function ColorSetSize($v$)
    make-set($v$)
    C($v$) ← 0
    if v is a leaf then
        $z$ ← find(P[color($v$)])
        if $z$ is not null then
            delete(P[color($v$)])
            C($z$) ← C($z$) −1
        end if
        P[color($v$)] ← $v$
        C($v$) ← 1
    else
        for each child $u$ of $v$ do
            ColorSetSize($u$)
            union($v$,$u$)
            C($v$) ← C($v$) + C($u$)
            if $u$ is NOT a leaf then
                delete($u$)
            end if
        end for
    end if
end function
ColorSetSize(root($T$))
```

The number of recursions of function ColorSetSize is bounded by the height of the tree T, that is the number of explicit nodes in the deepest path in the tree. Then, for a general tree having n nodes, the number of recursions is n in the worst case, but it is expected to be much less in practical cases, where the tree T is more balanced.

When CSS is used for the LCS problem, the tree T is a ST and it is proved by Szpankowski (1993, 2001) that the height of such trees tends almost surely to $O(\log n)$ under some probabilistic models, where n is the length of the text.

Moreover, a space linear in the height of the tree is also implicitly used by the recursion stack. Even if algorithms in this paper are stated in recursive form, it is easy to obtain non-recursive versions by using standard programming techniques, where a stack is explicitly used to perform the post-order visit of the tree.

Let us summarize the results presented so far in the following proposition.

Proposition 5.1. Assume a tree T is given having n nodes, equipped with a counter, and coloured leaves with q different colours. By using the *union-find-delete* data structure of (Alstrup *et al.* 2005), Algorithm 2 runs in $O(n\,\alpha(n))$ time and uses $O(q + height(T))$ extra space to compute all the C values.

The second observation of this section concerns the output size of CSS and LCS problems. Even if the two problems are deeply related, their outputs are totally different in size. In fact, the problem CSS has input and output of the size of the given tree T, while LCS has output of size q, that is the number of different colours.

While the output of the first problem maintains the tree structure, the output of the second is an array L that, for each k, $2 \leqslant k \leqslant q$, contains the length of a longest substring that appears in at least k strings *and*, optionally, a pointer to one of such substrings (i.e. the document and the position where it appears). The outputs is then proportional to q (more precisely it has the size of $O(q)$ integers).

We want to obtain a direct solution of the LCS problem that does not use a full solution of the CSS problem and, consequently, does not use CSS output space. First of all, we notice that array L can be obtained during the tree traversal of above algorithms. Indeed, if a node has been examined together with its rooted subtree during the visit, that is, at the end of a recursion, its colour counter C will not change anymore. Therefore, if the colour counter is equal to k and the string depth of the node is greater than the length contained in L[k], then we replace the contents of L[k] with the string length of such node[†]. In addition, a pointer to one occurrence (document and starting position) of such string can be stored as well. Since array L is defined for $2 \leqslant k \leqslant q$, according with LCS definition, and since the C values of leaves are equal to 1 by definition, then the check for an update of array L is executed right after a C value is computed for an internal node, avoiding useless checks on leaves.

In this way, L[k] contains, at any moment, the length of a longest string that is common to exactly k substrings, up to that step of the visit over the tree. Along a further linear scan of the array L, from the smallest to the bigger index, we can simply adjust the values to fit the at least k requirement of the LCS problem.

Function ColorSetSize of Algorithm 2 can easily report the correct C value of a node as soon as a recursion ends. But, the algorithm also stores temporary values in the colour counter C of parent nodes, while a subtree is traversed. The number of subtree roots whose colour counter can virtually be decremented, according to the presence of duplicate colours, is, at most, height(T). Hence, we maintain such temporary counters in a global array C of length height(T) indexed by node depth.

Now, since we already know how to get rid of the C counter fields in the tree nodes, we notice that it is possible to use a compressed index to simulate the functionalities of a ST, and, in particular, to perform a post-order visit on it. Obviously, such compressed structures have query time and space requirements strongly dependent on the underling compressed SA used and some parameters.

The research of efficient (mainly in space) data structures that can be used instead of STs has become an independent research field. Up to less than a decade ago, the most commonly used data structures are STs, SAs, DAWGs and compact DAWGs. Usually any problem that can be settled by using one of such data structures can also be settled by using any of the other ones.

[†] The string length of a node is classically defined as the length of the concatenated labels on the path from the root to such node.

Algorithm 3 Processing the collection $\mathcal{D}$ to solve the LCS problem by using a *compressed suffix tree* and a *union-find* data structure.

```
function LightweightCSS($v$)
    make-set($v$)
    C[node-depth($v$)] $\leftarrow 0$
    if $v$ is a leaf then
        $z \leftarrow$ find(P[color($v$)])
        if $z$ is not null then
            delete(P[color($v$)])
            C[node-depth($z$)] $\leftarrow$ C[node-depth($z$)] $- 1$
        end if
        P[color($v$)] $\leftarrow v$
        C[node-depth($v$)] $\leftarrow 1$
    else
        for each child $u$ of $v$ do
            C[node-depth($v$)] $\leftarrow$ C[node-depth($v$)] $+$ LightweightCSS($u$)
            union($v$,$u$)
            if $u$ is NOT a leaf then
                delete($u$)
            end if
        end for
        if L[C[node-depth($v$)]] $<$ string-depth($v$) then
            L[C[node-depth($v$)]] $\leftarrow$ string-depth($v$)
            S[C[node-depth($v$)]] $\leftarrow$ pos($v$)
        end if
    end if
    return C[node-depth($v$)]
end function

Initialize P, L, and S
Build CST$_{\mathcal{D}}$
LightweightCSS(root(CST$_{\mathcal{D}}$))
for $i = q, q-1, \ldots 2$ do
    if L[$i-1$] $<$ L[$i$] then
        L[$i-1$] $\leftarrow$ L[$i$]
        S[$i-1$] $\leftarrow$ S[$i$]
    end if
end for
return L,S
```

In the meantime, other space-efficient related data structures started to appear in research papers. For instance, compressed SA and CST are showed to have the potential to replace in many applications STs by using less space exploiting redundancy of the text. (Abeliuk et al. 2013; Abouelhoda et al. 2004; Fischer and Heun 2008; Fischer *et al.* 2009; Gog and Ohlebusch 2013; Grossi and Vitter 2005; Kim *et al.* 2008; Kim and Park 2005; Lin et al. 2009; Navarro and Mäkinen 2007; Ohlebusch and Gog 2009; Russo *et al.* 2011).

Latter data structures are space thrifty by using only $nH_k + O(n)$ *bits*. We refer to Fischer et al. paper (Fischer *et al.* 2009) for a comparison between different compressed indexes and their trade-off between occupied space and query time, where they summarize crucial values in Fischer *et al.* (2009, Table 1). We propose Algorithm 3 as a variant of our LCS solution using a CST. Given a document collection $\mathcal{D}$, we associate a unique colour to each of the q documents in $\mathcal{D}$. The Algorithm returns the array L and S; L contains, for $2 \leqslant k \leqslant q$, the length L[$k$] of a longest substring common to at least k documents, and S[k] contains a reference to one of such substrings. Function colour retrieves the colour associated to a leaf. Function make-set, union, find and delete are standard operations on *union-find-delete* data structures. Function node-depth, string-depth and pos are standard functions on STs (and CSTs), as well as, to know if a node is a leaf, and performing a post-order traversal of a rooted subtree. Notice that, every call to function LightweightCSS needs to query the CST to retrieve children, node-depth and string-depth of a given internal node.

Proposition 5.2. Assume a collection $\mathcal{D}$ of q documents having total length n is given. By using the compressed suffix tree $\mathrm{CST}_{\mathcal{D}}$ of Sadakane (2007) and the *union-find-delete* data structure of Alstrup *et al.* (2005), it is possible to solve the LCS problem in $O(n\ \alpha(n))$ time and $O(n) + (1 + 1/\epsilon)nH_k$ bits space, where $0 < \epsilon < 1$ and H_k is the k-order empirical entropy of the concatenated text in $\mathcal{D}$.

As last observation, we notice that in many compressed text index based on a compressed SA, many functionality are obtained by using a range minimum query on the longest common prefix (LCP) table, and, moreover, they support LCA queries without using extra space or extra preprocessing.

Let us recall how to simulate a LCA query over a SA. Given a tree T and two nodes u and v, the problem of finding the lowest node in the tree that has both u and v in its rooted subtree is called LCA. This problem was posed in Aho *et al.* (1976), but the first linear preprocessing time solution and constant-time query is due to Harel and Tarjan (1984) based on heavy path decomposition. Many improvements, mostly in practical space requirements, appeared recently (see, for instance, (Bender and Farach-Colton 2000; Fischer and Heun 2006; Schieber and Vishkin 1988)) which use different approaches: Cartesian trees, geometrical range queries and lookup tables.

In the case where T is the suffix tree ST_t of a text t, the LCA of two leaves is equivalent to the LCP of two suffixes. By using the suffix array SA of t and the associated LCP table (which contains the LCP values of consecutive suffixes in lexicographic order, that is, $\mathrm{LCP}[i] = \mathrm{LCP}(SA[i-1], SA[i])$), it is possible to find the LCP of two given suffixes u and v by computing the minimum value in $\mathrm{LCP}[x\,..\,y]$, where x and y are the index of i and j in the SA, respectively. Therefore, suppose to have the inverse suffix array SA^{-1} ($SA^{-1}[i] = x \iff SA[x] = i$), it is possible to compute in constant time the LCP of i and j as $min\ \{LCP[x] \mid SA^{-1}[i] \leqslant x \leqslant SA^{-1}[j]\}$ by using the range minimum query algorithm (see for instance (Fischer and Heun 2007)).

Since many CSTs (Fischer and Heun 2007; Russo *et al.* 2011; Sadakane 2007) support LCA query using no extra space, we show a variant of Algorithm 3 that uses LCA instead

Algorithm 4 Processing the collection $\mathcal{D}$ to solve the LCS problem by using a *compressed suffix tree* supporting the LCA query.

```
1:  function LightweightCSS($v$)
2:      C[node-depth($v$)] $\leftarrow 0$
3:      if $v$ is a leaf then
4:          if P[color($v$)] is not null then
5:              $z \leftarrow$ LCA(P[color($v$)])
6:              C[node-depth($z$)] $\leftarrow$ C[node-depth($z$)] $- 1$
7:          end if
8:          P[color($v$)] $\leftarrow v$
9:          C[node-depth($v$)] $\leftarrow 1$
10:     else
11:         for any child $u$ of $v$ do
12:             C[node-depth($v$)] $\leftarrow$ C[node-depth($v$)] $+$ LightweightCSS($u$)
13:         end for
14:         if L[C[node-depth($v$)]] $<$ string-depth($v$) then
15:             L[C[node-depth($v$)]] $\leftarrow$ string-depth($v$)
16:             S[C[node-depth($v$)]] $\leftarrow$ pos($v$)
17:         end if
18:     end if
19:     return C[node-depth($v$)]
20: end function
21:
22: Initialize P, L, and S
23: Build CST$_{\mathcal{D}}$
24: LightweightCSS(root(CST$_{\mathcal{D}}$))
25: for $i = q, q-1, \ldots 2$ do
26:     if L[$i-1$] $<$ L[$i$] then
27:         L[$i-1$] $\leftarrow$ L[$i$]
28:         S[$i-1$] $\leftarrow$ S[$i$]
29:     end if
30: end for
31: return L,S
```

of union-find. We simply replace a find(u) call in Algorithm 3 by a LCA(u,v) query. In fact, due to the post-order visit of the tree, i.e. the nested recursion of our algorithms, those two operations are equivalent. Algorithm 4 reflects this observation and it turns out to be conceptually very close to the classical solution of the CSS problem implemented by using CSTs (Ohlebusch 2013).

We state our results by using the most efficient compressed index, to the best of our knowledge, in terms of space occupancy.

Proposition 5.3. Assume a collection $\mathcal{D}$ of $q \geqslant 2$ documents having total length n is given. By using the Compressed Suffix Tree CST$_{\mathcal{D}}$ in Russo *et al.* (2011), Algorithm 4 solves the LCS problem in $O(n \log^{1+\epsilon} n)$ time and $O(q + \text{height}(ST_t)) + nH_k + o(n)$ bits of space, where H_k is the k-order empirical entropy of t, and $0 < \epsilon < 1$.

Notice that, $O(q + \text{height}(ST_t)) + nH_k + o(n)$ bits can be sublinear in n. Recall that, given a collection of strings of total length n whose longest string has length m, the height of the generalize ST of such collection is, in the worst case, $O(m)$, and, in average, $O(\log m)$ (Szpankowski 1993, 2001). Moreover, due to the simplicity of this solution, which essentially is a post-order visit on a CST supporting LCA queries and a simple book-keeping of q values, any further improvement of CST in terms of query time and occupied space, can be directly integrated to above algorithms and leads to better performances.

Acknowledgements

We are grateful to the anonymous referees that have contributed to improve the paper with their suggestions, especially for having pointed out that the query time on the Sadakane's CST for a string-depth of an internal node is constant and it is $O(\log^{\epsilon} n)$ for the leaves. This suggestion has lead us to get rid of a $O(\log^{\epsilon} n)$ factor in the running time stated in Proposition 5.2.

References

Abeliuk, A., Cánovas, R. and Navarro, G. (2013). Practical compressed suffix trees. *Algorithms* **6** (2) 319–351.

Abouelhoda, M. I., Kurtz, S. and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms* **2** (1) 53–86.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1976). On finding lowest common ancestors in trees. *SIAM Journal on Computing* **5** (1) 115–132.

Alstrup, S., Gørtz, I. L., Rauhe, T., Thorup, M. and Zwick, U. (2005). Union-find with constant time deletions. In: Caires, L., Italiano, G. F., Monteiro, L., Palamidessi, C. and Yung, M. (eds.) ICALP, *Springer Lecture Notes in Computer Science* **3580** 78–89.

Ben-Amram, A. M. and Yoffe, S. (2011). A simple and efficient union-find-delete algorithm. *Theoretical Computer Science* **412** (4–5) 487–492.

Bender, M. A. and Farach-Colton, M. (2000). The LCA problem revisited. In: Gonnet G. H., Panario, D. and Viola, A. (eds.) *Proceedings of the Theoretical Informatics, 4th Latin American Symposium*, Punta del Este, Uruguay, April 10–14, 2000. *Springer Lecture Notes in Computer Science* **1776** 88–94.

Berkman, O. and Vishkin, U. (1993). Recursive star-tree parallel data structure. *SIAM Journal on Computing* **22** (2) 221–242.

Breslauer, D. and Italiano, G. F. (2013). Near real-time suffix tree construction via the fringe marked ancestor problem. *Journal of Discrete Algorithms* **18** 32–48.

Crochemore, M., Hancart, C. and Lecroq, T. (2007). *Algorithms on Strings.* Cambridge University Press, 392.

Dillencourt, M. B., Samet, H. and Tamminen, M. (1992). A general approach to connected-component labelling for arbitrary image representations. *Journal of ACM* **39** (2) 253–280.

Farach-Colton, M., Ferragina, P. and Muthukrishnan, S. (November 2000). On the sorting-complexity of suffix tree construction. *Journal of ACM* **47** (6) 987–1011.

Fiorio, C. and Gustedt, J. (1996). Two linear time Union-Find strategies for image processing. *Theoretical Computer Science* **154** (2) 165–181.

Fischer, J. and Heun, V. (2006). Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Lewenstein, M. and Valiente, G. (eds.) *CPM*, *Springer Lecture Notes in Computer Science* **4009** 36–48.

Fischer, J. and Heun, V. (2007). A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M. and Zhang, G. (eds.) ESCAPE, *Springer Lecture Notes in Computer Science* **4614** 459–470.

Fischer, J. and Heun, V. (2008). Range median of minima queries, super-cartesian trees, and text indexing. In: Miller, M. and Wada, K. (eds.) IWOCA, College Publications 239–252.

Fischer, J., Mäkinen, V. and Navarro, G. (2009). Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science* **410** (51) 5354–5364.

Gabow, H. N. and Tarjan, R. E. (1985). A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences* **30** (2) 209–221.

Gagie, T., Kärkkäinen, J., Navarro, G. and Puglisi, S. J. (2013). Colored range queries and document retrieval. *Theoretical Computer Science* **483** 36–50.

Galil, Z. and Italiano, G. F. (1991). Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys* **23** (3) 319–344.

Gog, S. and Ohlebusch, E. (2013). Compressed suffix trees: Efficient computation and storage of LCP-values. *ACM Journal of Experimental Algorithmics* **18** 2.1:1–2.1:31.

Gonnet, G. H., Panario, D. and Viola, A. (eds.) (2000). In: *Proceedings of the* LATIN *Theoretical Informatics, 4th Latin American Symposium*, Punta del Este, Uruguay, April 10–14, 2000. *Springer Lecture Notes in Computer Science* 1776.

Grossi, R. and Vitter J. S. (2005). Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing* **35** (2) 378–407.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*, Cambridge University Press.

Gustedt, J. (1998). Efficient Union-Find for planar graphs and other sparse graph classes. *Theoretical Computer Science* **203** (1) 123–141.

Harel, D. and Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing* **13** (2) 338–355.

Hui, L. C. K. (1992). Color set size problem with application to string matching. In: Apostolico, A., Crochemore, M., Galil, Z. and Manber, U. (eds.) CPM, *Springer Lecture Notes in Computer Science* **644** 230–243.

Hui, L. C. K. (2000). A practical algorithm to find longest common substring in linear time. *International Journal of Computer Science and Engineering* **15** 73–76.

Kärkkäinen, J., Sanders, P. and Burkhardt, S. (2006). Simple linear work suffix array construction. *Journal of ACM* **53** (6) 918–936.

Kim, D. K., Kim, M. and Park, H. (2008). Linearized suffix tree: An efficient index data structure with the capabilities of suffix trees and suffix arrays. *Algorithmica* **52** (3) 350–377.

Kim, D. K. and Park, H. (2005). A new compressed suffix tree supporting fast search and its construction algorithm using optimal working space. In: Apostolico, A., Crochemore, M. and Park, K. (eds.) CPM. *Springer Lecture Notes in Computer Science* **3537** 33–44.

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

Ko, P. and Aluru, S. (2005). Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms* **3** (2–4) 143–156.

Lin, J., Jiang, Y. and Adjeroh, D. A. (2009). The virtual suffix tree. *International Journal of Foundations of Computer Science* **20** (6) 1109–1133.

Loebl, M. and Nesetril, J. (1988a). Linearity and unprovability of set union problem strategies. In: Simon, J. (eds.) STOC, ACM 360–366.

Loebl, M. and Nesetril, J. (1988b). Postorder hierarchy for path compressions and set union. In: Dassow, J. and Kelemen, J. (eds.) *IMYCS*, *Springer Lecture Notes in Computer Science* **381** 146–151.

Lucas, J. M. (1990). Postorder disjoint set union is linear. *SIAM Journal on Computing* **19** (5) 868–882.

Manber, U. and Myers, E. W. (1993). Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* **22** (5) 935–948.

McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Journal of ACM* **23** (2) 262–272.

Navarro, G. (March 2014) Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys* **46** (4) 52:1–52:47.

Navarro, G. and Mäkinen, V. (2007). Compressed full-text indexes. *ACM Computing Surveys* **39** (1).

Ohlebusch, E. (2013). *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*, Oldenbusch Verlag.

Ohlebusch, E. and Gog, S. (2009). A compressed enhanced suffix array supporting fast string matching. In: Karlgren, J., Tarhio, J. and Hyyrö, H. (eds.) SPIRE, *Springer Lecture Notes in Computer Science* **5721** 51–62.

Puglisi, S. J., Smyth, W. F. and Turpin, A. H. (2007). A taxonomy of suffix array construction algorithms. *ACM Computing Surveys* **39** (2) 1–31.

Russo, L. M. S., Navarro, G. and Oliveira, A. L. (2011) Fully compressed suffix trees. *ACM Transactions on Algorithms* **7** (4) 53.

Sadakane, K. (December 2007). Compressed suffix trees with full functionality. *Theor. Comp. Sys.* **41** (4) 589–607.

Schieber, B. and Vishkin, U. (1988). On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing* **17** (6) 1253–1262.

Szpankowski, W. (1993). A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM Journal of Computing* **22** 1176–1198.

Szpankowski, W. (2001). *Average Case Analysis of Algorithms on Sequences*, Wiley Series in Discrete Mathematics and Optimization, Wiley-Interscience.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica* **14** (3) 249–260.

Weiner, P. (1973). Linear pattern matching algorithms. In: *SWAT (FOCS)*, IEEE Computer Society, 1–11.