HAL archives-ouvertes.fr

On the longest common factor problem

Maxime Crochemore, Alessandra Gabriele, Filippo Mignosi, Mauriana Pesaresi

▶ **To cite this version:**

Maxime Crochemore, Alessandra Gabriele, Filippo Mignosi, Mauriana Pesaresi. On the longest common factor problem. 5th IFIP International Conference on Theoretical Computer Science (TCS'08), Sep 2008, United States. pp.143-155. hal-00620280

HAL Id: hal-00620280

https://hal-upec-upem.archives-ouvertes.fr/hal-00620280

Submitted on 14 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Longest Common Factor Problem

Maxime Crochemore[1], Alessandra Gabriele[2], Filippo Mignosi[3], Mauriana Pesaresi[4]

[1] King's College London, U.K. and Université Paris-Est, Institut Gaspard-Monge, France maxime.crochemore@kcl.ac.uk
[2] Dipartimento di Matematica e Applicazioni, Università di Palermo, Italy sandra@math.unipa.it
[3] Dipartimento di Informatica, Università dell'Aquila, Italy mignosi@ns.di.univaq.it
[4] Dipartimento di Informatica, Università di Pisa, Italy pesaresi@di.unipi.it

Abstract. The Longest Common Factor (LCF) of a set of strings is a well studied problem having a wide range of applications in Bioinformatics: from microarrays to DNA sequences analysis. This problem has been solved by Hui (2000) who uses a famous constant-time solution to the Lowest Common Ancestor (LCA) problem in trees coupled with use of suffix trees. A data structure for the LCA problem, although linear in space and construction time, introduces a multiplicative constant in both space and time that reduces the range of applications in many biological applications.
In this article we present a new method for solving the LCF problem using the suffix tree structure with an auxiliary array that take space $O(n)$. Our algorithm works in time $O(n \log a)$, where n is the total input size and a is the size of the alphabet.
We also consider a different version of our algorithm that applies to DAWGs. In this case, we prove that the algorithm works in both time and space proportional to data DAWG's size.

1 Introduction

In *1976* E.M.McCreight settled a Kunt's open problem by introducing a new data structure on string: the *Suffix Tree*. Since then, many other problems have been settled by using suffix trees or similar structures such as Patricia trees, DAWG, CDAWG and suffix array (cf. for instance [2,8,7,5,14] and references therein). Some other applications can be retrieved by exploring the "Pattern Matching Pointers" maintained by S. Lonardi (cf. [13]).

The most commonly used data structures are *Suffix Trees, Suffix Arrays, DAWGs,* and *CDAWGs*. Usually any problem that can be settled by the aid of one of such data structure can also be settled by using any of the other ones. Despite this fact the passage from one data structure to another is not automatic nor always easy and, in some rare cases, not yet proved (see [1] for example). Each of these structures has some advantage and some disadvantage.

Some relation among the data structures and their size is reported in [3]. The size of an implementation of the above data structures is often evaluated by the average number of bytes necessary to store one letter of the original text. It is commonly admitted that these ratios are 4 for suffix arrays, 9 to 11 for suffix trees, and 5 for CDAWGs (cf. [3] for further information).

This paper deals with particular data structures: DAWGs.

The problem we consider is reported by D. Gusfield, [8, Sec. 7.6, 9.4]. Given a set of m strings, for any $k = 2, .., m$ find the longest factors that are common to at least k strings. The word *common* in the exact case means *occurring with equality.* The first solution in the exact case has been given by Hui, ([10], [11]). who uses a famous constant-time solution to the Lowest Common Ancestor (LCA) problem in trees coupled with the use of suffix trees (see [9,16,6]). A data structure for the LCA, although it is linear in space and time, introduces a multiplicative constant in both space and time that reduces the range of applications in many biological applications.

Since DAWGs and CDAWGs are not trees, this solution cannot be used for the structures we are interested in. Therefore we look for a totally new solution. So, our solution turns out to be simpler and more efficient than Hui's one of about one order of magnitude. This solution is an extension from that of suffix trees to DAWGs.

This paper is organized as follows. In the next section we describe our solution for the problem based on the use of suffix trees, while in the Section 3 we extend our solution to DAWGs. The fourth section contains our conclusions and some conjectures on the approximate case of the problem. Hence in the Appendixes A and B, we report the specialized pseudo-code related to the procedures used in our algorithm.

2 A Simpler Solution

We assume the reader familiar with suffix trees and Generalized Suffix Trees.

Let S be a set of input strings S_i, $1 \leq i \leq m$, on the alphabet $\{0, 1\}$. Let u be the word composed of the concatenated labels of transitions along the unique path from the root to the node p in the Generalized Suffix Tree.

We want to compute a table ℓ having $m-1$ entries: where entry $\ell[k]$ provides the length of the longest factor common to at least k of the input strings and also points to one of the common factors having that length.

Our preprocessing is as follows. We build the Generalized Suffix Tree for the m strings. Then perform a depth-traversal of the tree and put all nodes in a stack in the order they appear. Define s to be an array of pointers representing the input strings useful to increase the algorithm's performances.

Each node stores the following information:

- i represents the string identifier whose suffix is the node path-label. If this is not a suffix, this field is empty.
- *num* is the number of distinct string identifiers that appear at the leaves in the subtree rooted in p. Observe that this approach is the same as the one

used by Gusfield in [8, Sec.7.6]. The difference lays in how to compute these values, that he calls $C[p]$.

So we must first compute the *num* values and then we use them to update the table.

2.1 Computing the *num* values

For each node p, we create an auxiliary node *size* that stores the values $num(p)$ and points to the strings it represents in s.

When for the vertex p we have $num(p) = b$, this means that in its subtree there are nodes representing suffixes from b different input strings. In other words, p is the common factors of exactly b different input strings. In the algorithm we call these nodes representative in the operation of $Union$ that plays an important role in the computing of our values.

The operation $Union$ is the union between disjoint sets of elements that, in our case, are nodes *size* linked to visited nodes. All pointers to auxiliary node of smaller size must point to the other node *size* and, naturally, we must also update the sizes of the involved nodes, i.e. the field *num*.

$Union$ operates as follows. Let a be a node with $num(a) = 2$ and let b with $num(b) = 3$. When we visit the a and b's father p, we execute a $Union$ of his children. The result is that $num(p) = 5$ and the p's label becomes a common factor of 5 input strings.

We keep the disjoint strings sets as follows. We use an array s of m pointers that represent the input strings and for which $s[i]$ points permanently to the last met factor of the string s_i. Since the last factor of a string is unique, the sets to merge are always disjoints.

In the algorithm we use three procedures called $NodeSize\ Test$, $String\ Test$ and $Union$ (that implements the union's operation). Now we explain how they work, while in the Appendix A we show the code of them.

- $NodeSize\ Test$ procedure: we check if the node *size* is already created. If not, we create it.
- $String\ Test$ procedure: when we visit a new node, we must update the information about the last visited factor of some string. Note that after this test and related "cut-append" of pointers, node *size* stores the current *num* value, while the internal node stores the real one. Because nodes *size* are representatives in the $Union$, then they must be updated in every time. In Figure 1 we show the effect of this test.
- $Union$: after we have found the smallest son its pointer is redirected to the largest one, the *num* value is updated, and the new node *size* resulting from the merging is merged with the father node *size*.

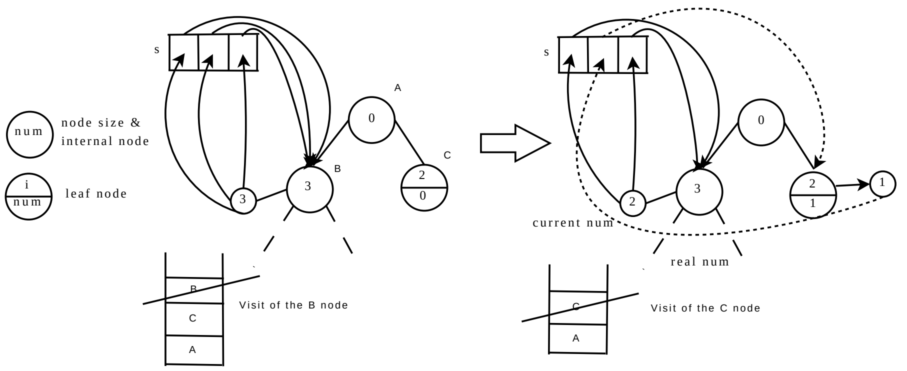

Fig. 1: In the left figure, is shown the situation after the visit of the B node, at the top of the stack. It's the common factor of all input strings. Then the node C is traversed. Therefore it's a leaf node representing the string s_2, then the algorithm "cut" the pointer from B's node *size* to the element $s[2]$, appending it to C's node *size*. In fact, the last factor of s_2 is the path-label of C. Observe that the values about the size of the nodes are updated during this test.

Now we describe the algorithm to compute in an efficient way the *num* information.

```
CountNum (stack, s)
while (stack isNotEmpty) do
        p = pop(stack);
        NodeSize test;
        String test;
        if p has sons then
                Union operation;
End CountNum.
```

2.2 The method

Once the *num* values are known and the string-depth of every node is known, the desired $\ell(k)$ values can be easily found with a linear-time traversal of $GST(S)$.

When encountering a node p with $num(p) = k$, we compare the string-depth of p to the current value of $\ell(k)$. If the first value is greater than the second, we change $\ell(k)$ to the depth of p and update its pointer to the node representing the factor with the current value of $\ell(k)$.

Eventually, the resulting table holds the desired $\ell(k)$ values.

2.3 Time and space analysis

Building $GST(S)$ requires linear space and time in the size of input [8, Sec. 6.4], $O(n)$ with $n = |S_1| + \cdots + |S_m|$. Algorithm $CountNum$ executes a single post-order traversal of $GST(S)$ and its main operation is the $Union$. Since the operation "cut-append" of a pointer from a node to another is done in constant time, then we have to know how many pointers could be involved during it.

Theorem 1. ***[CountingNum]*** *During the execution of $CountNum(S, s)$ algorithm, the number of "cut-append" operations is less than n, with $n = |S_1| + \cdots + |S_m| = |S|$.*

Proof. The statement is proved by induction on the total size n of the representatives u of the nodes p. Recall that u is p's representative if it is the concatenated labels of transitions of the unique path from the root to the node p in $GST(S)$.

Basic step: let m be the minimal size of S. The root points directly to the m leaves. Therefore the number of "cut-append" operations is equal to $m-1$: we append all auxiliary leaves to the root. Since the input's size is equal to m then our thesis is proved.

Inductive step: by induction we suppose that our thesis is true for every tree with representatives' size equal to $n-1$.
We prove the thesis is true at the level n.
Let our visit be stopped in a node with two sons. The first subtree has NL_1 leaves and representatives' size equal to n_1, while the second has NL_2 leaves and size equal to n_2.
When we get to the bottom level n, we add a character to every representative for each leaf. So the total representatives' size of the level n is equal to $n_1 + NL_1 + n_2 + NL_2$.
The $Union$ simply appends all leaves of one subtree to the other subtree.

$$\begin{aligned} cut - append &= NL_1 + NL_2 < \\ &< n_1 + NL_1 + n_2 + NL_2 = n. \end{aligned} \tag{1}$$

When we visit the $GST(S)$'s root, the total representative's size is equal to the input length, n. Hence, the total number of $Union$ operations is linear in the input length. ■

During a run of the algorithm $O(n)$ "cut-append" operations are executed, each of which takes constant time, so the overall $Union$ takes $O(n)$ time.

Hence only $O(n)$ time is needed to execute the algorithm and to compute all num numbers. Once these are known, only $O(n)$ additional time is needed to build the output table.

Hui's solution take $O(mn)$ time because it uses an array of k elements for each node of the tree to calculate the num values. We solve the original problem simply using $O(m+n)$ space, because the algorithm makes use of a unique array.

Theorem 2. *The Lowest Common Factor Problem on a set of m input's strings, represented by a Generalized Suffix Tree, can be solved in $O(n)$ time, with $n = |s_1| + \cdots + |s_m|$, and $O(m+n)$ space.*

3 An Optimal Solution

In this section we deal with the data structures that plays an important role in this paper, the Generalized Directed Acylic Word Graph (Generalized DAWG). We assume the reader familiar with DAWGs.

Now we recall the definition of DAWG.

Definition 1. *The DAWG for a set of strings $s_1, \cdots, s_m$ is a directed acyclic graph, with a node marked as initial and m distinct nodes $F_1, \cdots, F_m$ marked as final. Edges are labeled with non empty factors of at least one of the strings. Labels of two edges leaving the same node cannot begin with the same character. For every string s_i in the set, all suffixes of s_i are spelled by patterns starting at the initial node and ending at node F_i. Paths ending at non final nodes correspond to strict classes of factors of the congruence relations $\equiv_{Suf(S)}$.*

Let S be our input set of strings.

We want to analyse the meaning of the state u in terms of "representative". In $DAWG(S)$ there are more edges entering the same state than in the corresponding tree, according to Def.1. So we define the *representative* of a state as the longest path from the initial state to it.

Like for Suffix Trees, we want to compute a table that gives for entry k the length of a longest factor common to at least k strings and also points to it.

Now our preprocessing is less easy than in the previous section because more paths are not distinct. We build a Generalized DAWG for the m input strings. Each final state represents an input string (e.g., s_i) and is marked with a non null identifier (e.g., i).

Observe that in a DAWG two or more outgoing edges from the same state could finish in the same state and so we would like that the path from an internal state to other one is unique. Hence we keep only the representative of a factor's class. Since to solve the LCF Problem we need the longest labels of the paths, we keep only the transitions with the longest labels and we delete all other ones that have the same origin and target states. In this way the number of transition is drastically reduced and we obtain a pruned DAWG, denoted by D, having a deterministic transition function between adjacent states.

Now we are ready to perform a particular breadth-traversal of the new structure to store all states in a stack, in a way that is similar to the procedure done on suffix trees. We put nodes in the stack in the order they appear. Our problem is that we traverse some nodes more times and we must store them only once. Hence, if a node is already stored, we delete its previous occurrences, we put its new occurrence and we increase a counter related to the node. Define s to be an array of pointers representing the input strings like above. In our data structure each state stores the following informations:

- i is the string identifier whose suffix is the state path-label,
- num is the number of distinct string identifiers that appear in the subgraph rooted in the state,

– *count* is the counter mentioned above.

As in the previous section, we first compute the *num* values and then we use them to update the output's table.

3.1 How to compute desired values?

The algorithm to calculate *num* is almost the same as for suffix trees. The only difference is in the *String Test* procedure, because here there is another parameter to check, the *count* value. In the algorithm we use three procedures called *NodeSize Test* procedure, *StringD Test* procedure and *UnionD*. *NodeSize Test* procedure has already been described in the previous section.

Now we explain how the *StringD* test works, while in the Appendix B we show its code and the *UnionD* one. First we perform the following test on the field *count* of the sons of the current state:

– if *count* is not null for some son, we decrease the value of *count* and we "cut" only the pointer from array s to the previous state to link it to the actual node, because this one represents the last factor of the interesting string. Observe that we delete a node *size* when *count* become null. So, for *count* times we must replace the node *size*. This fact causes an additional extra-space but it permits to perform the execution in linear time;
– otherwise we call the classical *String* test.

The complete algorithm is the following:

```
CountNumBIS (stack, s)
 1. while (stack isNotEmpty) do
 2.        p = pop(stack);
 3.        NodeSize test;
 4.        StringD test;
 5.        if p has sons then
 6.              UnionD operation;
 7. End CountNum.
```

3.2 Building the output table

Once the *num* value and the string-depth of every state are known, the desired $\ell(k)$ values can be easily found with a linear-time traversal of D.

When encountering a state p with $num(p) = k$, the string-depth of p is compared to the current value of $\ell(k)$ and if the first one is greater than the second, $\ell(k)$ is changed to the depth of p and its pointer updated to the node representing the factor with the current value of $\ell(k)$.

Finally the resulting table holds the desired $\ell(k)$ values.

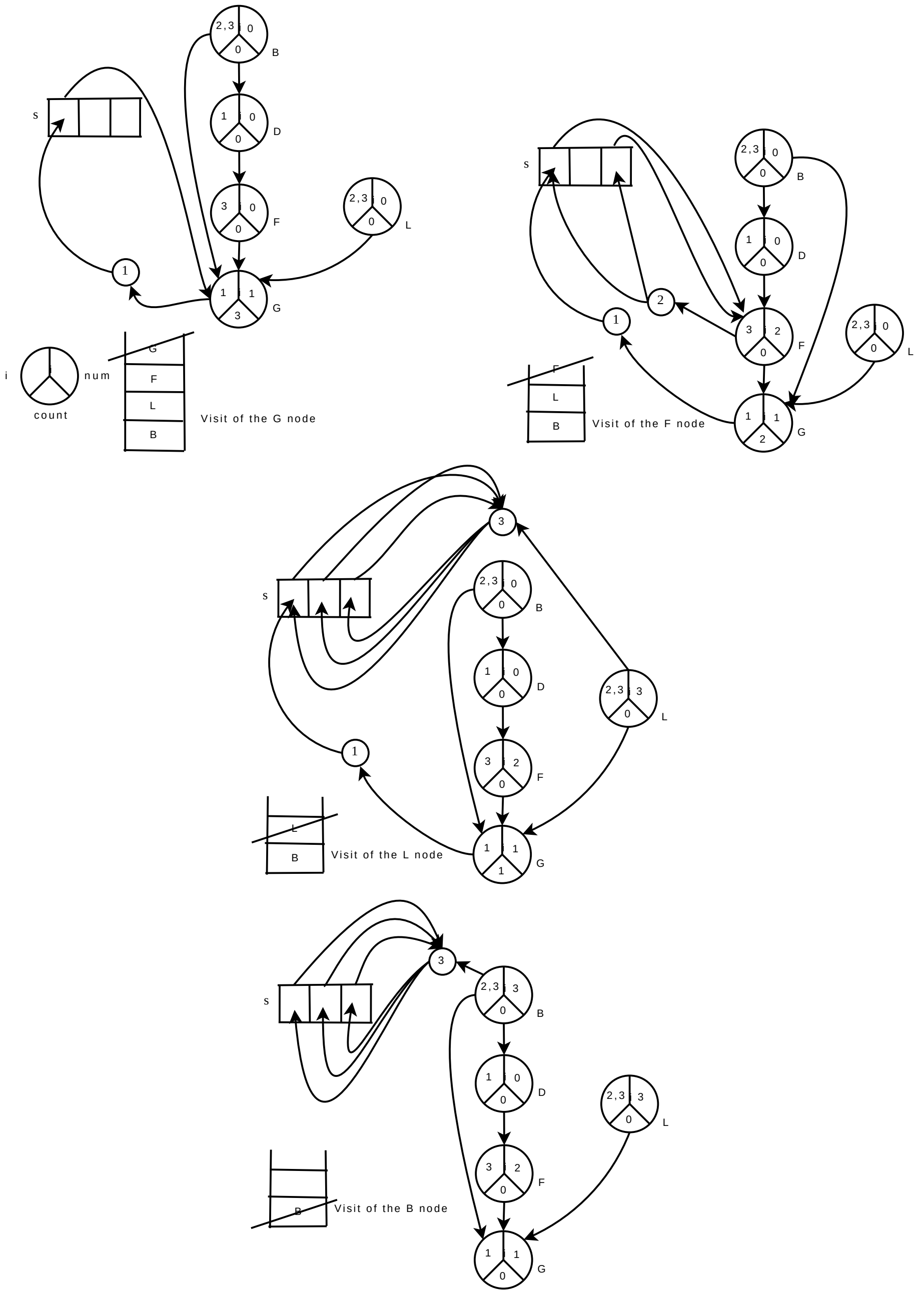

Fig. 2: How *StringD* works. Visiting the state G is the same as visiting the corresponding tree node. When we traverse state F, we create a duplicate pointer to $s[1]$ not to lose the information related to state G: in fact, two other edges arrive in this state and they need to know that G is a suffix of $s[1]$. Note that the field *count* of state G is decreased. Therefore F is also a suffix of $s[2]$, then we perform a traditional *Union*. The same happens when traversing states L and B. In last case, since the field *count* of state G is null, then we can delete its node *size* because we have visited all its neighbors.

3.3 Time analysis

Let n the input size, with $n = |S_1| + \cdots + |S_m|$. Building $DAWG(S)$ requires linear time in the input size as described in [12].

Algorithm $CountNumBIS$ executes a single traversal of D and its main operation is $Union$. Since the operation "cut-append" of a pointer from a node to another is constant, then we would like to know how many pointers could be involved during it.

Let D be the Generalized $DAWG$ over S. We can use a breadth-first visit of D to re-create the original Suffix Tree. Each path from initial state to a final state in $DAWG$ is used to build a path from the root to a leaf in the Suffix Tree. Note that the technique is the same as McCreight's one (cf. [15]) to create suffix trees directly from input's strings.

After this traversal, we have created a suffix tree with a number ns of nodes that is larger than the number nc of $DAWG$ states, with same edges and related labels. Hence representatives of suffix tree states are the same as that of $DAWG$s.

Since $nc \leq ns$, from Theorem 1, we have the following result:

Theorem 3. ***[LCSS Counting Bis]*** *During the execution of* $CountNumBIS(S, s)$ *algorithm, the number of "cut-append" operations is less than* n*, with* $n = |S_1| + \cdots + |S_m| = |S|$*.*

During the run of the algorithm there are $O(n)$ "cut-append" operations executed, each of which takes constant time, so all $Union$ executions take $O(n)$ time in total.

Hence only $O(n)$ time is needed to execute the algorithm and to compute all num_S numbers. Once these are known, only $O(n)$ additional time is needed to build the output table.

Finally, we can state:

Theorem 4. *The Lowest Common Factor Problem on a set of* m *input strings, represented by a Generalized Directed Acyclic Graph, can be solved in* $O(n)$ *time, with* $n = |s_1| + \cdots + |s_m|$*, and* $O(m + n)$ *space.*

4 Conclusions

Recent experiments [4] have showed that DAWGs are space thrifty not only in exact problems, but also in the approximate cases, where some "errors" or "faults" are allowed. To build the approximate DAWG of a word in optimal time remains an open problem. Now, we think that our solution of exact problem can be applied to these data structures to solve the approximate one. If the conjecture reported in [4] is true and if it possible to build approximate DAWGs in optimal time, then our solution will drastically outperform previous solutions.

References

1. M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.
2. A. Apostolico. The myriad virtues of suffix trees. In A.Apostolico and Z.Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *F*, pages 85–96. 1985.
3. M. Crochemore. Reducing space for index implementation. *Theoretical Computer Science*, 292(1):185–197, 2003.
4. M. Crochemore, C. Epifanio, A. Gabriele, and F. Mignosi. On the suffix automaton with mismatches. In *To appear in Lecture Notes in Computer Science*. CIAA'07, 2007.
5. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte*. Vuibert Informatique, 2001.
6. J. Fischer and V. Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *Springer LNCS*, volume 4009 of *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching(CPM'06)*, pages 36–48, 2006.
7. R. Grossi and G.F. Italiano. Suffix trees and their applications in string algorithms. Proceedings of the 1st South American Workshop on String Processing, pages 57–76, 1993.
8. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
9. D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13:338–355, 1984.
10. L.C.K. Hui. Color set size problem with applications to string matching. In *Proceedings 3rd Symposium on Combinatorial Pattern Matching*, volume 644 of *Springer LNCS*, pages 227–240, 1992.
11. L.C.K. Hui. A practical algorithm to find longest common substring in linear time. *International Journal of Computer Systems Science & Engineering*, 15(2):73–76, 2000.
12. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. In *Discrete Applied Mathematics*, volume 146 of *12th Annual Symposium on Combinatorial Pattern Matching*, pages 156–179, 2005.
13. S. Lonardi. http://www.cs.ucr.edu/∼stelo/pattern.html.
14. M.G. Maas. Matching statistics: efficient computation and a new practical algorithm for the multiple common substring problem. *Software Practice and Experience*, 36:305–331, 2006.
15. E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
16. B. Schieber and U. Vishkin. On finding lowest common ancestors: simplifications and parallelizzation. *SIAM Journal on Computing*, 17:1253–1262, 1988.

A Appendix

Now we detail the procedures used by the algorithm for Suffix Tree. Recall the data structures in a formally way.
The auxiliary structure s is an m-array of pointers.
The node of $GST(S)$ are formed by three fields (and not two):

- the fields i and num;
- the field ns is a pointer to the node $size$ related to our node.

The node size has two fields (and not one):

- the field num;
- the field ns is a set of pointers to the structures s, one for each string that the node representing.

NodeSize test ($GST(S)$, s)
if $p.ns = Nil$ **then**
    $p.ns$ =new node.
Return($GST(S)$, s).

String test ($GST(S)$, s)
for *every i of p* **do**
    if $p.i! = Nil$ **then**
        $s[p.i].ns.num - -$
        if $s[p.i].ns.num = 0$ **then**
            delete $s[p.i].ns$;
    $s[p.i] = p$;
    $p.ns.ns = s[p.i]$;
    $p.ns.num + +$;
    $p.num + +$.
Return($GST(S)$, s).

Union ($GST(S)$, s)
Merge between nodes $size$
    merge between pointers to s;
    sum between the fiels num;
    have created a new node $size$ m;
$p.ns = \text{merge}(p.ns, m)$;
$p.num = p.ns.num$.
End $Union$ operation.

B Appendix

Now we detail the procedures used by the algorithm for DAWG. Recall the data structures in a formally way.
The auxiliary structure s is an m-array of pointers.
The node of $DAWG(S)$ are formed by four fields (and not three):

- the fields i, num and $count$;
- the field ns is a pointer to the node $size$ related to our node.

The node size has two fields (and not one):

- the field num;

– the field ns is a set of pointers to the structures s, one for each string that the node representing.

NodeSize test (D, s)
if $p.ns = Nil$ **then**
$p.ns$ =new node.
Return(D, s).

StringD test (D, s)
for *every i of p* **do**
if $s[p.i].count! = 0$ **then**
$s[p.i].count - -$;
if $s[p.i].count = 0$ **then**
delete $s[p.i].ns$;
$s[p.i] = p$
else
$s[p.i].ns.num - -$
if $s[p.i].ns.num = 0$ **then**
delete $s[p.i].ns$;
$s[p.i] = p$;
$p.ns.ns = s[p.i]$;
$p.ns.num + +$;
$p.num + +$.
Return(D, s).

UnionD (D, s)
Merge between nodes *size* of p's sons with the field *count* null
merge between pointers to s;
sum between the fields *num*;
have created a new node *size* m;
if $q.count! = 0$ *and* $q.ns.ns! = p.ns.ns$ *with q son of p* **then**
$q.count - -$;
if $q.count = 0$ **then**
delete $q.ns$;
duplicate $q.ns$ pointers and append to m;
$m.num = m.num + +$
$p.ns = \text{merge}(p.ns, m)$;
$p.num = p.ns.num$.
End *UnionD* operation.