Suffix Tree Construction with Slide Nodes

Mario Huerta[1]
mhuerta@lsi.upc.es

[1] Departament de llenguatges i Sistemes, Universitat Politècnica de Catalunya.

Abstract

The Suffix-tree is one of the most powerful and versatile structures in the string-matching area. For this structure there are three classical O(n)-construction algorithms: Weiner's algorithm, McCreight's one and E. Ukkonen's one. This study shows a review of the construction algorithm that, while maintaining Ukkonen's on-line property, mainly represents an improvement of two of their non-functional characteristics: efficiency and usability. The new algorithm reduces construction time (around 40% in practice), the query time and also the space taken up by the structure, allowing working with bigger size sequences. To achieve that, the algorithm features a new and more useful design of its *suffix-links* structure. This new design will provide a greater independence among the *suffix-links*, which leads to a greater locality for the suffix-tree sub-trees (a useful example will be shown later). In addition, the resulting algorithm is far more understandable, making easier to modify it when dealing with the resolution of concrete problems.

Keywords: suffix tree construction algorithm, string matching.

1 Introduction

Suffix-tree is a trie which contains all the suffixes in a given S sequence which are accessible from the root in an O(1) time. Suffixes with common prefixes will share such prefix in the tree (fig 2). This is one of the most powerful and versatile structures used for solving string-matching related problems [Gus 97].

However, there are several alternatives to suffix-tree: the *Knuth-Morris-Pratt* [KMP77] and the *Boyer-Moore* [BoMo77] iterative algorithms, the *Aho-Corasaick* [AhCo75] algorithm and structure, which is specially useful for solving dictionary-type problems, the *inverted-lists* and the *signature-files* [ZMR98], widely used in the World Wide Web but both unable to work with unstructured texts, and the *suffix-array* [MM93],[Sad97],[CrFe99], [JlarSad99], which was for a long time one of its more immediate rivals. Although it doesn't feature the suffix tree linear time, with a O($n$log$n$) in the worst-case scenario, their construction time is in practice similar[JlarSad99] and in many of the queries it improves the time achieved by suffix-tree *tree* [MM93], [AKO02], [AOK02].

There are other structures that are an evolution from suffix-tree: the *suffix-cactus* [Kär95], a mixture between suffix-tree and suffix-array, and CDAWG[CreVer97],

compact-direct-aciclic-word-graf, in which, in addition to the prefixes, the common suffixes are shared too unless it would create a circle.

The first person that constructed the suffix-tree was P. Weiner in 1973 [Wei73], but in 1975 M. McCreight presented an algorithm much more efficient than the P. Weiner's one [McC75], and the last person that created a construction algorithm for the classical suffix-tree was E. Ukkonen in 1995 [Ukk95]. This latest algorithm didn't offer more efficiency than the McCreight's one, but it offered an easier understanding and it also had the power of being on-line. This easier understanding, and the fact of have been widely exposed in the D. Gusfield's work [Gus97]made that, in spite of being slightly less efficient in practice than M.McCreight's algorithm[GiKu97], it has become nowadays the most widely used and spread construction algorithm.

Some partial changes of suffix-tree have appeared lately seeking a greater efficiency: the optimizations in space introduced by S. Kurtz [Kur99], the *sparse-suffix-tree* [KU96], the lazy execution algorithms [GKS99], the utilization of external memory [ClMu96], the utilization of compressed data structures [GrVi00], whereas some versions of parallel construction algorithms [AILSV88],[Har94].

Other suffix-tree versions try to support new data types. In 1997, Farach [Far97] constructs a suffix-tree for strings using large alphabets (suffix-arrays will also be focused on this problem). Anderson et al [ALS99] assumes tokens that divide text in words we can deal with in an individual way. Baker[Bak93] generalizes the suffix-tree for parameterized-strings[CoHa00]. Giancarlo[Gian95] expands the use of suffix-tree to support two dimensional texts [KP99][CoHa00].

We won't leave by now the line of most common problems [Apos85],[Gus97], and we'll analyze the suffix-tree new design, with its respective construction algorithm, and we'll discuss examples of applications for the new chances it offers to us. We'll formally compare such algorithm and structure with the E. Ukkonen and M. McCreight's algorithms, and we'll remark the good results it provides in practice.

2 Constructing the suffix tree

A suffix tree is a trie which contains all the suffixes in a given text that are underhand by their prefixes and accessible all of them from the tree's root.

Now we'll remark the crucial points of the *suffix-tree* construction. In the course of this exposition we'll use the following terminology: ε will represent the empty string. Σ will represent the alphabet of the provided text. Σ^* will represent the set of strings on the alphabet Σ. $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. $t \in \Sigma^*$ will be the provided text. The t_i-prefix will be the $t_1...t_i$ substring of t and the t_i-suffix the substring $t_i...t_{|t|}$..T will be a Σ^+-*tree* containing suffixes of *t.* And finally, the *cst(t)* will be the *compact- suffix- tree* of *t.*

2.1 The canonical representation

E. Ukkonen introduced the concept *canonical form* in order to point a specific place in the tree.

Definition 1. Being $\underline{b}$ and $\underline{v}$ two tree nodes in which the second one is the child of the first one, the *parent-child* relationship between both two will be described as the transition $\underline{b} \xrightarrow{(l,r)} \underline{v}$, where the pair of indexes (l,r) represent the *label* of the $t_l...t_r$ substring of t for the *transition string* covered by the transition from $\underline{b}$ to $\underline{v}$. We will also refer to $\underline{b}$ as the *previous node* of the transition and to $\underline{v}$ as the *subsequent node* of the same transition. We'll tell that a path or string that exists in T and can be covered from the node $\underline{b}$ is *accessible* from $\underline{b}$.

Definition 2. Let be $\underline{b}$ a T node and u a string in T which is accessible from the $\underline{b}$ node, we will define the canonical pair (b,u) as the point in T to which we will access after covering the u string from the $\underline{b}$ node. This T point will be a t index that will provide the character to that point.

In the E. Ukkonen algorithm [Ukk95], the *canonize()* function is the responsible for covering the canonical pair suffix from the pair node until reaching an irreducible canonical pair, that is a canonical pair whose suffix doesn't reach the end of the transition string of the last transition, or a canonical pair whose suffix is fully consumed. If the suffix was totally consumed, the *canonize()* function would return back the empty sequence ε as the suffix of the non-reducible pair.

2.2 The *suffix-link*s

*Suffix-link*s are links between tree nodes that make possible crossing it transversely, avoiding that way to cover the common prefix between the new suffix to be inserted and the previously inserted in T ones [McC75].

Definition 3. We'll define f as the structure that, after being consulted for a concrete node $\underline{b}$ returns the node $\underline{b'}$ linked by $\underline{b}$ by means of a *suffix-link.* Let be $\underline{b}$ an internal node crossed by the suffix $w = cps$ of t, in which c is its first character, $f(\underline{b})$ will be an internal node crossed by the suffix $w' = ps$ of t, being s a accesible suffix from b as from $f(\underline{b})$.

Definition 4. Let w be an accessible string from a node $\underline{b}$, according to the *suffix-link* definition (Def. 3), it will be accessible a string w' from $f(\underline{b})$, which is identical to w. We'll name the path covered by w' the *equivalent path, equivalent node,* the node in the canonical pair after covering w' from $f(b)$, and *equivalent point,* the point where this non-reducible pair will point in T, which corresponds with the last character of w'.

Lemma 1. The strings that are accessible from the last character of a path $w \in$ T are also accessible from its equivalent point.

Note that $\underline{b}$ and $f(\underline{b})$ doesn't share a bijective relationship, therefore the lemma 1 doesn't involve that any string accessible from $f(\underline{b})$ is also accessible from $\underline{b}$.

2.3 The root's *suffix-link* (dismissing the first character problem)

If we place ourselves on the root node it doesn't exist any prefix that, if we follow using *suffix-link,* we won't have to cover. Going back to Def.3, we could affirm that $p = \varepsilon$, so the t suffixes $w = cps$ and $w' = ps$ would respectively be the suffixes $w = cs$ and $w' = s$. cs would then be the accessible suffix from the *root* node and s would be the accessible from $f(\underline{root})$. Therefore, the equivalent position to w_i in w' would be the position w'_{i-1}, being $|w'| = |w|-1$.

This fact causes an outstanding condition, and until nowadays there have been exposed two treatments to deal with it.

The first one entails maintaining the *suffix-link* f(root) = root and detecting the moment in which we take the *suffix-link* of the root node. Once is detected the jump from the root node, we would manually dismiss the first character on the equivalent path we want to cover. This is the option chosen by M. McCreight[McC75] and D. Gusfield[Gus97].

The second option, the one used by our algorithm and the E. Ukkonen[Ukk95]'s algorithm, would entail using a $\perp$ node previous to the root node, and that would be linked by the root node by *suffix-link* (f(root) = $\perp$). This $\perp$ node would have $\forall c \in \Sigma$ a transition to the root node with a transition string of length one. Then it would use this set of transitions to automatically dismiss the first character on the equivalent path when it's taken the *suffix-link* from the root node.

Due to these two measures we are able to correct the character of difference between a t suffix and the immediate lower length suffix. With the first measure the third definition of the *suffix-link* for the root's *suffix-link* won't be observed.

Once the *suffix-tree* is constructed, f doesn't use to be utilized for the queries. Further information on the *suffix-links* properties can be found at [McC75] , [Ukk95] and [Gus97].

2.4 Constructing operations

To construct the *suffix-tree* of t starting from an empty tree, we'll use two constructing operations[GiKu97].

Definition 5. Let be $(\underline{a}, w)$ a non-reducible canonical pair and $|w|=k+1-l$, we'll define the operation *split*($\underline{a}$, w) as the constructing operation that will replace the transition $\underline{a}^{(l,r)} \rightarrow \underline{b}$ in T with the transitions $\underline{a}^{(l,k)} \rightarrow \underline{new}^{(k+1,r)} \rightarrow \underline{b}$. To achieve this target it'll create a new node called *new*, it'll modify the string label to be covered by the old transition and it'll create a new transition from *new* to $\underline{b}$.

Definition 6. Let be s the t_i-suffix that we are trying to make accessible from the $\underline{a}$ node of the tree, we'll define *add*($\underline{a}$, s) as the constructing operation that will add to T a new transition $\underline{a}^{(i,\infty)} \rightarrow \underline{new}$ and the new leaf *new*.

Note that: the constructing operation *split* involves the destruction of a tree's transition. Performing a *split* operation only makes sense if it's associated to an *add* operation on the new node provided by *split*

We'll be then in front of two new kind of operations on the tree: the one that inserts a new transition string directly on the given node: *add-only*, and the one that needs to split a transition to concatenate this new string to an original transition string prefix: *split+add.*

When a new node is created as leaf node, it'll always be a leaf node [Gus97], because the constructing operation *split* always places the new node between two already created nodes (Def. 5), and the *add* operation is never performed on a leaf node(Def. 6).

The combination of jumps by *suffix-link*, the coverage of the equivalent path by means of *canonize()* and the application of the constructing operations *add* and *split* will allow to construct the definitive *suffix-tree* and be able to perform it in a linear time [McC75],[Ukk95].

3 The new data structures

One of the most immediate problems of the classical suffix-tree structure lies in the dispersion of the information it stores. This fact implies an increase of the required space and a slower navigation through the structure.

Our structure seeks for an O(1) access to all the required data from any node. To achieve this, in opposite of the classical structure, it places the string label for every transition in the subsequent node and modifies the structure of *f* linking by *suffix-link*s the subsequent nodes too.

This new structure will entail a set of changes:

i) the transition label must necessarily be accessible from the subsequent node. Note that origin and destination nodes of the *suffix-link* are the subsequent nodes.

ii) a new *split* operation will be necessary to keep fulfilling the classical *suffix-link* definition (Def. 3).

iii) the new *suffix-link*s will have properties that won't be shared with the ones from the classical structure.

iv) a new approach for the problem of the root's children will be necessary ($1.2)

i) the fact of placing the transition label at the subsequent node is part of our structure optimisation. We'll describe a transition with the new location as: $\underline{a} \rightarrow^{(l,r)} \underline{b}$

ii) a new *split* operation will be necessary

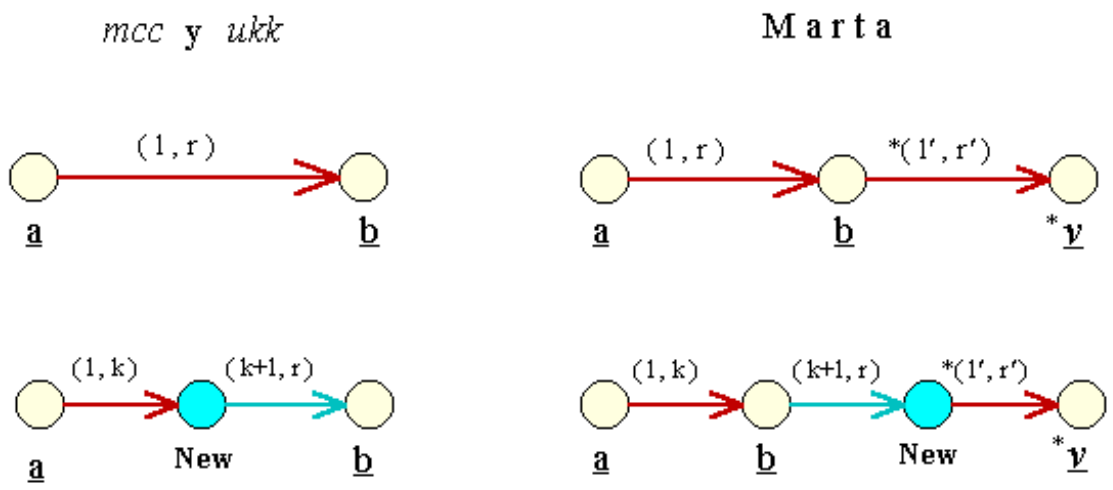

fig. 1 *Split* operation, on the left side operating on the old data structure, on the right side the new *split* operating on the new data structure.

Definition 7. Let be $(\underline{b},w)$ a non-reducible canonical pair and $|w|=k+1-l$, we'll define the new *split*($\underline{b}$, w) as the constructing operation that replaces the transition $\underline{a}\rightarrow^{(l,r)}\underline{b}$ in T by the two transitions $\underline{a}\rightarrow^{(l,k)}\underline{b}\rightarrow^{(k+1,r)}\underline{new}$. To achieve this, it'll create a new node $\underline{new}$, it'll modify the transition string covered by the old transition and it'll create a new transition from $\underline{b}$ to $\underline{new}$. The $\underline{new}$ node will inherit all the $\underline{b}$ children in the tree (fig 1.)

Theorem 1: The *suffix-link*s of the new structure *f* observe the Def.3 of any *suffix-link*
Proof: If we've got a suffix-tree with the new *f* design and we apply a split on a node $\underline{b}'=f(\ \forall\underline{b}\ \mid\ f(\underline{b})=\underline{b}')$ that makes that the transition passes from $\underline{c}\rightarrow^{(l,r)}\underline{b}'$ to $\underline{c}\rightarrow^{(l,k)}\underline{b}'\rightarrow^{(k+1,r)}\underline{new}$. It can be proven by induction that the paths initially accessible from *b'* will keep being accessible from *b'*. Therefore Def. 3 will be observed for any *suffix-link* of the new *f*.

iii) However, the *suffix-link*s of *f* will have properties that won't be shared with the *suffix-link*s of the classical structure.

Definition 8: Considering the $\underline{a}\rightarrow^{(l,r)}\underline{b}$ transition, in which $\underline{a}$ is $t_i..t_{l-1}$ is the string from the root to a, then for the $\underline{c}\rightarrow^{(l,rr)}f(\underline{b})$ transition:

i) $t_{i+1}..t_{l-1}$ is the string from the root to c
ii) the transition string $t_l..t_{rr}$ is a prefix of the transition string $t_l..t_r$.

Note that the cause of the existence of the point ii) is that after each new split the distance of a node with the immediately previous node in T is reduced. So we can say that as the T construction goes ahead, the nodes with their respective *suffix-link*s slide towards the root node. As a result of this behaviour we can't assure any more that a leaf node is always a leaf node.

Lemma 2: After successively splitting the equivalent path $t_l...t_r$ of the transition $\underline{a}\rightarrow^{(l,r)}\underline{b}$, in the $\underline{c}\rightarrow^{(l,k)}f(\underline{b})\rightarrow^{(k+1,...)}...\rightarrow^{(..,r)}\underline{z}$ way, $\underline{z}$ will be the equivalent node to $\underline{b}$ after following its *suffix-link*.

Note that neither the equivalent point nor the equivalent path have been modified.
The main issue of these new *suffix-links* lies in the fact that the set of nodes that share the immediately previous node in T have totally independent links. Therefore each of the transitions with a previous common node can be dealt independently (fig. 2) [HuMe02].

iv) How to approach the problem of *dismissing the first character*.

The suffix-tree will also be affected by the elimination of the ⊥ node previous to the root and of the transitions between this node and the root. In spite of the change we can approach the problem of *dismissing the first character* as the E. Ukkonen's algorithm uses the ⊥ node.

The ⊥ node is the previous node for the |Σ| transitions ⊥→root and the root node is its subsequent node. The root node is also the previous node of the root→(Σ-root's child) transitions too. Since the *suffix-links* of the new *f* link the subsequent nodes of the transition, the *suffix-link f(*root*)* = ⊥ will be replaced by the *suffix-links f*(Σ-root's child) = root. By placing the string label also in the subsequent node, the ⊥ node won't be useful any more and it'll be removed (fig.2)

The fact of placing the transition label at the subsequent node implies improvements in space, in the time spent covering the tree and a decrease of the required time for the *split* and *add* operations. The fact of placing the *suffix-links* at the subsequent node linking subsequent nodes implies a space loss of |*t*| words and an improvement in construction time. The use of the new constructing operation *split* doesn't imply any cost variation. We'll prove it all in the next section.

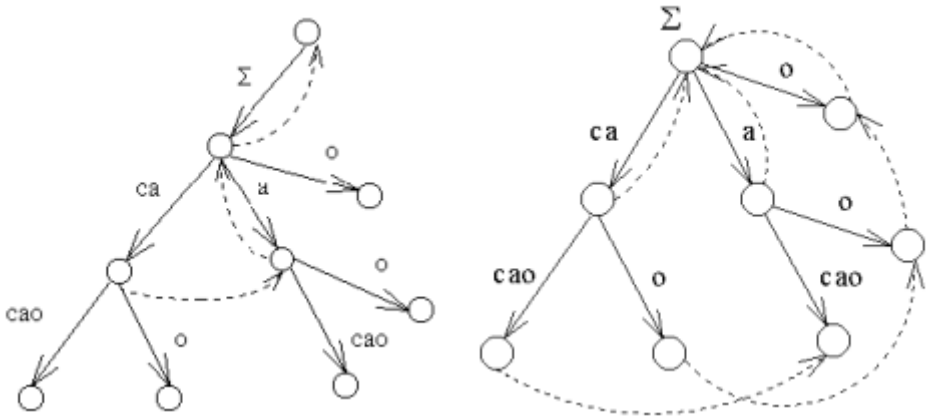

fig. 2. Suffix-tree for the string "cacao". On the left side, according to the old data structure, on the left side, according to the new data structure.

With our new structure we've overcome the problem of the dispersed information in the tree, and at the same time we've provided a greater locality to the sub-trees.As a result, unlike the algorithms that use the classical structure, we can provide our algorithms with a greater liveliness for the navigation through the structure, and this fact will ease obtaining better execution times in addition of making possible new approaches to the problems [HuMe02].

3 Comparing the use by the E. Ukkonen and M.McCreight algorithms between the classical suffix-tree structure and the new structure

Starting from the functional description of the E. Ukkonen (*ukk95*) and M.McCreight (*mcc75*) algorithms, which was introduced by G.Griegerich and J.Kurtz in [GiKu95] and [GiKu97], we'll perform the needed changes to be able to operate with the new data structure. Later we'll evaluate the cost of the changes in space and time.

3.1 Defining *ukk95*

Let be T the set of transitions between the tree nodes, in which those nodes get automatically defined because they make up a transition, and let be *f* the set of *suffix-links* between those nodes, the *ukk95* algorithm builds the *suffix-tree* of *t* inserting the suffixes for all the prefixes, from left to right. Therefore, for every step: T = *cst* (t_{i-1}-*prefix*).

3.1.1 The original *ukk95*

The *ukk95* algorithm is defined as:

The assignation of initial values for T and *f*.

$$T = \{\} \qquad (0.1)$$

$$\forall c \in |\Sigma|,\ T \cup \bot \xrightarrow{(0,0)} \text{root} \qquad (0.1.2.1)$$

where $\bot \xrightarrow{(0,0)} \text{root}$ is a *c*-edge (0.1.2.2)

$$f = \{\} \cup \{ (\text{root}, \bot) \} \qquad (0.2)$$

The *canonize()* function introduced in the previous section. It covers the equivalent path and returns the equivalent point [1].

$$\begin{aligned}
&\textit{canonize}\ (T, (\underline{b}, \varepsilon)) && = (\underline{b}, \varepsilon) && && (1.1)\\
&\textit{canonize}\ (T, (\underline{b}, cw)) && = (\underline{b}, \text{cw}), && \textbf{if } |w| < r - l && (1.2.1)\\
& && = \textit{canonize}(T, (\underline{v}, \textit{drop}(r - l, w))), && \textbf{otherwise} && (1.2.2.1)\\
& && \quad \textbf{where } \underline{b} \xrightarrow{(l,r)} \underline{v} \in T \text{ is a } c\text{-edge} && && (1.2.2.2)
\end{aligned}$$

[1] When *canonize()* returns the pair($\underline{b}$, ε), the first character of one of the $\underline{b}$'s childs (2.1.1) will constitute the equivalent point.

The *update()* function. In the recursive case ((2.1.3) and (2.2.2)), after updating T and *f,* performs a new call to the *update()* function to update T and *f* at the equivalent point.

$$update\,(T, f, (\underline{b}, \varepsilon), i) = (T, f, canonize\,(T, (\underline{b}, t_i))), \quad \textbf{if } \underline{b} \text{ has a } t_i\text{–edge} \quad (2.1.1)$$
$$= (T \amalg ((\underline{b}, \varepsilon), i), f, (\underline{b}, \varepsilon)), \quad \textbf{else if } \underline{b} = \text{root} \quad (2.1.2)$$
$$= update\,(T \amalg ((\underline{b}, \varepsilon), i), f', (f(\underline{b}), \varepsilon), i), \quad \textbf{otherwise} \quad (2.1.3)$$
$$f' = f \cup \{(\underline{b}, f(\underline{b}))\} \quad (2.1.3.2)$$

$$update\,(T, f, (\underline{b}, cw), i) = (T, f, canonize\,(T, (\underline{b}, cwt_i))), \quad \textbf{if } t_{l+|cw|} = t_i \quad (2.2.1)$$
$$= update\,(T \amalg ((\underline{b}, cw), i), f', (\underline{b}', u'), i), \quad \textbf{otherwise} \quad (2.2.2.1)$$
$$\textbf{where } \underline{b} \xrightarrow{(l,r)} \underline{v} \in T \text{ is a } c\text{-edge} \quad (2.2.2.2)$$
$$(\underline{b}', u') = canonize\,(T, (f(\underline{b}), cw)) \quad (2.2.2.3)$$
$$f' = f \cup \{(\underline{bcw}, \underline{b'u'})\} \quad (2.2.2.4)$$

Update() presents three direct cases: (2.1.2), (2.1.1) and (2.2.1). The last two cases would be equivalent to find the suffix to be inserted in T for the t_i-prefix, so it would also exist in T all the suffixes for this suffix (Lemma 1) and, in the first case, it would be the equivalent to have inserted all the suffixes for this t_i-prefix. The update of *f* in (2.2.2.4) links the new node created with the *split* operation with the equivalent point node. The update in (2.1.3.2) is a redundancy typical of the E. Ukkonen's algorithm, since *f(b)* already exists in *f.*

The case (2.1.3) of the *update()* function implies an update using *add-only.*
The case (2.2.2) of the *update()* function implies an update using *splt+add.*
The case (3.1) of the $\amalg$ stands for the constructing operation *add-only.*
The case (3.2) of the $\amalg$ stands for the constructing operation *split+add.*

$$T \amalg ((\underline{b}, \varepsilon), i) = T \cup \{\underline{b} \xrightarrow{(i,\infty)} \underline{bt_i}\} \quad (3.1)$$

$$T \amalg ((\underline{b}, cw), i) = (\{T \setminus \underline{b} \xrightarrow{(l,r)} \underline{v}\}) \cup \{\underline{b} \xrightarrow{(l,k)} \underline{bcw} \xrightarrow{(k+1,r)} \underline{v}, \underline{bcw} \xrightarrow{(i,\infty)} \underline{bt_i}\} \quad (3.2.1)$$
$$\textbf{where } \underline{b} \xrightarrow{(l,r)} \underline{v} \in T \text{ is a } c\text{-edge} \quad (3.2.2)$$
$$k = l + |w| \quad (3.2.3)$$

ukk() is the main loop, it will call *update()* for all the $|t|$ prefixes of *t*.

$$ukk(T, f, (\underline{b}, u), i) = T, \quad \textbf{if } i = n+1 \quad (4.1)$$
$$= ukk(T', f', (\underline{b}', u'), i+1), \quad \textbf{otherwise} \quad (4.2.1)$$
$$\textbf{where } (T', f', (\underline{b}', u')) = update(T, f, (\underline{b}, u), i) \quad (4.2.2)$$

Theorem 2: If $|t| = n$ then $ukk(T, f(\text{root}, \varepsilon), 1)$ returns $cst\,(t)$ in $O(n)$.
Proof : Proof in [Ukk95]. □

3.1.2 First optimization: placing the string label at the subsequent node in the transition

Placing the string label at the subsequent node of the transition makes impossible to create the $|\Sigma|$ transitions between the $\perp$ node and the root (*orig*(0,1)), since for all of them the root node is the subsequent node whereas it only contains one label. Therefore, we won't be able to replicate the use of the $\perp$ node performed by the original *ukk.*

The initialization of *f* (*orig*(0.2)) will be kept, but the initialization of T will turn into:

$T = \{\} \cup \{ \perp \rightarrow^{(0,1)} \text{root} \}$ *new* (0.1)

The *canonize()* function will be modified, and then the node $\underline{b}$ of the pair passed as parameter won't be any more the previous node of the transition and it'll become the subsequent one.

~~*canonize* (T, (*b*, ε))~~	~~= (*b*, ε)~~		*delete* (1.1)
canonize(T, ($\underline{b}$, *cw*))	= ($\underline{b}$, *cw*) ,	**if** $\|w\| < r - l$	*new*(1.2.1)
	= ($\underline{b}$, ε) ,	**else if** $\| w\| = r - l$	*new*(1.2.2)
	= *canonize*(T,($\underline{v}$,*c'w'*)) ,	**otherwise**	*new*(1.2.3.1)
	where $\underline{a} \rightarrow^{(l, r)} \underline{b} \in$ T		*new*(1.2.3.2)
	$c'w' = drop(r - l , w)$		*new*(1.2.3.3)
	$\underline{b} \rightarrow^{(l',r')} \underline{v} \in$ T is a *c'*-edge		*new*(1.2.3.4)

Canonize() needs to read the labels to know the transition string length (*orig*(1.2.2.2), n*ew*(1.2.3.2)). Since the labels are now placed in the subsequent node of the transition, *canonize()* will start to work with the subsequent node.

In the *update()* function, the operation *add-only* works with the subsequent node of the transition, like the *canonize()* function does, however, the operation *split+add* uses the previous node of the transition to update T and also to update *f* (*new*(2.2.21)). It will be necessary an interface treatment between both functions.

There'll be two interface treatments, one for passing from *update()* to *canonize()* after *split+add* and the other one for passing from *canonize* to *update* before *split+add*.

The first one will be performed by a new *link()* function, that will move from the previous node provided by *update()* to the subsequent one, that's required by *canonize()*. We'll also perform by means of the *link()* function the treatment handed to solve the problem of *dismissing the first character*, since it lacks the $\perp$ node (*new*(1'.1))

link (T, *f*, ($\underline{b}$, *chw*))	= *canonize* (T, ($\underline{v}$,*hw*)) ,	**if** $\underline{b}$ = root	*new*(1'.1)
	where *f* ($\underline{b}$)$\rightarrow^{(l,r)}\underline{v} \in$ T is a *h*-edge		*new*(1'.1.2)
	canonize (T, ($\underline{v}$,*chw*)) ,		*new*(1'.2)
	where *f* ($\underline{b}$)$\rightarrow^{(l,r)}\underline{v} \in$ T is a *c*-edge		*new*(1'.2.2)

There are two possible solutions that fit for the second interface, that involves moving from the subsequent node of the pair, provided by *canonize()*, to the previous one required by *update()*: the first one would entail keeping the parent of the last transition we had taken, as in *canonize()* (*new*(1.2.3)) as in *update()* (*new*(2.1.1)). This measure would involve an increase of the construction time for every transition we take during the construction process. The second solution would entail keeping a link with its parent node for every node in the tree. This measure would involve an increase, as of the construction time, in two updates when using a *split+add* and one when using an *add-only,* as of the space required, with an extra *word* for every node.

$$
\begin{aligned}
update\,(T, f, (\underline{b}, \varepsilon), i) &= (T, f, canonize\,(T, (\underline{v}, t_i))), && \textbf{if } \underline{b} \text{ has a } t_i\text{–edge} && new(2.1.1.1)\\
&\quad \textbf{where } \underline{b} \rightarrow^{(l,r)} \underline{v} \in T \text{ is a } t_i\text{-edge} && && new(2.1.1.2)\\
&= (T \coprod ((\underline{b}, \varepsilon), i), f, (\underline{b}, \varepsilon)), && \textbf{else if } \underline{b} = \text{root} && (2.1.2)\\
&= update\,(T \coprod ((\underline{b}, \varepsilon), i), f', (f(\underline{b}), \varepsilon), i), && \textbf{otherwise} && (2.1.3)\\
&\qquad f' = f \cup \{(\underline{b}, f(\underline{b}))\} && && (2.1.3.2)
\end{aligned}
$$

$$
\begin{aligned}
update\,(T, f, (\underline{b}, cw), i) &= (T, f, canonize\,(T, (\underline{b}, cwt_i))), && \textbf{if } t_{l+|cw|} = t_i && (2.2.1)\\
&= update\,(T \coprod ((\underline{a}, cw), i), f', (\underline{b}', u'), i), && \textbf{otherwise} && new(2.2.2.1)\\
&\quad \textbf{where } \underline{a} \rightarrow^{(l,r)} \underline{b} \in T && && new(2.2.2.2)\\
&\qquad (\underline{b}', u') = link\,(T, \underline{a}, cw)) && && new(2.2.2.3)\\
&\qquad f' = f \cup \{(\underline{bcw}, \underline{b'u'})\} && && (2.2.2.4)
\end{aligned}
$$

The constructing operations *add* and *split* would be kept, but the changes in the data structure will also affect the cost in time of the two operations[2].

orig.		*1st opt,*	
Placing label at the transition	+2	+2	Placing label at the leaf node
Linking transition with previous node.	+1	+1	Linking leaf node with previous node
Linking transition with leaf node.	+1	**0**	
TOTAL	+4	+3	

Table 1. cost in time for *add* according to the original *ukk* (left column) and according to the first optimization (right column).

[2] we won't take into account the cost involved in creating nodes.

orig.		*1st opt.*	
Placing label at the new transition node	+2	+2	Placing label at the previous node
Linking new node with old transition	+1	+1	Linking new node with subsequent node
Linking old previous node with new transition	+1	+1	Linking previous node with the new node.
Linking new transition with new subsequent node	+1	**0**	
Modifying old transition label	+1	+1	Modifying old subsequent node label
TOTAL	+6	+5	

Table 2. cost in time for *split* according to the original *ukk* (left column) and according to the first optimization (right column)

3.1.2.1 Efficiency

Definition: We will consider the same cost unit for all the assignation, comparison, crossing of a transition, and node linking operations.

We will achieve a time profit over the use of the original structure regarding to:
i) the transitions taken deep covering the tree
ii) the updates by means of *add-only*
iii) the updates by means of *split+add*

Lemma 3. We will achieve a +1 profit per every parent-child transition we take.
Proof: Not having transition node, when we take a parent-child transition instead of carrying out the node-transition-node step, we'll carry out a node-node step, and this way we'll avoid one jump per transition.

Lemma 4. We won't achieve neither additional profit nor loss regarding to the transitions we took in the *canonize()* function.
Proof: For every parent-child transition we take we must keep the parent, since as the *split* (*new*(2.2.2.1)) as the jump by *suffix-link* after this (*new*(2.2.2.3)) are carried out from the previous node (*new*(2.2.2.2)), whereas the label read by *canonize()* is placed at the subsequent node (*new*(1.2.3.2)). However, the original *ukk95* already suffers this cost, although is not reflected in the R. Giegerich and S. Kurtz [GiKu95] and [GiKu97], since *canonize()* works with the transition node and *update()* with the previous node.

Lemma 5. We will achieve a +1 profit per every *split.*
Proof. The way the *split* operation works, Table 2.

Lemma 6. We will achieve a +1 profit per every *add.*

Proof. The way the *add* operation works, Table 1.

Lemma 7. We will achieve a -1 loss per every *split.*
Proof. It's necessary to detect the root at *link()* to deal with the problem of *dismissing the first character* (*new*(1',1) y *new*(2.2.2.3))

Lemma 8. We will achieve a +1 profit per every *add-only* and *split+add.*
Proof. The fact of having to jump from $f(\underline{b})$ to the subsequent transition node to start covering the string of the equivalent path (*new*(1',1) and *new*(2.2.2.3)) is compensated for by the extra access to the transition node of the original *ukk95* at the last recursive call to *canonize()* to read the string label, if the construction op. is a *split+add*, or by the extra access to the subsequent node, if it's an *add-only* (*orig*(1.1) *vs new*(1.2.2) and *delete*(1.1)). Then we depend only on 5,6 and 7 lemmas.

Theorem 3. We will achieve a profit regarding to the construction time of the original algorithm in: number of *adds* + number of parent-child transitions we take.
Proof. Lemas 3,4 and 8.

Theorem 4. We will achieve a profit regarding to the query time of the original structure in: number of parent-child transitions we take.
Proof: Lemma 3.

Theorem 5. The changes suffered by the original cost in space, because of the action of this first optimization will be: – number of *words* nodes, and the number of nodes will be in the range [$|t|$, $2|t|$].
Proof: By placing the data at the subsequent node, instead of having node-transition-node type links, we'll have node-node type links. As a result, we'll keep the same information but using one less link. The number of explicit suffixes are $|t|$, so there will be a minimum of $|t|$ nodes. The cst(t) will have a maximum of $2|t|$ nodes. Proof in [McC75].

3.1.3 Second optimization: linking the subsequent nodes by means of *suffix-links*

We'll get back the method for *dismissing the first character* of the E. Ukkonen's algorithm as we introduced in the previous section.

$T = \{\} \cup \{ \perp \rightarrow^{(0,1)} \text{root} \}$	*1ra opt.* (0.1)
$f = \{\} \cup \{ (\text{root}, \text{root}) \}$	*new*(0.2)

Now we don't need the interfaces between *canonize()* and *update(),* since both functions work over the subsequent node (*1st opt*.(2.2.2) *vs orig*(2.2.2.1) *and new*(2.2.2.4)).

So we'll keep the *canonize()* function from the first optimization (*1st opt*(1)) and we'll part with the *link()* function. However, and due to the performance of the new constructing operation *split* (lemma 2), when we follows a *suffix-link* after an *add-only* op. we'll also have to cover the equivalent path (*new* (2.1.3.3)).

$$update\,(T, f, (\underline{b}, \varepsilon), i) = (T, f, canonize\,(T, (\underline{v}, t_i))), \quad \textbf{if } \underline{b} \text{ has a } t_i\text{–edge} \qquad (2.1.1.1)$$

$$\textbf{where } \underline{b} \rightarrow^{(l,r)} \underline{v} \in T \text{ is a } i\text{-edge} \qquad (2.1.1.2)$$

$$\sout{= (T \amalg ((\underline{b}, \varepsilon), i), f, (\underline{b}, \varepsilon)),} \quad \sout{\textbf{else if } \underline{b} = root} \quad delete(2.1.2)$$

$$= update\,(T \amalg ((\underline{b}, \varepsilon), i), f', (\underline{b}', \varepsilon), i), \quad \textbf{otherwise} \quad new\,(2.1.3.1)$$

$$\textbf{where } (\underline{b}', \varepsilon) = canonize\,(T, f(\underline{b}), w_{l-r})) \qquad new\,(2.1.3.2)$$

$$\underline{a} \rightarrow^{(l,r)} \underline{b} \in T \qquad new\,(2.1.3.3)$$

$$f' = f \cup \{(\underline{bt_i}, \text{root})\} \quad \textbf{if } \underline{bt_i} = \underline{b't_i} \qquad new\,(2.1.3.4.1)$$

$$= f \cup \{(\underline{bt_i}, \underline{b't_i})\} \quad \textbf{otherwise} \qquad new\,(2.1.3.4.2)$$

$$update\,(T, f, (\underline{b}, cw), i) = (T, f, canonize\,(T, (\underline{b}, cwt_i))), \quad \textbf{if } t_{l+|cw|} = t_i \qquad (2.2.1)$$

$$= update\,(T \amalg ((\underline{b}, cw), i), f', (\underline{b}', u'), i), \quad \textbf{otherwise} \quad orig(2.2.2.1)$$

$$\textbf{where } \underline{a} \rightarrow^{(l,r)} \underline{b} \in T \qquad (2.2.2.2)$$

$$(\underline{b}', u') = canonize(T, (f(\underline{b}), cw)) \qquad orig(2.2.2.3)$$

$$f' = f \cup \{(\underline{bt_i}, \underline{b't_i}), (\underline{bcw}, \underline{b'u'})\} \qquad new(2.2.2.4)$$

The explicit insertion of the last t_i-prefix suffix implies a decrease of the costs produced by the detection of the final case for the *i* step. While in the original *ukk95* we had to detect whether the node to be inserted was a root child (*delete*(2.1.2)), this query can be simplified by using the new *suffix-links* structure. We'll only have to detect whether the equivalent node is the one created in the previous update checking whether it has still unassigned its *suffix-link* (*new*(2.1.3.4.1)).

The cost can be even more reduced just by waiting to validate as true the condition ($\underline{bt_i} = \underline{b't_i}$) a maximum of $|\Sigma|$ times, once per every root child, and after skipping the query.

We have modified the way the *split* is performed in order to keep the strength of *f* (theorem 1).

$$T \amalg ((\underline{b}, cw), i) = (T \setminus \{\underline{a} \rightarrow^{(l,r)} \underline{b}, *(\underline{b} \rightarrow^{(l',r')} \underline{v})\}) \cup$$

$$\{\underline{a} \rightarrow^{(l,k)} \underline{b} \rightarrow^{(k+1,r)} \underline{bcw}\ *(\rightarrow^{(l',r')} \underline{v}), \underline{b} \rightarrow^{(i,\infty)} \underline{bt_i}\} \qquad new(3.2.1)$$

$$\textbf{where } \underline{a} \rightarrow^{(l,r)} \underline{b} \in T \qquad new(3.2.2)$$

$$*(\underline{b} \rightarrow^{(l',r')} \underline{v}) \in T \text{ are the } \Sigma\text{-edges} \qquad new(3.2.3)$$

$$k = l + |w| \qquad new(3.2.4)$$

The transfer of the $\underline{b}$ children to $\underline{bcw}$ (*new*(3.2.2) y *new*(3.2.1)) can be carried out in just one operation with a cost O(1) as if the children structure is a table as it's a list. It'll be only necessary to transform *r* from the old transition (*new*(3.2.2)). As can be seen in table 4, the new *split* operation doesn't involve any additional cost in time.

1st opt.			*2nd opt.*
Placing label at the leaf node	+2	+2	Placing label at the leaf node
Linking previous node with leaf node	+1	+1	Linking previous node with leaf node
	0	**0**	
TOTAL	+3	+3	

Table 3. cost in time for *add* according to the first optimization (left column) and according to the second optimization (right column).

1st opt.			*2nd opt.*
Placing label at the subsequent node.	+2	+2	Placing label at the new node
Linking subsequent node with new node	+1	+1	Linking old subsequent node with new node.
Linking new node with previous node	+1	+1	Copying childs
	0	**0**	
Modifying old subsequent node label	+1	+1	Modifying old subsequent node label
TOTAL	+5	+5	

Table 4. cost in time for *split* according to the first optimization (left column) and according to the second optimization (right column).

3.1.3.1 Efficiency

We will achieve a time profit over the first optimization regarding to:
i) the transitions taken when covering the equivalent path
ii) the updates by means of *split*+*add*
iii) the updates by means of *add-only*

Lemma 9. We will achieve a +1 profit per every parent-child transition we take when covering the equivalent path.
Proof: We don't need any more keeping the parent node, as in the first optimization was necessary (*1st opt*(2.2.2) *vs 2nd opt*(2.2.2)).

Lemma 10. We will achieve a –2 loss per every parent-child transition we take when covering the equivalent path after *add-only,* compensated by a +1 profit per *add-only.*

Proof. Unlike the first optimization, in this second one it's necessary to cover an equivalent path also after *add-only* (*new*(2.1.3.2) and *new*(2.1.3.3)). Since *ukk95* doesn't make any distinction between the *add-only* and the *split+add* operations, it also calls the *canonize()* function although after every *add-only* it doesn't cover any equivalent path.

Lemma 11. The execution of the constructing operations *split* and *add* doesn't involve any change in the cost.
Proof: Tables 3 and 4.

Lemma 12. We will achieve a +2 profit per every *split.*
Proof. We don't have to access to the child before covering the equivalent path after a *split+add* (*orig*(2.2.2.3) *vs 2nd opt.*(2.2.2.3) and *2nd opt*(1')). We also don't have to detect the root before executing *canonize()* to dismiss the first character (*orig*(2.2.2.3) *vs 2nd opt.*(2.2.2.3) y *2nd opt*(1'.1)).

Lemma 13. We will achieve a +1 profit per every *add-only.*
Proof. We don't have to detect the root before executing *update()* to check if the child to be inserted is a root child in order to determine whether we have updated in T all the suffixes for the t_i-prefix (*delete*(2.1.2), (2.1.1), *new* (2.1.3.4.1)).

Lemma 14. We will achieve a -1 loss per every *split+add.*
Proof: In this case it'll be performed an update of *f* for each new node created adding, two updates per each *split+add* operation (*new*(2.2.2.4)) and one per each *add-only* (*new*(2.1.3.4)), unlike the feature of the older *suffix-links* structure(just one update per each *split+add* and another one per each *add-only*).Pay special attention to the fact that the *ukk55*'s *update()* function reinserts after each *add-only* a link which had been previously inserted in *f,* since it doesn't distinguish the *add-only* from the *split+add* when it's time to update *f.* Therefore, we'll suffer a +1 extra cost per *split+add* and +0 per *add-only*.

Theorem 6. The profit in the construction time regarding to the first optimization is: +(number of transitions when covering the equivalent path after a *split+add* – number of transitions when covering the equivalent path after an *add-only*) + 2* number of *add-only* + number of *split+add*.
Proof : Lemmas 9,10,11,12,13 y14.

Theorem 7. By applying this second optimization we'll achieve a +|*t*| *words* loss in space.
Proof: With the new *suffix-links* structure every node will have its *suffix-link*, even the leaf nodes. So we'll have |*t*| extra pointers, one per each leaf node [Gus97].

Although with the new *split* operation a leaf node can become an internal node, new nodes inherit their child structure from the old subsequent node(Def. 7) so, if this node had been created as a leaf node without child structure, the new node will adopt this null child structure. As we have seen before, the new constructing operation *split* doesn't involve any extra cost neither in time nor in space.

Corollary 1. The profit in costs of this second optimization regarding to the original *ukk95* will be:

i) in construction time we'll achieve a profit of: + number of parent-son transitions we take + number of transitions when covering the equivalent path after a *split+add* – number of transitions when covering the equivalent path after an *add-only* + 3*number of *add-only* + 2* number of *split+add*.

ii) in query time we achieve a profit of: + number of parent-child transitions we take.

iii) in space we achieve a profit: - number of internal nodes *words*, where the number of internal nodes will be in the range [1 , $|t|$-1].

Proof . Theorem 3 , Theorem 4, Theorem 5, Theorem 6 y Theorem 7. □

3.2 Defining *mcc75*.

Instead of some minor differences, the profit regarding to the McCreight's algorithm is similar to the obtained using E. Ukonnen's algorithm. Although it can't be shown here because of some lack of space, these differences can be checked in [Hue01] together with the profit in costs and the transformation of the *mcc75* algorithm after applying both optimizations that were applied on *ukk95*.

4 The new algorithm

One of the greatest lacks of the algorithms introduced by M. McCreight and by E. Ukkonen lies in their lack of determinism when anticipate the different status of the algorithm. They don't make any distinction between the *add-only* and *split+add* operations and, of course, they don't try to anticipate the order of appearance of the two operations.

As we mentioned before, one of the most important suffix-tree features is its versatility to solve different problems. However, modifying and adding information to the one that is inherent to the tree, is necessary or at least advisable to be able to solve many of these problems. The possibilities of the suffix tree in this field are increased again by the greater independence obtained with the new *suffix-link*s. But to take advantage of these features is necessary to know the behaviour of the algorithm deeply enough to be able to carry out, the appropriate changes in a way that makes them efficient.

Like the E. Ukkonen's algorithm did, we'll cover the *t* prefixes from the shortest one to the longest one, so our algorithm will also be *on-line*. Though, the suffixes will be introduced in T starting from the lowest to greatest length [GiKu97], and form which is only necessary to know their common longest prefix (*lcp)* with the ones previously inserted in T.

We'll keep two indexes into the main loop of the algorithm. The first index: *i*, will cover *t* looking for the end of the longest common prefix for the next suffix to be inserted. The

second index: *j*, will lead us through the tree looking for this longest common prefix among the previously inserted suffixes.

```
for ( i = 1 to |t| ) do
    if (label of current node ended(j) ) do                    (C1)
        if ( can access to a following node(j) ) do            (C2)
            take parent-child transition ( j );                (A0)
        else do
            1 + n T updates by means of add-only;              (A1)
        endif
    else do
        if ( t_i = t_j ) do                                    (C3)
            increase the common prefix;
        else do
            1 + m T updates by means of split+add
              + n T updates by means of add-only;              (A2)
        endif
    end if
enddo
```

Alg. 1. First level algorithm

At each step of the algorithm we'll compare t_j and t_i. As soon as we detect a difference between the pointed characters we can assure that the common prefix just found is the longest one. We'll have to update then T and *f* in a transverse walk on the tree (note that T and *f* are only updated in the transverse walk and that *i* and *j* only alter its position regarding to *t* in the main loop).

The $t_{i-|lcp|}$-suffix = *lcp* | | t_i-suffix will be the next *t* suffix to be inserted. According to Def. 3, after each new *suffix-link* traversed j-1 is placed at the equivalent point and the *lcp* losses its initial character. Therefore we can prove by induction that in any update of the transverse walk for the step *i* we only explicitly insert the t_i-suffix. The transverse update finishes when the common prefix pointed by *i-1* and *j-1* is not the longest one.

Notice that *i* is showing us the sequence of the t_i-prefixes pointing the last character of each one.

We'll name *lcp** the longest common prefix at the time of starting the transverse walk for the *i* step. The walk will be by (A1) or (A2) of the algorithm 1, depending on whether *j* has placed itself at the middle or at the end of the transition string in the last transition taken in (A0) of the algorithm 1.

Proposition 1. Let be $0 \le n < |lcp^*|$, then 1+ *n* will be the number of suffixes of *t* inserted at the step *i* if C2 wasn't observed.
Proof: We'll carry out a first update on T and *f* to insert the new suffix, since it's not implicit in the tree because of the C1 condition failure. Then we'll have to transversely

update the subsequent suffixes until cover the *lcp* or until finding out that the common prefix pointed by *i*-1 and *j*-1 is not the longest one.

Proposition 2. The tree updates of the proposition 1 will be carried out by means of *add-only*.
Proof: According to the Lemma 1 any string accessible from a node will also be accessible from the equivalent node.

Proposition 3. Let be $0 \leq m+n < |lcp^*|$, then $1+ m+ n$ will be the number of suffixes of *t* inserted at the *i* step if C3 wasn't observed.
Proof: In this case C3 is the condition that fails because *lcp** finished in the middle of a transition string. The updates will take place as shows the proposition 1 proof.

Proposition 4. The inserts of the proposition 3 will be carried out by means of 1+*m* *split+add* followed by *n add- only*.
Proof : In the first update after C3 a *split+add* will be carried out, since if it wouldn't have been necessary a split we wouldn't have passed the condition C1. after it, we'll execute *m split+add* until finding out that the equivalent point is placed at the end of a transition string, so it won't be necessary to perform a *split.* According to lemma 1, once is carried out the first *add-only* we'll only execute updated by means of *add-only* for the remaining *n*-1 updates.

To perform the *canonize()* operation we'll utilize two functions:

i) *search equivalent node()* will provide us the equivalent node. We'll use it if we previously know that the next update of T will be executed by means of an *add-only.*

ii) *search equivalent node and position()* will return the canonical pair of the equivalent point. It's necessary to perform the *split+add.*

For each new update of T inside the same step, we'll link by *suffix-links* each new node with the node inserted at the previous transverse update.

Finally, the root node children need some special functionality. To find out whether we are inserting a child of root and therefore update its *suffix-link*, we'll check at the last of the *n add-only* if the parent node is linked to itself, since this fact represents an unequivocal evidence that we're dealing with the root node. Once are inserted the $|\Sigma|$ children of root we won't need this verification any more. If you wish to construct a *suffix-tree* with all the explicit suffixes it's enough concatenating to the text *t* a final character *'$'* where $t \in \Sigma^*$ and *'$'* $\notin \Sigma$ and jointly detecting the appearance of *'$'* and of the Σ-children of root

The second and third level of abstraction algorithms are widely explained in [Hue01].

Theorem 8: The newly presented algorithm involves a profit regarding to the E.Ukkonen and M. McCreight's ones in: +2* *add-only*.

Proof: Distinguishing the *n add-only* updates from the *m split+add* makes possible that we don't need to detect which update we're going to carry out next, since we know that after an *add-only* a *split+add* never will be necessary (Prop. 4).

Other effects that result of this lack of distinction between *split+add* and *add-only* in classical algorithms are, as we have seen in the previous section, the update of already existing in *f suffix-link*s and the unnecessary call to the function *canonize()* after each *add-only*. fig 4 shows the status diagram of the E.Ukkonen algorithm.

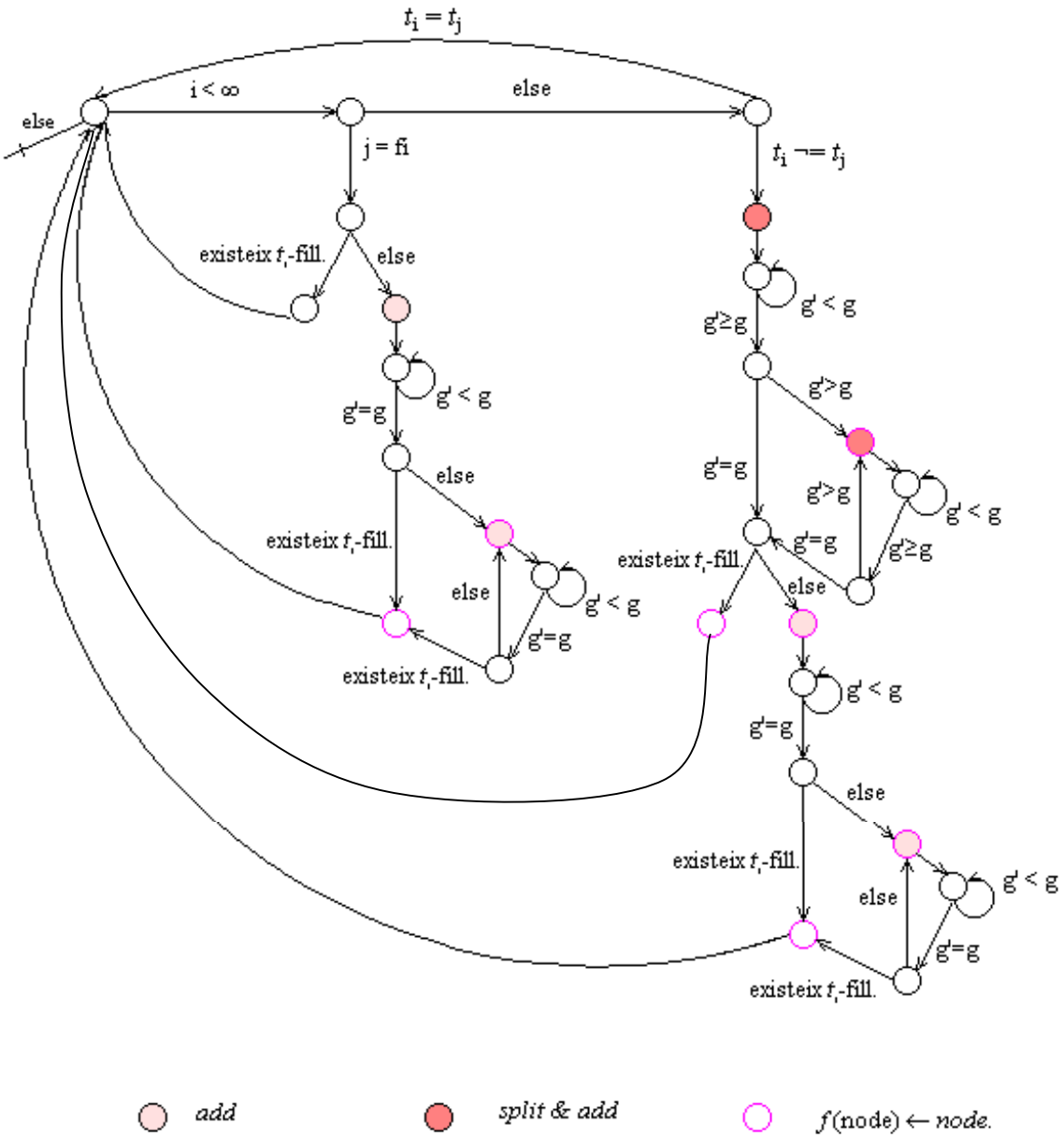

fig 3. Status diagram for the new algorithm. g represents the equivalent branch length and g' the length of the strings concatenation of the transitions taken after the jump by *suffix-link*. fi contains the end value of string label.

4.1 Understandability

Together with the problem of the memory space, one of the most quoted problems in the *suffix-tree* literature has been how little understandable it is [Apos85], [CroLec90], [GiKu95], [MNe96], [GiKu97], [Gus97], [Jlar98], [AOK02].

In fig. 3 diagram it can be seen the execution sequence for both transverse updates of our algorithm, (A1) and (A2) in alg. 1. For the 1+*n add-only* operations in the first case, there's an initial process, a recursive process and a final process previously to come back to the beginning of the algorithm. For the 1+*m split+add* + *n add-only* in the second case, we'll have an initial process, a recursive process and a final process similar to the first case processes, followed by the first case processes. Reaching this point we can affirm, that the almost mythical unintelligibility of the construction algorithms of the *suffix-tree*, gets reduced by this new algorithm, to a set of elementary iterations.

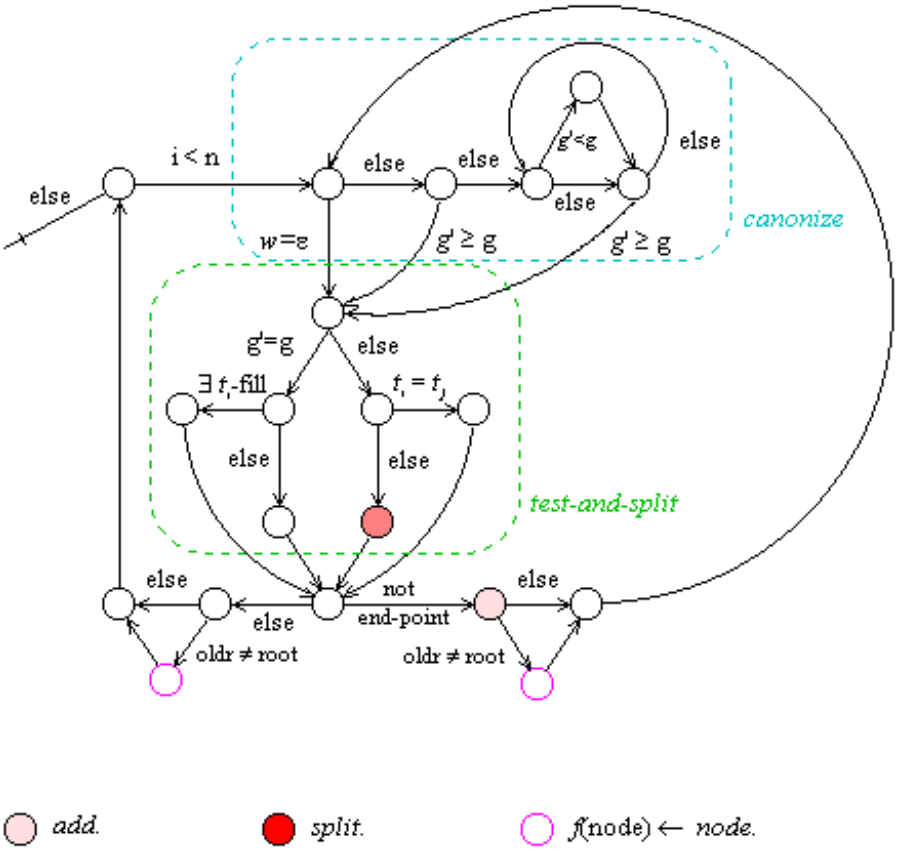

fig 4. Status diagram for *ukk*. The broken lines embrace the different status of the *canonize()* and *test-and-split()* functions. g represents the equivalent path length and g' the length of the strings concatenation of the transitions taken after the last jump by *suffix-link*.

In addition, the operations on *f* will also get simplified. The *suffix-link*s update won't depend on the kind of update on T and the transverse walks will be unified following a common behaviour regardless of the kind of operation (*split+add* or *add-only*).

One of the most outstanding suffix-tree optimizations is the space optimization carried out by S. Kurtz in [Kur99]. He shows the results obtained from applying to the *mcc75* a set of space optimizations that save around 40% of the space consumed by the classical *suffix-tree*. This algorithm has been used in such relevant computational biology applications as Mummer [DKF99],[DPC02] o Reputer [KCO01]. Let's see how useful can be the intelligibility of our algorithm with a real example. Let's check the effort necessary to apply the S. Kurtz optimizations on our algorithm.

The key point of the improvement that represents [Kur99] lies in his *chain* concept:

i) Set of nodes which are linked by a *suffix-link*

ii) All the internal nodes, except the root, belong to a chain and only to one chain.

iii) All the internal nodes that take part in a transverse path will be distributed in a series of consecutive chains.

The nodes of a chain will be placed consecutively in memory, so they won't need a link by *suffix-link*, and therefore the space will be highly reduced. This is possible due to:

i) The nodes are inserted in consecutive immediate updates.
ii) once a node is T inserted, its distance to root it's not altered.

Let's see how could we apply this optimization to the algorithm we're explaining here.

The chains are set up at the moment of creating the internal nodes, it means, in each *split* operation. As we can see in alg.1, the 1+*m* *split+add* operations for a concrete step are performed consecutively. Therefore, each chain will contain the internal nodes created in these 1+*m* updates.

Although in the new algorithm we're talking about the nodes vary its distance to root(def. 7), the *suffix-links* remain unaltered (theorem 1), and so we can apply the optimization described by [Kur99].

In [Kur99], S.Kurtz describe the way to apply the optimization to the McCreight's algorithm. It's easy to observe how the process that's expose there involves a greatest complexity, due to the utilization of the McCreight's algorithm. Another interesting exercise would be trying to apply the optimization on the E. Ukkonen's algorithm, leading us this time by the diagrams in fig. 3 and 4.

5 Experimental results

Once we've formally proved that our algorithm is the most efficient of the classical construction algorithms, let's see its behaviour in practice.

The alphabet size won't be taken into consideration. The alphabet size affects the cost of the algorithms, since if we utilize a bigger alphabet the tree loses height, so the number of parent-child transitions we take is reduced and together with them the advantage of our algorithm (corollary 1). We'll utilize DNA to tests, which is a reduced alphabet.

We'll use a table structure to access the children nodes. This fact will involve a direct access to them and a required |∑|*Words space for the three algorithms.

number of bases	E. Ukkonen'95			McCreight'75		Huerta'02
		Mc	Hu		Hu	
Micoplasma pulmoniae 2000	6.29	106%	165%	5.92	155%	3.8
Micoplasma pulmonis 7000	8.46	108%	187%	7.80	173%	4.51
Helicobacter j99 20000	14.99	106%	173%	14.11	163%	8.63
Helicobacter 26695 50000	15.13	104%	174%	14.45	163%	8.68
Ecoli 1100000	63.84	103%	178%	61.82	173%	35.70
Ecoli 2200000	86.75	101%	197%	85.54	194%	44.04
Pseudomonas aunuginosa 3000000	151.81	108%	277%	139.70	255%	54.70
Streptomyces coelicolor A3 8600000	417.03	102%	298%	407.26	291%	139.62

Table 5. *Suffix-tree* construction time and speed up for the three algorithms and some complete genomes from GenBank.

The comented source code of the new algorithm, the source code of the classical ones used for the tests, and more statistical tests are provided by www.revolutionresearch.org and http://www.lsi.upc.es/~alggen/

6 Conclusions and future work

In this paper we've offered a different design for the suffix-tree data structure, a new constructing operation and a high-level simplified algorithm that uses them. As a result we've achieved a greater time and space efficiency, the ability to work with longer sequences, a more understandable construction algorithm and the independence amongst the *suffix-links* of the same node children.

The greater efficiency of the algorithm we've presented has been formally proved and experimentally measured. We've observed how easily can be adapted the algorithm with a practice example and, as a prove of the power offered by the independence between the *suffix-links* there is [HuMe02], the on-line algorithm to search for MUMs[DKF99]. As is showed in [HuMe02], means of the use of additional information which is accessible from the *suffix-links* and using an algorithm similar to the matching statistics[ChLa94], we get the unique-maximal-matchings between two sequences: S1,S2, with a space cost of O(|S1|), instead of the original O(|S1|+|S2|). If we compare a set of sequences: ST, with

a pattern sequence *p*, then the cost time is $O(|p|+\sum_{i=0}^{|ST|}|STi|)$, instead of the original $O(\sum_{i=0}^{|ST|}(|p| + |ST_i|))$. [HuMe02] proposes some really complex structures, showing several lists and a multiple linked in both directions suffix-tree. The simplicity of the algorithm we're introducing is the key point that has made possible to adapt the algorithm to this purpose.

This *suffix-link* feature will represent an advantage in new problems which are to be found, with on-line query algorithms like the one we've seen or with other kind of algorithms (like the supermaximal repeats).

Aknowledgements

Thanks to Rafael Jimenez, Marta Rojas and Maria Jose Serna for their help. All work have been supported by the *Revolution Research Institute*.

References

[AhCo75]. A. V. Aho and M. Corasick. Efficient String Matching: An Aid To Bibliographic search. *Comm. ACM*, 18(6):333-340,1975.

[AILSV88]. A. Apostolico, C. Iliopolous, G. M. Landau, B. Schieber, and U. Vishkin. Parallel Construction of a Suffix Tree with Aplications. *Algorithmica*, 3:347-367, 1988.

[Apos85]. A. Apostólico. The myrad virtues of suffix trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 85-96, Springer-Verlag, Berlin, 1985.

[ALS99]. A Andersson, N. J. Larson, and K. Swanson. Suffix trees on Words. *Algorithmica*. 23: 246-260, 1999.

[AKO02]. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The Enhanced Suffix Array and its Applications to Genome Analysis. *In Proceedings of the Second Workshop on Algorithms in Bioinformatics*. Springer Verlag, Lecture Notesin Computer Science,accepted for publication, 2002.

[AOK02]. M.I. Abouelhoda, E. Ohlebusch and S. Kurtz, Optimal Exact String Matching Based on Suffix Arrays. *In Proceedings of the Ninth International Symposium on String Processing and Information Retrieval.* Springer-Verlag, Lecture Notes in Computer Science, 2002.

[Bak93]. B.Baker. A theory of parametrized pattern matching:Algorithms and applications. *Proceedings of the 25th ACM Symposium on Theory of Computing*, 71-80, 1993.

[BoMo77]. R. S. Boyer and Moore. A Fast String Searching Algorithm. *Comm. ACM*, 20(10):62-72,1977.

[ClMu96]. D. R. Clark and J. I. Munro, Efficient suffix trees on secondary storage, *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms* 383-391 1996.

[CrFe99]. Andreas Crauser and Paolo Ferragina. A Theorical and Experimenal on the Construction of Suffix Arrays in External Memory. *MPI* 1999.

[ChLa94]. W.I. Chang and E.L. Lawler. Sublinear Approximate String Matching and Biological Applications. *Algorithmica* 19:331-353 1994.

[CroLec90]. Maxime Crochemore and Thierry Lecroq. *Hand book of Computer Science and Engineering*. Chapter 6.

[CreVer97]. M. Crochemore, R. Vérin. Direct Construction of Compact Direct Acyclic Word Graphs. In A. Apostolico and J. Hein, editors, *Proc th Annual simposium on combinatorial Pattern Matching*, volume 1264 of Lecture Notes in Computer Science, pages 116-129. Springer-Verlag, 1997.

[DKF99]. A.L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White and S. L. Salzberg, Alignment of whole genomes, *Nucleic Acids Res.,* 27:2369-2376, 2002.

[DPC02]. A.L. Delcher, A. Phillippy, J. Carlton and S. L. Salzberg, Fast Algorithms for large-scale genome alignment and comparison, *Nucleic Acids Res.,* 30(11):2478-2483, 2002.

[Far97]. M.Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proc. of the 23th Annual Symposium on the Foundations of Computer Science (FOCS)*, 1997,137-143.

[Gian95]. R. Giancarlo. A generalization of suffix-trees to square matrices with applications*, SIAM Journal of Computing*, 1995, 520-562.

[GiKu95]. R. Giegerich and S. Kurtz. A Comparison of Imperative and Purely Functional Suffix Tree Constructions. *Science of Computer Programming*, 25(2-3):187-218,1995.

[GiKu97]. R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Constructions. *Algorithmica*, 19:331-353, 1997.

[GKS99]. Robert Giegerich, Stefan Kurtz, Jens Stoye. Efficient Implementation of Lazzy Suffix Trees. 3*rd Workshop on Algorithm Engineering,* 1999.

[GrVi00]. Roberto Grossi, Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *Comm. ACM*, 2000.

[Gus97]. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997. http://www.cs.ucdavis/~gusfield/strmat.html

[Har94] R. Hariharan, Optimal parallel suffix tree construction, *IEEE Symp. Found. Computer Science* 290-299, 1994.

[HuMe02] M. Huerta and X. Messeguer. *On-Line search of Maximal Unique Matchings Using Suffix-links*, Departament LSI, Universitat politécnica de Catalunya, technical report.2002.

[Hue01] M. Huerta. *Estructuras de datos para la biología computacional*, Universitat politécnica de Catalunya, FIB, 2001. http://www.revolutionresearch.org/

[Jlar98]. N. Jesper Larsson. *Attack Of Mutant Suffix Trees*. Phd, 1998.

[JlarSad99]. *Faster Suffix Sorting*. Technical Report LU-CS-TR:99-214, Dept. of Computer Science, Lund University, 1999.

[Kär95]. J. Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Combinatorial Pattern Matching*, volume 937 of *Lecture Notes in Computer Science,* pages 191-204. Springer, 1995.

[KMP77]. D. E. Knuth, J. H. Morris Jr, V. R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6(1):323-350, 1977. http://sunburn.informatik.uni-tuebingen.de/~buehler/AC/AC.html

[KU96]. J. Kärkkäinen., E. Ukkonen. Sparse suffix trees. Lecture Notes in Computer Science, 1090:219-230, 1996.

[Kur99]. S. Kurtz. Reducing the Space Requeriments of Suffix Trees. *Software- Practice and Experience,* 29(13):1149-1171, 1999.

[KCO01]. S. Kurtz, J.V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Griegerich. REPuter: The Manifold Applications of Repeat Analysis on a Genomic Scale. *Nucleic Acids Res.,* 29(22):4633-4642, 2001.

[MNe96]. Mark Nelson. Fast String Searching With Suffix Trees. *Dr. Dobb's Journal*. August 1996

[McC75]. E. M. McCreight. A Space–Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262-272, 1976.

[MM93]. U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935-948, 1993.

[Sad97]. Kunihiko Sadakane. Comparison Among Suffix Array Construction Algorithms. *IPSJ S16 Notes 97-AL-59*.

[SaJe99] N. Jasper Larsson and K. Sadakane. Faster Suffix Sorting. *Lund University, Sweden, pp. 129-138*. 1999.

[Ukk95] E. Ukkonen. On-Line Construction of Suffix-Trees. *Algorithmica*,14(3), 1995.

[Wei73] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory,* pages 1-11, The University of Iowa, 1973.

[ZMR98] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for test indexing. *ACM transactions on Database Systems*, 23(4):453-490, Dec. 1998.