The at most k-deep factor tree

Julien Allali* Marie-France Sagot†

Résumé

Cet article présente un nouvelle structure d'indexation proche de l'arbre des suffixes. Cette structure indexe tous les facteurs de longueur au plus k d'une chaîne. La construction et la place mémoire sont linéaires en la longueur de la chaîne (comme l'arbre des suffixes). Cependant, pour des valeurs de k petites, l'arbre des facteurs présente un fort gain mémoire vis-à-vis de l'arbre des suffixes.
Mots Clefs: arbre des suffixes, arbre des facteurs, structure d'indexation.

Abstract

We present a new data structure to index strings that is very similar to a suffix tree. The new structure indexes the factors of a string whose length does not exceed k, and only those. We call such structure the *at most k-deep factor tree*, or *k-factor tree* for short. Time and space complexities for constructing the tree are linear in the length of the string. The construction is on-line. Compared to a suffix tree, the k-factor tree offers a substantial gain in terms of space complexity for small values of k, as well as a gain in time when used for enumerating all occurrences of a pattern in a text indexed by such a k-factor tree.

keywords: suffix tree, at most k-deep factor tree, string index, pattern matching

1 Introduction

The suffix tree is one of the best known structures in text algorithms. It is used in many fields such as string matching [5] [6], data compression [15][16], computational biology [3] [7] [17] [22] etc. A suffix tree indexes all suffixes of a string and can be constructed in time linear in the length of the string. Furthermore, the size and depth of the tree are also linear in the length of the string. The constant however may be big. One of the original implementations of a suffix tree [18] thus requires 28 bits per input char. Numerous efforts have been made over the years to reduce this space. In particular, S. Kurtz proposed in [14] various implementations which require 20 bits per input char in the worst case and 10 on average. The same author with co-workers suggested in another paper [9] a variant of the suffix tree, called the *lazy suffix tree*, which can be built on the fly as the need arises.

In many applications, the length of the patterns to be searched in the suffix tree can be limited. Typically, this concerns applications to motif extraction in biological sequences where

*Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France, allali@univ-mlv.fr

†Inria Rhône-Alpes, Université Claude Bernard, Lyon I, 43 Bd du 11 Novembre 1918, 69622 Villeurbanne cedex, France, Marie-France.Sagot@inria.fr

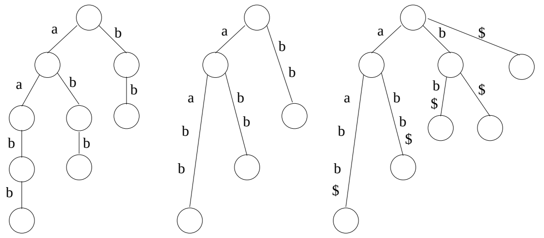


Figure 1: Suffix trie of $s = aabb$ (left), implicit suffix tree of s (center) and suffix tree of $aabb\$$ (right).

the motifs of interest are often short or are composed of short parts at constrained distances from one another [25], applications to data compression using a sliding window [11], [20], etc.

There is therefore often no need to have the whole suffix tree. This is our motivation for proposing a new structure, called the *at most k-deep factor tree*, or, for short, the *k-factor tree*. The k-factor tree retains the linear properties of the suffix tree. Indeed, its construction follows closely that of the suffix tree as proposed by Ukkonen [24]. For small values of k, there is furthermore a gain in time and space that may be important. The gain in time concerns both the construction itself and the use of the tree, for instance, to do pattern matching.

Since our construction of the k-factor tree is strongly based on Ukkonen's suffix tree construction, we start by recalling Ukkonen's algorithm in Section 2. We then introduce our own algorithm for the k-factor tree construction. The k-factor tree has been implemented using the coding proposed by Kurtz in [14] and the results of the experiments are discussed in Section 5.

2 Suffix trees

Before recalling Ukkonen's suffix tree construction, we start by introducing some notations.

2.1 Notations

In what follows, s will denote a string over a finite alphabet Σ whose cardinality is $|\Sigma|$. The length of s is $|s|$. A string s is therefore an element of the free monoid Σ^*. The i^{th} letter of s is denoted by s_i and $s_{i \dots j}$ represents the factor $s_i s_{i+1} \dots s_j$ (also named substring or subword). The suffix of s starting at position i is the factor $s_{i \dots |s|}$. If s ends with a letter which is not in Σ, we say that s has an *ending symbol*, denoted by \$. Given two strings u and v, we note $u.v$ the concatenation of u with v.

Given a tree T with root R whose edges are labelled by elements of Σ^* and a node N of T, $path(N)$ is the string corresponding to the concatenation of the labels encountered along the path from R to N. The depth of N, denoted by $depth(N)$, is the number of nodes between R and N, N included ($depth(R) = 0$).

2.2 Definition of a suffix tree

The suffix trie of s is a tree with edges labelled by elements of Σ. For each factor of s, there exists a node N such that $path(N)$ is equal to that factor. If the factor is a suffix of s, some

```
SuffixTree(s)
Tree T
For i from 1 to |s|
    For j from 1 to i
        Add(T,s_{j...i})
```

Figure 2: Naive algorithm

of these nodes may be leaves. If s has an ending symbol, all nodes N whose path from R spells a suffix of s are leaves.

The *implicit* suffix tree of s is a tree with edges labelled by non-empty elements of Σ^*. The tree is a compressed version of the suffix trie. Each internal node N of the suffix trie that has only one son is deleted and its two adjacent edges are replaced by an edge that goes from N's father to N's son. The label of the new edge is equal to the concatenation of the label of the edge going from N's father to N and of the label of the edge from N to its son. This tree is called implicit because not all suffixes of s lead to a leaf. The true suffix tree is obtained when an ending symbol \$ is added at the end of s.

2.3 Suffix tree construction

A first suffix tree construction algorithm of complexity linear in the length of the string was presented by Weiner in 1973 [26]. McCreight introduced a similar but more performing algorithm in 1976 [18]. In 1995, Ukkonen published an on-line construction of the suffix tree using a very different approach [24]. Later, Giegerich and Kurtz [8] showed how all three constructions are in fact similar.

We now briefly present Ukkonen's algorithm which is the basis for the k-factor tree construction. For more details, we refer the reader to the original paper [24]. We follow here the description of the algorithm given in [13].

3 Ukkonen's algorithm

The algorithm is divided into $|s|$ phases. The i^{th} (for $1 \leq i \leq |s|$) phase consists in the insertion of all suffixes of $s_{1...i}$ into the tree. Each phase i is then divided into j steps, one step for each insertion of a suffix of $s_{1...i}$ (see Figure 2). This naive algorithm is clearly in $O(|s|^3)$. We show now how Ukkonen obtains a linear complexity.

Cases met when inserting a word w in the tree

Let us call v the longest prefix of w which is already in the tree. We have $w = vu$. During the insertion of w, three cases may occur:

1. u is the empty word so that w is already in the tree: there is nothing to do.

2. spelling v from the root leads to a leaf: all we need to do is append u to the label of the edge leading to this leaf.

3. the last case may be divided into two subcases:

   (a) spelling v leads to an internal node N: we attach a new leaf to this node; the edge from N to the leaf is labelled u.

   (b) spelling v leads to the middle of an edge: we have to cut this edge at the position reached and insert a new node N; to the node N thus created, we then attach a new leaf; the edge from N to the leaf is labelled u.

Suffix links

Suffix links are fundamental elements in all linear time suffix tree construction algorithms. We shall see later that they enable to reduce the cost of string insertion during the algorithm.

A suffix link is an oriented link between two internal nodes in the suffix tree. Given a node N, its suffix link points to the node, denoted by $S_l(N)$, such that $path(S_l(N))$ is equal to $path(N)_{2\ldots|path(N)|}$ (that is, to $path(N)$ without the first letter). In other words, $path(S_l(N))$ is the longest proper suffix of $path(N)$. All internal nodes have a suffix link.

Suffix links are added to the tree during construction as follows: let us suppose that, during the insertion of string $s_{j\ldots i}$, a node N is created to which a new leaf is attached (case 3b). We know that $s_{j\ldots i-1}$ is in the tree because we inserted it during the previous phase. We therefore have that $path(N)$ is equal to $s_{j\ldots i-1}$. By construction, when a string is in the tree at the end of a phase, all suffixes of this string are also in the tree. The factor $s_{j+1\ldots i-1}$ is thus already in the tree. Two cases may then happen: the path labelled $s_{j+1\ldots i-1}$ ends in a node, let us denote it by N', or it ends inside an edge. In the second case, the insertion of factor $s_{j+1\ldots i}$ leads to the creation of a node, that we also denote by N'. In both cases, the suffix link of N points to N'.

Fast insertion

Let N be the last node reached during the insertion of $s_{j\ldots i}$ that has a suffix link. We have that $s_{j\ldots i}$ is equal to $path(N).w.\sigma$, where w can be empty and $\sigma \in \Sigma$. If $S_l(N)$ is the root, then the insertion of $s_{j+1\ldots i}$ is done naively. If $S_l(N)$ is not the root, in order to insert $s_{j+1\ldots i}$, we just have to follow w from $S_l(N)$ and add σ if necessary.

During the insertion described above, we found w in the tree by successively comparing each letter of w. In fact, we can avoid such comparisons in the following way. At each node met during the insertion, we just need to know which edge to take. Once this edge is identified, we can go directly to the node it points to and, as a consequence, move in w by possibly more than one letter.

The time spent during an insertion is essentially composed of the time needed to calculate the length of an edge plus the number of nodes traversed during the insertion. The next section shows how the length of an edge can be computed in constant time and how, simultaneously, the space required by the tree may be reduced. In the proof of the time linearity of the algorithm, we show that the overall number of nodes traversed is then $O(|s|)$.

Edge coding

The labels of the edges can be coded by a pair of integers which denote, respectively, the start and end positions in s of the string labelling the edge. This allows to obtain a linear space occupancy for the tree as is detailed below. It also leads to an improvement in the time complexity of the suffix tree construction as leaves can then be automatically extended.

Automatic leaf extension and condition for ending a phase

Let us call *end position of an edge* the second value in the label of an edge, which corresponds to the end position of the string labelling that edge. Suppose a leaf is created during the insertion of $s_{j\ldots i}$. The end position of the edge pointing to this leaf is i. In the next phase, the insertion of $s_{j\ldots i+1}$ will lead to case 2, and the end position of the edge will now become $i+1$, and so on for all the following phases. When a leaf L with edge E leading to it is created, we can then set the end position of E to a global variable whose value is the number of the phase. This will exempt us from having to make the insertions corresponding to the extension of all edges leading to a leaf. Furthermore, if the insertion of $s_{j\ldots i}$ requires the creation of a leaf, then all strings $s_{\ell\ldots i}$ for ℓ from 1 to j lead to a leaf in the tree. During the next phase, if the insertion of $s_{j+1\ldots i}$ has not created a leaf, we can therefore start the insertion of the suffixes of $s_{1\ldots i+1}$ from the insertion of $s_{j+1\ldots i+1}$ (that is, from step $j+1$) because all the longer suffixes are implicitly inserted. Each phase then starts from the last leaf created.

In the same way, if, during the insertion of $s_{j\ldots i}$, we end up in case 1, we do not have to insert $s_{\ell\ldots i}$ for ℓ from $j+1$ to i because if $s_{j\ldots i}$ is already in the tree, then all suffixes of $s_{j\ldots i}$ are also in the tree.

In what follows, all non implicit insertions such as those described above are called *explicit*.

3.1 Algorithm

Figure 3 gives the complete algorithm for Ukkonen's construction. We see that index j does not appear anymore. As just showed, the beginning of each phase is deduced from the end of the previous phase, and the end of a phase is deduced from the result of a current insertion.

Moreover, we may observe that the algorithm of Figure 3 constructs the implicit suffix tree of s. To obtain the suffix tree of s, an ending symbol \$ must be appended to s.

3.2 Complexity

We now analyse the time and space complexities of the suffix tree construction. These will be the same for the k-factor tree.

Time complexity

Ukkonen's assertion [24] that his suffix tree construction algorithm is linear in the length of the string rests upon the following lemma [13]:

Lemma 1 *Let N be a node. We have that $depth(N) \leq depth(S_l(N)) + 1$.*

The implicit leaf extensions inside a phase take constant time, so the implicit insertions made over the complete algorithm take $O(|s|)$ time. The time taken by the explicit insertion of a string in the tree is directly related to the number of nodes traversed after the jump along the suffix link since the ascent from a node to its father and the jump along the suffix link can be done in constant time. Instead of counting the number of nodes traversed after the jump along the suffix link for each separate insertion, we can upper bound the number of nodes traversed during the whole construction. Let j be the index of the extension currently considered.

```
AddString(node,s,start,end)
endJump ← false
while(not endJump) and ((end − start) not equal to 0) do
    set child to the child of node that start with the letter s_start
    if (end − start) is greater or equal to length(node, child)
        start ← start + length(node, child)
        node ← child
    else
        endJump ← true
done
if (end − start) is equal to 0 and node has not a child for letter s_end
    add a child to node with edge label start equal to end
e ←the label of the edge between node and child
if e_{end−start+1} not equal to s_end
    split e at position end − start
    add a leaf with start position equal to end to the new node
done.

Suffix_Tree(s)
Add to R a leaf L with edge label s_1
lastLeaf ← L
For i from 2 to |s|
    endPhase ← false
    do
        forward ← length(Father(lastLeaf), lastLeaf) − 1
        if S_l(Father(lastLeaf)) is undefined and Father(lastLeaf)! = R
            forward ← forward + length(Father(Father(lastLeaf)), Father(lastLeaf))
            if Father(Father(lastLeaf)) is R
                AddString(R,s,i − forward + 1,i)
            else
                AddString(S_l(Father(Father(lastLeaf))),s,i − forward,i)
        else
            if Father(lastLeaf) is R
                AddString(R,s,i − forward + 1,i)
            else
                AddString(S_l(Father(lastLeaf)),s,i − forward,i)
        if a node was created during the previous step
            set suffix link of this node to the last node reached during the insertion
        if a leaf was created in the call to AddString
            set lastLeaf to this leaf
        if a node was not created in the call to AddString
            endPhase ← true
    while(not endPhase)
end for
return R
```

Figure 3: Ukkonen's algorithm: construction of the suffix tree for s; R is the root of the tree. The function *length* returns the size of the edge label between two nodes. The function *AddString* achieves the fast insertion.

Index j remains unchanged between two successives phases, and it never decreases. We may observe that we do at most $2|s|$ explicit insertions because we have $|s|$ phases and j is at most $|s|$. During an explicit insertion, the depth of the node currently considered is first decreased by at most two, one to reach the father of a leaf and one when jumping along the suffix link, and then increased at each skip down the tree. Since the maximal depth of the tree is $|s|$ and we do at most $2|s|$ insertions, the total number of nodes considered is $O(|s|)$. If we suppose that the access to a child of a node is done in $O(1)$ time, then the total time complexity of the suffix tree construction is in $O(|s|)$.

Space complexity

Let T be the suffix tree of s. The root of T has exactly $|\Sigma| + 1$ children. One of them corresponds to a leaf (the string label of the edge leading to it is \$). The worst case for the space complexity will be obtained if each internal node has exactly two children and each edge is labelled with a single letter (*i.e.* its length is one).

In that case, the total number of nodes whose depth is k in the tree is:

$$2^{k-1}(|\Sigma| + 1) - 2^k + 2 \texttt{ that is } 2^{k-1}(|\Sigma| - 1) + 2.$$

The number of leaves in T is $|s| + 1$ and we can then deduce the depth d of T in the worst case which is:

$$2^{d-1}(|\Sigma| - 1) + 2 + d - 1 = |s| + 1 \texttt{ that is } 2^{d-1}(|\Sigma| - 1) + d = |s|$$

An upper bound for d is:

$$d_{max} = \log_2 \frac{2|s|}{|\Sigma| - 1}.$$

We can now compute the total number of nodes in T which is:

$$1 + \sum_{k=1}^{k=d_{max}} [2^{k-1}(|\Sigma| - 1) + 2]$$

that is:

$$2|s| + 2d_{max} - |\Sigma| + 2.$$

4 Factor trees

We now present the k-factor tree construction. A naive approach consists in building the suffix tree and then pruning the tree in such a way that for each node N of the remaining tree, $|path(N)| \leq k$. Clearly, this approach does not improve the space complexity because it requires first constructing the suffix tree.

Recall that one of the main ideas behind Ukkonen's suffix tree construction is the automatic leaf extension allowed by the use of a global variable. We must be able to preserve this idea in the k-factor tree construction if we wish to keep the linear time complexity. Let us call *length of a leaf* L, the length of the string labelling the path from R to L. Preserving Ukkonen's idea requires finding a way of stopping the extensions when a leaf reaches the length k.

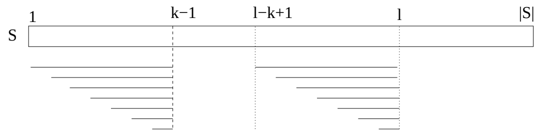


Figure 4: k-Factor tree construction algorithm. The horizontal lines indicate the substrings of s which are inserted into the tree during phase i. On the left, the substrings inserted for $i = k - 1$ are shown and, on the right, the substrings of length at most k inserted for $i = \ell$ (for $\ell \geq k$).

4.1 Construction

The k-factor tree contruction is based on Ukkonen's algorithm. It is divided into two parts:

- building the implicit suffix tree for $s_{1...k-1}$;
- adding to the tree all the suffixes of $s_{i-k+1...i}$ for i from k to $|s|$.

We now detail these two parts. Henceforward, we call *end position of a leaf* what corresponds to the end position of the edge leading to that leaf.

First part: construction of the implicit tree of the suffixes of $s_{1...k-1}$

The first part is done following Ukkonen's algorithm in a straightforward way. The only difference is that, when a leaf is created, it is now added to a queue called $queue_{leaf}$.

Second part: construction of the implicit tree of the suffixes of $s_{i-k+1...i}$ for i from k to $|s|$

For each string corresponding to a suffix of $s_{i-k+1...i}$ for i from k to $|s|$ that needs to be inserted in the tree, we proceed essentially as in Ukkonen, that is, we go up to the last node which has a suffix link, jump along the suffix link and then go down in the tree. As for the first part, we add each newly created leaf to $queue_{leaf}$. We now also have to remove an element from the head of the queue at the end of each phase. Furthermore, there may be another subtle but important difference in relation to Ukkonen's algorithm depending on the state of $queue_{leaf}$ at the beginning of a phase.

Indeed, in Ukkonen's suffix tree construction, a phase starts from the last leaf created. At the start of a phase i in our algorithm, two cases may happen:

- there is a leaf L at the end of $queue_{leaf}$;
- $queue_{leaf}$ is empty.

In the first case, we proceed with phase i straightforwardly from leaf L, which corresponds to the last leaf created.

In the case where $queue_{leaf}$ is empty, we do not proceed as in Ukkonen from the last leaf that was created in the algorithm, but from the last position reached in the tree during the previous phase. This position may be in the middle of an edge. The reason for this is the following. Let i be the phase for which $queue_{leaf}$ is empty when the phase begins and let

L be the last leaf created before phase i. The leaf L corresponds therefore to $s_{i-k\ldots i-1}$. At phase i, we would start from leaf L, jump up to L's father, along L's father suffix link and then try to insert $s_{i-k+1\ldots i}$. If $s_{i-k+1\ldots i}$ were already in the tree, phase i would be terminated and phase $i+1$ would start again from the leaf L. The first string needing to be inserted would be $s_{i-k+2\ldots i+1}$ and we would have to follow two suffix links from L's father in order to reach $s_{i-k+2\ldots i-1}$ and check for the existence of $s_i s_{i+1}$ at the end of $s_{i-k+2\ldots i-1}$. If $s_{i-k+2\ldots i+1}$ were already in the tree, phase $i+1$ would also be stopped, and phase $i+2$ would again start from the leaf L and require now three suffix link jumps before trying for the insertion of the first suffix of the phase. If we proceeded in this way, we could no longer guarantee a linear complexity for the algorithm. This will not be the case if, instead, we start each phase for which $queue_{leaf}$ is empty at the last position reached in the tree during the previous phase. Inserting $s_{i-k+1\ldots i}$ implies then just checking, from such a position, for the existence of s_i, which can be done in constant time. If a leaf needs to be created, we create it, add it to $queue_{leaf}$ (it will become the head of the queue) and go to the next step of the phase. If a leaf L is reached ($path(L)$ is thus already equal to $s_{i-k+1\ldots i}$), we proceed with the next step of the phase. Any leaf that needs to be created during the phase is added to the queue.

At the end of phase i, the leaf L at the head of $queue_{leaf}$ is removed. The end position of L is set to i.

Lemma 2 *Stopping the automatic extension:*
For each phase i, $k \leq i \leq |s|$, if $queue_{leaf}$ is not empty let us call L the leaf at the head of $queue_{leaf}$. The length of $path(L)$ is equal to k at the end of phase i.

Proof. When a leaf is created in the suffix tree, its length is always equal to the length of the last created leaf minus one. This can be observed in Ukkonen's algorithm. Indeed, if in the phase i, we start with step j and we create a leaf L, it will be because during a previous phase ℓ, the insertion of $s_{j-1\ldots\ell}$ ended in the creation of a leaf L', while during phases $\ell+1\ldots i-1$ there was no leaf created. At the end of phase $i-1$, leaf L' will have a length of $i-(j-1)+1$ because of the automatic extensions. We then insert $s_{j\ldots i}$ which creates leaf L whose length is equal to $i-j+1$.

During the first phase, which is phase $i=k$, of the second part of the algorithm (when the suffixes of $s_{i-k+1\ldots i}$ for i from k to $|s|$ are inserted), the leaf L at the head of $queue_{leaf}$ is the one that was created during the insertion of s_1. It was extended to $k-1$ during the first part of the algorithm. At the end of phase $i=k$, L is removed from the queue, its end position is set to i and thus its length is clearly k. If, at this point, any leaf remains in the queue, the length of the one at the head is $k-1$, and will become k at the end of the next phase $(i+1)$ when it is removed from the queue, and so on.

Suppose now that at the beginning of a phase i, $queue_{leaf}$ is empty. We start the phase by trying to insert $s_{i-k+1\ldots i}$. If this creates a leaf, it will be put in the queue and will be at the head of it. When the phase ends, the leaf will be removed and have length k. If no leaf is created in phase i, we proceed with phase $i+1$ and apply the same reasoning. □

4.2 Algorithm

The pseudo code for the first part of the k-factor tree construction algorithm is the same as for Ukkonen (Figure 3), we just have to add to $queue_{leaf}$ the leaf created in line 21. The pseudo code for the second part of the k-factor tree construction is given in Figure 5.

```
Factor_Tree(R, s , k , queue_leaf)
For i from k to |s|
    endPhase = false
    if queue_leaf is not empty
        set lastLeaf to the leaf at the end of queue_leaf
    else
        add s_i from last position reached during the last insertion
        if a leaf is created
            add this leaf at the end of queue_leaf
            set lastLeaf to this leaf
        else
            set lastLeaf to the leaf reached
    do
        forward ← length(Father(lastLeaf), lastLeaf) − 1
        if S_l(Father(lastLeaf)) is undefined and Father(lastLeaf)! = R
            forward ← forward + length(Father(Father(lastLeaf)), Father(lastLeaf))
            if Father(Father(lastLeaf)) is R
                AddString(R,s,i − forward + 1,i)
            else
                AddString(S_l(Father(Father(lastLeaf))),s,i − forward,i)
        else
            if Father(lastLeaf) is R
                AddString(R,s,i − forward + 1,i)
            else
                AddString(S_l(Father(lastLeaf)),s,i − forward,i)
        if a node was created during the previous step
            set the suffix link of this node to the last node reached during the insertion
        if a leaf was created in the call to AddString
            set lastLeaf to this leaf
            add this leaf at the end of queue_leaf
        if a node was not created in the call to AddString
            endPhase ← true
    while(not endPhase)
    remove the leaf at the head of queue_leaf and set its end value to i
end for
return R
```

Figure 5: Second part of the algorithm for the construction of the at most k-deep factor tree of a string s.

4.3 Complexity

We now analyse the time and space complexities of the k-factor tree.

Time complexity

The time taken by the construction of the k-factor tree is linear in the length of s. The proof is very similar to the one for Ukkonen's suffix tree construction presented in section 3.2. The first part of the algorithm requires $O(k)$ time. Lemma 1 remains true for the k-factor tree. During the second part of the algorithm, we have $|s|-k+1$ phases. We can start by observing that all operations related to $queue_{leaf}$ are done in constant time and therefore take $O(|s|-k)$ time for the whole algorithm. As in Ukkonen, between two phases, the start index j in s of the first substring of s that must be inserted can never decrease. The complexity thus still depends only on the total number of nodes encountered during all insertions. This number is at most $2|s|$ (we have $|s| - k + 1$ phases and j is at most $|s|$). As the depth currently reached in the tree remains unchanged between two phases and we can go up by at most two nodes at each insertion, we can go up in the tree by at most $4|s|$ nodes only during the whole algorithm. Since the depth of the tree is at most k, the total number of nodes encountered is $O(|s|)$.

If we make the assumption that the cost to reach the child of a node is constant ($|\Sigma|$ is fixed), the whole algorithm therefore runs in linear time.

Space complexity

We now compute the number of nodes in the k-factor tree in the worst case. Clearly, the worst scenario corresponds to the case where each node has $|\Sigma|$ children. The number of nodes is then $\sum_{\ell=0}^{\ell=k} |\Sigma|^{\ell}$, that is:

$$\frac{|\Sigma|^{k+1} - 1}{|\Sigma| - 1}.$$

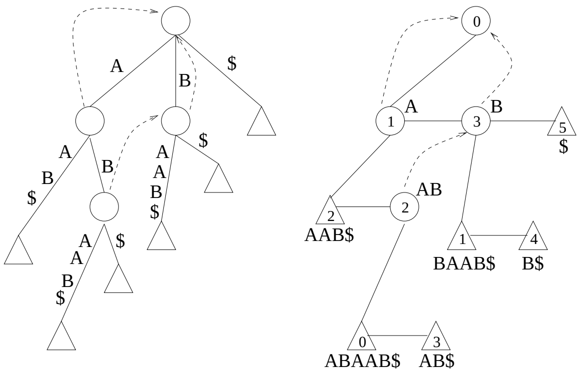
The total number of nodes in the worst case is therefore:

$$min(\frac{|\Sigma|^{k+1} - 1}{|\Sigma| - 1}, 2|s| + 2\log_2(\frac{2(|s| - 1)}{|\Sigma| - 1}) - |\Sigma| + 2).$$

We can thus guarantee a gain in memory when k is less than $\log_{|\Sigma|}(2(|\Sigma| - 1)|s|) - 1$. This result has been confirmed in practice. Experiments are presented in the next section.

5 Coding and experiments

The construction algorithm we have just presented may be used with any currently existing coding of the suffix tree. We chose to use it with the coding adopted by S. Kurtz in [14] for building suffix trees. This employs an "Improved Linked List Implementation" and is called the *illi* coding. The choice is motivated by the fact that, to the best of our knowledge, this is the best implementation in practice which is currently available. We start by explaining the coding techniques adopted by S. Kurtz then describe the modifications or extensions we had to do to it in order to adapt the coding to the construction of k-deep factor tree. Basically, the coding must be changed so that it can efficiently handle the fact that a leaf in the factor tree may now store more than one position. We end by presenting some experimental results.



Node	head(N)	$\lvert path(N)\rvert$	first child	right sibling	suffix link
0	0	0	1	--	--
1	2	1	△2	3	0
2	3	2	△0	--	3
3	4	1	△1	△5	0

Leaf	heaf(F)	$\lvert path(F)\rvert$	right sibling
△0	0	6	△3
△1	1	5	△4
△2	2	4	2
△3	3	3	--
△4	4	2	--
△5	5	1	--

Figure 6: The suffix tree of $s = ABAAB\$$, the right tree shows the suffix tree as it is coded by *illi*.

5.1 Kurtz's coding

We briefly explain the *illi* coding. For further details, the reader is referred to [14].

The internal nodes and the leaves of the suffix tree are coded using two different tables. In both tables, the index of a node corresponds to the order in which the nodes are created during construction and thus put in the right table. In particular, the index of a leaf corresponds to the position in s of the suffix that is spelled by the path from the root to the leaf. In all cases, the children of a node are stored in a linked list (first-son, right-sibling). The coding for a node contains the following information:

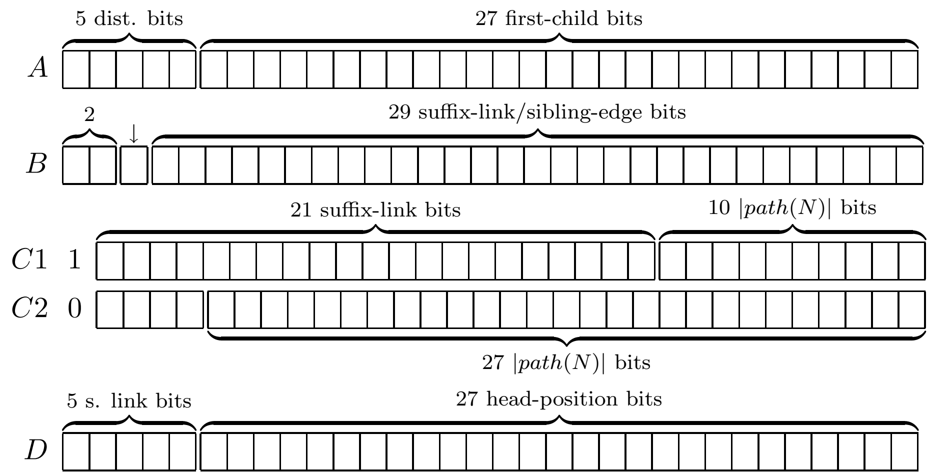
- The node is an internal node N:
  - start position of the first occurrence of $path(N)$ in s that required the creation of a node (this corresponds to the first occurrence of $path(N)$ in s that is followed by a letter different from the one following all previous occurrences; possibly there is only one). Such position will be denoted by $head(N)$;
  - $|path(N)|$;
  - the first child of N;
  - the right sibling of N;
  - the suffix link of N.
- The node is a leaf F:
  - the right sibling of F.

  Observe that this information is enough since the values of $head(F)$ and $|path(F)|$ for a leaf F may be deduced directly from the index of the leaf in the table. Indeed, $head(F)$ is that index (of creation of the leaf F) and $|path(F)| = |s| - head(F)$.

Figure 6 presents a suffix tree coded according to *illi*. One of the main characteristics of the coding comes from the distinction between two types of nodes: the so-called *Small* and *Large* nodes. The idea is that nodes are created in batches, one batch (of possibly more than one node) inside each phase of the construction algorithm. During phase i (lines 12-32 in Figure 5), if z nodes are created, say $N_1, \ldots, N_z$, then for each node N_i, $1 \leq i < z$, we have that N_{i+1} is the node to which points the suffix link of N_i (line 26), $|path(N_i)| = |path(N_z)| + (z-i)$ and $head(N_i) = head(N_z) - (z-i)$. The nodes N_i for $1 \leq i < z$ are the so-called *Small Nodes* and the node N_z is the so-called *Large Nodes*. The coding of a *Small Node* will also include a field indicating the distance (*i.e.*, the number of nodes) until the *Large Node* with which it is associated. This distance is equal to $z - i$.

The coding is detailed now. As indicated in [14], the index of a leaf may be stored on 27 bits, that of an internal node on 28 bits. To code the index of any node, we thus need 29 bits, the first one indicating if the next 28 correspond to the index of a leaf or of an internal node.

A leaf F occupies 32 bits coded according to the model B shown below. The third bit ($\downarrow$) indicates whether the sibling of F is *nil*. If this is the case, it is flagged and the next 29 bits code for the suffix link of the father (as explained below), otherwise the next 29 bits code for the sibling of F. We shall come back to the use of the first 2 bits later.



The coding of a *Small Node* is done using $2 * 32$ bits (models A and B above) and the coding of a *Large Node* on 4*32 bits (models A, B, C and D above). We denote by N the coded node, and by i its index in the table of nodes. The first 5 bits of A are zero if N is a *Large Node* and, if N is a *Small Node*, are equal to the *distance* until the *Large Node* with which N is associated. In the case where $distance \geq 32$, a "dummy" *Large Node* is inserted at the index $i + 32$. The first child of N is stored on the last 27 bits of A and the first 2 bits of B. The remaining 30 bits code for the right sibling. If N is a *Small Node*, its coding is finished. Indeed, the suffix link, $head(N)$ and $|path(N)|$ may all be calculated from the *Large Node* whose value is $I_N + distance(N)$ as described above. If N is a *Large Node*, it has 2*32 more bits. If $|path(N)|$ can be coded on 10 bits (*i.e.* its value is strictly less than 1024), then the next 4 bytes are coded according to scheme $C1$: the first bit is flagged, the next 21, the first 5 bits of D and the 2 free bits of the leaf whose value is $head(N)$ are used to code the suffix link of N. If $|path(N)| \geq 1024$, then the next 4 bytes are coded according to scheme $C2$. The 27 last bits code for $|path(N)|$. The suffix link of N is then coded in the rightmost child (coding of B) whose field "right sibling" is zero. Finally, the last 27 bits of D code for $head(N)$.

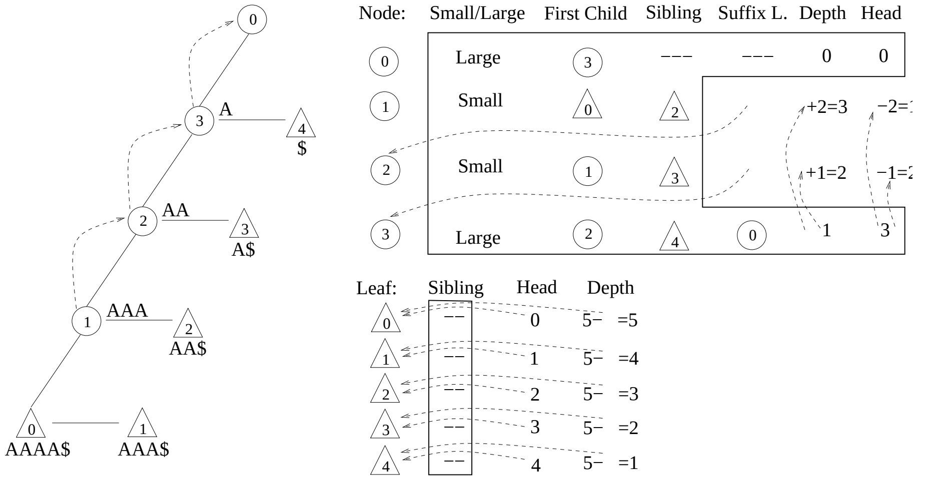


Figure 7: An example of a best case for the *illi* coding, $s = AAAA\$$. In the boxes, what the data that are really coded.

5.2 Coding for the factor tree construction algorithm

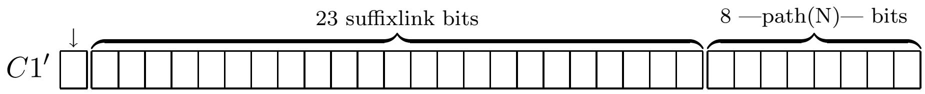
Let us examine now how we must change the *illi* coding in order to use it in the implementation of our factor tree construction algorithm. As observed previously, the values $head(F)$ and $|path(F)|$ of a leaf F can be deduced directly from the index of the leaf: the leaf F whose index is i is such that $head(F) = i$ and $|path(F)| = |s| - i$. This is not true anymore in the case of a k-deep factor tree. Indeed, since some leaves are cut, a shift is introduced in the order in which a leaf is created and then inserted in the table for the leaves. We can clearly identify the moment when a leaf is cut and a shift is thus introduced. This happens at line 11 of the algorithm given in Figure 5. An easy way out would be at each such point to create and insert in the table a dummy leaf as we could then again deduce the values of $head(F)$ and $|path(F)|$ from the index of F. This solution is however not satisfying because memory space is unecessarily lost and, furthermore, we cannot produce the list of all the occurrences associated with a leaf.

Another way of solving the problem consists in introducing a coding of the occurrences of a leaf in the form of a linked list. At line 11 in the algorithm, if a leaf F is reached at step p, a new cell is created in the table for all the leaves whose index in the table is $p - k$. We then store in the cell the index of F. In the tree, the link to F (field first child of the father or sibling of the left brother) is replaced by a link to the leaf whose index is $p - k$.

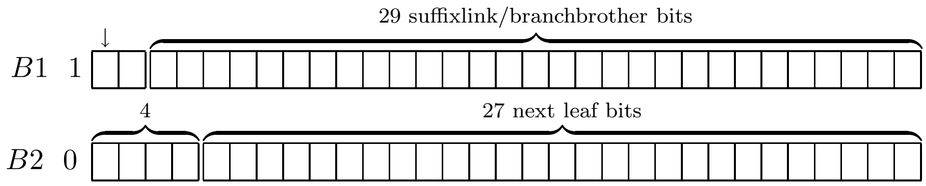
This new procedure presents two problems:

- when one traverses a list of occurrences, an additional bit is required to indicate that we have reached the end of the list, but, as we have seen above, all the bits of the field have already been used;
- to obtain the information "right sibling/suffix link of the father", we need to examine all the occurrences of the leaf, which implies that the complexity of the construction algorithm is no longer in $O(|s|)$.

In order to address these problems, the coding of a node has to be modified. In particular, we need to "free" bits in the coding scheme of a leaf. We thus modify the coding of the nodes in the following way. The suffix link of a node is still either to a node which exists already, or to the next node which will be created just after. In the case where the suffix link is to the next node, we have a *Small Node* and the index of the suffix link is not coded but is deduced from the node. In the other case, we have a *Large Node*. Let us denote by i the index of N in the table of the nodes. If $i \leq 2^{26}$, we know that the suffix link can be coded using 26 bits and therefore, we employ the same coding as above without using the 2 bits of the leaf whose index is $head(N)$. If $i > 2^{26}$, we use the coding scheme $C1'$ below. If $|path(N)|$ can be coded on 8 bits (*i.e.*, its index is less than 256), then the 23 bits of $C1'$ plus the 5 bits of D are used to code the suffix link. If $|path(N)| \geq 256$, we follow the coding scheme $C2$ and the suffix link is found in the rightmost child.



We thus have 32 bits per leaf and not 30 anymore. We recall that the index of a leaf requires 27 bits. We also need 1 bit to indicate whether we have reached the end of a linked list. In the case where we are at the end of a list, we remain with 31 bits, 29 to code the value *right sibling/suffix link* and 1 to indicate whether the index of the right sibling is *nil*. If we have not reached the end of a list, then of the 31 remaining bits, 27 are used to code the position of the next occurrence. To guarantee that we can obtain the value *right sibling/suffix link* in constant time, we use the last 4 bits to code part of the index of the last cell. In this way, in the worst case, we need to traverse the first 7 cells of the list before being able to jump directly to the last cell. To improve access time to the last cell, we optimise the code in the following way: if we are in cell c of the list, then we have already read $c * 4$ bits of the index of the last cell. If i is smaller than 2^{c*4}, then the value read is the index of the last cell (because this index is less than c). Another possible optimisation that has not yet been implemented consists in using in cell c, with c coded on j bits, $32 - 1 - j$ bits to code part of the index of the last cell. Indeed, the cell after i has a smaller index and thus can be coded on j bits. In what is shown below, $B1$ represents the model used for the last cell, otherwise it is the model $B2$ that is employed.



The algorithm used to insert a cell in the linked list is the following. Denoting by j the position of the next occurrence (last considered cell in the table for the leaves), by i the index of the leaf to which an occurrence is added and by e the index of the leaf at the end of the list whose first cell is i, we do the following:

- while looking for e, we:
  - check whether the value e has already been coded in the first cells of the list;
  - determine the index l of the last cell reached before e.

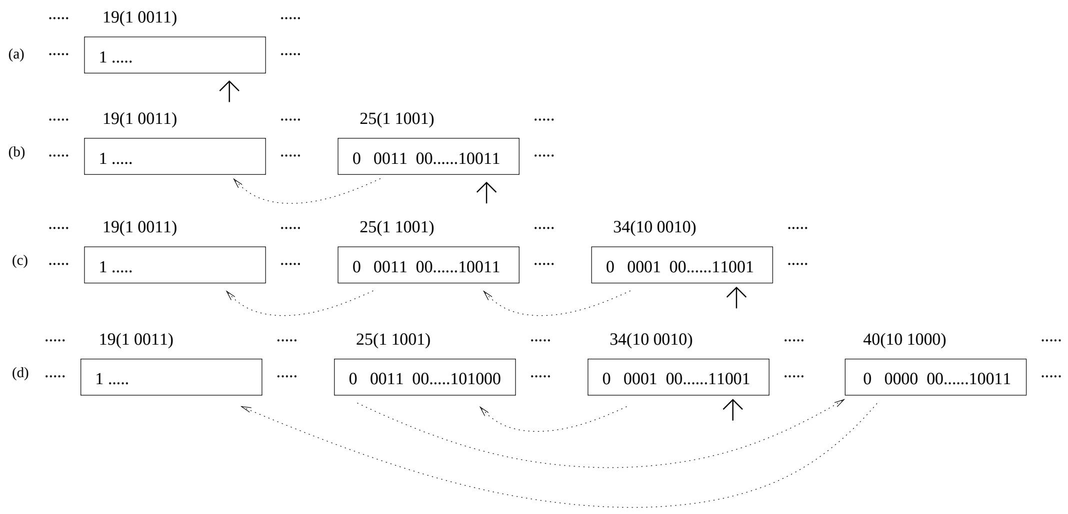


Figure 8: Insertion in a list of occurrences. (a) The list contains only one cell, 19. (b) Insertion of occurrence 25, $i = e = 19$. (c) Insertion of occurrence 34 $i = 25$, $e = 19$ and $l = 25$.(d) Insertion of occurrence 40 $i = 34$, $e = 19$, $l = 25$, 19 is already coded in the cells 34 and 25, 40 is added after 25.

- if e has already been coded, we insert cell j after cell l and i stays at the head of the list;
- otherwise, we place j at the head of the list. Its coding follows model $B2$. We then need to update the pointers to i in the tree so that they point to j.

An example of the procedure for inserting a cell is given in Figure 8.

5.3 Experimental results

5.3.1 Suffix and factor tree construction time and space occupancy

Our first test uses 43 files from the Canterbury Corpus [1] and the Calgary Corpus [4]. All tests have been done on a PC with a single $1GHz$ processor with $4Go$ of RAM memory under linux Debian/Gnu. Whenever possible, we added a termination symbol at the end of the file. The table presented below contains the following information (in order):

- the name of the file with, in parenthesis, its type: t for a text file, b for a binary file, d for DNA and f for a file in formal language;
- the length of the text;
- the size of the alphabet used in the file;
- the results obtained with a suffix tree using the $illi$ coding:
  - execution time in seconds;
  - space occupied in bytes;

- the results obtained with a k-deep factor tree using our coding and for $k = 20$ then $k = 10$:
  - execution time in seconds;
  - the memory gain (in %) in relation to a suffix tree with the *illi* coding;
- the results obtained with the k-deep factor tree with our coding for a memory gain of at least 15%:
  - execution time in seconds;
  - value of k.

The last line indicates the total size of the data, followed by the average value (time, space or gain) by column. In the case where there are 2 values in a cell, the second value corresponds to the average of the values in the column except for the files *aaa* and *alphabet*.

Let us start by commenting the space memory used by both data structures. We may observe first that the type of results obtained depends on the size of the file. Indeed, cutting the tree at a depth of 20 when the text is short (for instance, *paper5* which makes 11Kb) produces a weaker space gain than cutting it also at a depth of 20 when the text is long (for instance, *world192* which makes 2,5Mb). However, independently from the type and size of the file, the factor tree offers an often significant gain for $k = 10$ as well as for $k = 20$, except for the files *random* and *kennedy*. The text in these two files present the same characteristic: they are not structured and do not contain long repeats.

The files containing a text written in a formal language are the most structured ones and are among the ones yielding the best results. For example, a file such as *cp* which is in html format contains long repeats that are specific of the language (*e.g.* the sentence `<a href="http://www.` with 20 characters). The structure itself of formal texts favours the existence of deep nodes in the tree and therefore a reasonable cut ($10 \leq k \leq 20$) leads to very good memory gains such as 25.98% for $k = 10$. The files *aaa* and *alphabet* are equally favourable to both a suffix tree with the *illi* coding and a k-factor tree with our coding in the sense that they too contain long repeats. In fact, files such as these two represent the best cases for both approaches. They lead to the best gains for the factor tree relatively to a suffix tree.

Text files produce also good memory gains. However, natural languages do not in general contain very long repeats and the gain is thus in general less strong, although there may be exceptions to this. For instance, the *bible* file where sentences like " `And God ...`" favour the presence of nodes at a depth of more than 10. The average gain of 18.28% for a height $k = 10$ indicates that the factor tree is well adapted to this type of file.

The results on binary files are less easy to predict. Indeed, the file *pic*, which codes for an image, contains long repeats due to the repetitition of identical lines and the gain obtained on it is good.

For the other files, the results are disappointing. There are two reasons for this: the alphabet of these files is often big and since the text in the files are not structured, long repeats are rare. These files present therefore some of the less good results with, nevertheless, an average gain of 14.25% for $k = 10$.

file	length	$\|\Sigma\|$	suffix tree		k=20		k=10		15%	
			time	space	time	gain	time	gain	time	k
alice29 (t)	152090	75	0.50	9.84	0.51	1.95	0.43	16.12	0.47	10
asyoulik (t)	125180	69	0.38	9.77	0.48	1.30	0.38	9.37	0.38	8
bib (t)	111262	82	0.31	9.46	0.31	9.55	0.28	25.65	0.32	15
bible (t)	4047393	64	13.56	9.64	14.17	9.94	12.50	40.60	14.29	17
book1 (t)	768772	83	3.05	9.83	3.33	0.24	2.97	14.36	3.03	9
book2 (t)	610857	97	2.05	9.67	2.09	3.81	2.04	23.36	2.15	12
lcet10 (t)	426755	85	1.35	9.66	1.60	4.33	1.35	22.39	1.54	11
paper1 (t)	53162	96	0.14	9.82	0.17	4.96	0.14	16.12	0.14	10
paper2 (t)	82200	92	0.22	9.82	0.26	1.72	0.25	13.54	0.23	9
paper3 (t)	46527	85	0.15	9.80	0.14	0.79	0.12	9.21	0.15	8
paper4 (t)	13287	81	0.04	9.91	0.04	1.11	0.04	8.12	0.03	7
paper5 (t)	11955	92	0.03	9.80	0.04	1.24	0.03	8.44	0.04	7
paper6 (t)	38106	94	0.11	9.89	0.10	5.11	0.10	16.17	0.11	10
plrabn12 (t)	481862	82	1.75	9.74	2.00	0.34	1.96	11.59	1.88	9
world192 (t)	2473401	95	7.70	9.22	7.96	20.85	7.93	37.03	8.14	27
SUBTOTAL:	9290719		2.20	9.72	2.33	4.66	2.15	18.28	2.31	11.36
aaa (f)	100001	2	0.14	12.26	0.06	67.35	0.04	67.36	0.16	77734
alphabet (f)	100001	27	0.27	12.26	0.08	67.35	0.08	67.35	0.27	77713
cp (f)	24604	87	0.07	9.34	0.07	13.81	0.07	24.43	0.07	18
fields (f)	11151	91	0.01	9.79	0.04	10.69	0.01	25.97	0.03	15
grammar (f)	3722	77	0.02	10.14	0.02	7.83	0.01	21.93	0.01	13
news (f)	377110	99	1.49	9.54	1.43	10.00	1.44	20.16	1.56	12
progc (f)	39612	93	0.11	9.59	0.14	5.44	0.10	17.37	0.09	10
progl (f)	71647	88	0.17	10.23	0.18	19.87	0.16	34.11	0.16	27
progp (f)	49380	90	0.12	10.31	0.12	22.85	0.12	36.13	0.11	41
trans (f)	93696	100	0.23	10.50	0.20	30.73	0.19	42.96	0.25	59
xargs (f)	4228	75	0.01	9.63	0.01	1.68	0.01	10.79	0.01	8
SUBTOTAL:	875152		0.24	10.32	0.21	23.41	0.20	33.50	0.25	14150
				9.90		13.65		25.98		22.55
geo (b)	102401	256	0.85	7.49	0.87	0.17	0.89	0.40	0.60	4
kennedy (b)	1029745	256	10.06	4.64	9.21	0.00	5.99	8.20	1.57	2
obj1 (b)	21505	256	0.10	7.69	0.07	6.65	0.11	10.86	0.10	6
obj2 (b)	246815	256	1.01	9.30	1.08	14.03	1.00	26.03	1.08	18
pi (b)	1000001	11	4.13	10.13	4.43	0.00	4.39	0.00	3.20	6
pic (b)	513217	160	1.20	8.95	0.85	44.01	0.74	47.30	1.16	204
random (b)	100001	65	0.50	7.05	0.49	0.00	0.49	0.00	0.32	3
sum (b)	38241	255	0.14	8.92	0.15	14.55	0.12	21.20	0.13	18
SUBTOTAL:	3051926		2.25	8.02	2.14	9.93	1.72	14.25	1.02	32.62
TOTAL:	13369887		1.53	9.52	1.55	11.89	1.36	22.19	1.29	4490.58
						8.42		19.37		19.78

file	length	$\|\Sigma\|$	suffix tree		k=20		k=10		15%	
			time	space	time	gain	time	gain	time	k
C14	87164908	5	319.98	12.43	342.30	7.35	243.73	67.34	328.53	15
C22	34553832	5	116.28	12.29	124.82	9.83	92.30	66.20	120.99	15
J03071	66495	4	0.14	12.36	0.13	21.15	0.11	33.05	0.14	30
K02402	38059	4	0.10	12.59	0.09	0.14	0.11	3.69	0.07	8
M64239	94647	4	0.23	12.62	0.25	0.51	0.26	7.44	0.23	9
AE000111	4639221	4	14.84	12.56	16.33	1.04	9.80	59.44	14.11	12
V00636	48502	4	0.11	12.57	0.13	0.00	0.14	3.02	0.10	8
X14112	152261	4	0.44	12.55	0.48	0.88	0.38	18.79	0.39	10
TOTAL:	126757925		56.51	12.49	60.56	5.11	43.35	32.37	58.07	13.37

Let us now examine the results obtained on biological data, in this case, DNA sequences. The sequences considered were chromosomes 14 and 22 of *Homo sapiens* (retrieved from the database of the NCBI [21] and denoted by C14 and C22), and seven sequences retrieved from the EMBL database (the first column indicates their primary access number). The results on chromosomes 14 and 22 of man are encouraging given the size of the tree in memory (more than 1Go), the 15% gain of the factor tree over the suffix tree represents a real gain. Otherwise, although the gain remains important for $k = 10$, results become disappointing for $k = 20$.

Figure 9 shows the results for different values of k on the sequence with EMBL primary access number *AE000111* which corresponds to the sequence covering the first 400 entries of the complete genome of *Escherichia coli* in the EMBL database. We see that, as expected [2] [23], $\log_4 |s|$ is a good indicator of the value of k starting from which the gain diminishes as this value is increased. The curve, that presents the same behaviour with random sequences (results not shown), thus confirms that the tree is almost complete up to a depth of $\log_{|\Sigma|} |s|$. More details may be found in [2] and [23], where the authors study the internal repeats of a text and average height of a suffix tree. Observe that for DNA, the value of $\log_4 |s|$ goes from 10 for a sequence of 1Mo to 13 for a sequence of 100Mo. However, one must remember that the nature of these sequences may change these results.

Finally, let us consider the construction time required by both structures. In the case of the factor tree, we can see that the tests added by our algorithm during construction lead to a slight increase in the execution time. However, as soon as values of k between 10 and 20 are reached, the factor tree is at least as performing as the suffix tree.

5.3.2 Execution time for pattern matching using a suffix or factor tree

We now present the results concerning the time required to do pattern matching with a suffix tree or a factor tree. More precisely, since the time for simply searching for a pattern (that is, just telling whether a motif is in a text) is the same for both trees, we rather examine the time taken in each case to enumerate all the positions of a pattern in the text. This is expected to be different because factor trees are cut at a depth k and a leaf points now to a list of occurrences instead of a single one as in the case of a suffix tree. We implemented for this a procedure that searches for a pattern and then realises a depth-first traversal of the tree (suffix or factor) until all the positions of the pattern are reached. This traversal is done in an iterative fashion using in both cases a static stack. At each position of pattern x in s, a function is called that does nothing.

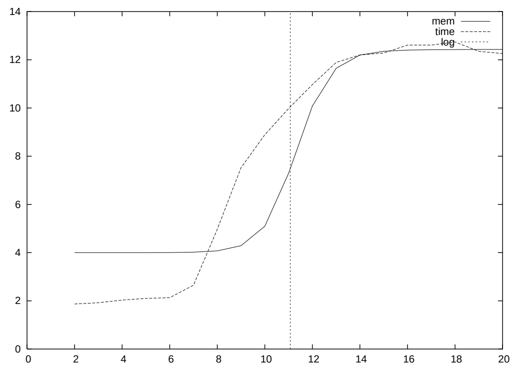


Figure 9: Results obtained with the factor tree on the sequence with EMBL primary access number *AE000111*. The Y-axis indicates the space occupied by the factor tree in bytes per input char and the execution time in seconds, the X-axis plots the values at which the tree is cut. The log curve represents $\log_4 |s|$.

Since a single search plus enumeration takes a time that is too small to be measured, for each pattern sought, we repeated the same procedure until we obtained an overall time of the order of a tenth of a second. We did this operation for 1000 patterns randomly taken among the factors of s (the same for the suffix and the factor trees).

The table below shows the results obtained. The columns indicate the following (execution times are in microseconds):

- name of the file;
- size of the file;
- length of the pattern sought;
- results obtained with the suffix tree using the *illi* coding:
  - minimum execution time of the operation;
  - average execution time of the operation;
  - maximum execution time of the operation;
- results obtained with the factor tree using our coding:
  - value of k;
  - minimum execution time of the operation;
  - average execution time of the operation;
  - maximum execution time of the operation;

file	length	pattern length	suffix tree			factor tree			
			min	mean	max	k	min	mean	max
bible(t)	4047393	10	2.30	14.90	368.34	17	3.01	12.28	183.40
bible(t)	4047393	10	2.30	14.90	368.34	10	2.03	7.80	13.75
bible(t)	4047393	15	2.53	8.29	134.81	17	3.50	8.79	17.54
bible(t)	4047393	15	2.53	8.29	134.81	15	1.99	8.66	15.95
world192(t)	2473401	15	1.81	10.72	77.03	20	1.28	8.71	39.07
world192(t)	2473401	25	1.97	8.90	70.02	25	0.0	8.59	18.69
cp(f)	24604	10	0.84	5.50	19.27	18	0.85	5.79	17.08
cp(f)	24604	10	0.84	5.50	19.27	12	0.99	5.08	13.43
trans(f)	93696	20	0.92	5.28	45.79	59	0.46	6.12	26.92
trans(f)	93696	40	1.05	5.28	39.33	40	1.36	6.01	11.82
pic(b)	513217	60	0.0	8423.35	23333.33	204	0.0	4782.99	12999.99
U00096	4639221	10	2.06	3.44	17.55	12	1.84	3.21	4.80
U00096	4639221	12	2.16	3.22	12.68	12	1.97	3.13	4.93

We see that the factor tree has search plus enumeration times that are equivalent and often better than those obtained using the suffix tree with the *illi* coding. It is interesting to observe that, in all cases, the maximal search time observed is greatly improved for a factor tree as against a suffix tree.

6 Conclusion

We introduced in this paper a useful structure for indexing the at most k-long factors of a string, and only those factors. The structure preserves the properties of a suffix tree, including the on-line characteristic of its construction, and can therefore replace it advantageously in some cases. We also showed that the modifications one must do to Ukkonen's construction in order to obtain a k-factor tree for a string s are easy.

Our experimental results show that this structure leads to often important gains in terms of the time for constructing the tree, memory space and time for some usages of the tree.

Acknowledgment

We would like to thank Maxime Crochemore and Mathieu Raffinot for their help and for interesting discussions.

References

[1] Arnold R. and Bell T. A Corpus for the Evaluation of Lossless Compression Algorithms. In *Proceedings of Data Compression Conference*, pages 201–210, 1997.

[2] Apostolico A. and Szpankowski W. Self-Alignement in Words and Their Applications. *J. Algorithms*, 13:446–467, 1992.

[3] Brazma A., Jonassen I., Vilo J. and Ukkonen E. Predicting gene regulatory elements *in silico* on a genomic scale. *Genome Research*, 11:1202–1215, 1998.

[4] Bell T.C., Cleary J.G. and Witten I.H. *Text Compression.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[5] Crochemore M. and Rytter W. *Text Algorithms.* Oxford University Press, 1994.

[6] Crochemore M. and Rytter W. *Jewels of Stringology.* World Scientific Pub Co., 2002.

[7] Dorohonceanu B. and Nevill-Manning C.G. Accelerating Protein Classification Using Suffix Tree. In *Proc. 8th Intl. Conference on Intelligent Systems for Molecular Biology ISMB 2000*, pages 128–133, 2000.

[8] Giegerich R. and Kurtz S. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353. 1997.

[9] Giegerich R., Kurtz S. and Stoye J. Efficient Implementation of Lazy Suffix Trees. In *Proc. of the Third Workshop on Algorithmic Engineering (WAE99)*, pages 30–42, 1999.

[10] European Bioinformatics Institute. EMBL Nucleotide Sequence Database *http://www.ebi.ac.uk/embl/.*

[11] Fiala E. R. and Greene D. H. Data compression with finite windows. *Commun. ACM*, 32: 490–505, 1989.

[12] Giegerich R., Kurtz S., Stoye J. Efficient Implementation of Lazy Suffix Trees. In *Proc. Third Worshop on Algorithm Engineering WAE' 99*, pages 30–42, 1999.

[13] Gusfield D. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press: Cambridge, 1997.

[14] Kurtz S. Reducing the Space Requirement of Suffix Trees. *Software: Practice and Experience*, 29:1149–1171, 1999.

[15] Larsson N. Structures of String Matching and Data Compression. *PhD thesis, Dept. of Comp. Science, Lund Univ.*, 1999.

[16] Larsson N. Extended Application of Suffix Trees to Data Compression. In *Proc. of the IEEE Data Compression Conference*, pages 190–199, 1996.

[17] Marsan L. and Sagot M.-F. Extracting structured motifs using a suffix tree – algorithms and application to promoter consensus identification. In *RECOMB*, pages 210–219, 2000.

[18] McCreight E.M. A space-economical suffix tree construction algorithm *J. Algorithms*, 23:262–272, 1976.

[19] Munro J.I., Raman V. and Srinivasa Rao S. Space Efficient Suffix Trees. *J. Algorithms*, 39:205–222, 2001.

[20] Na J.H., Park K. Data Compression with Truncated Suffix Trees. In *Data Compression Conference*; pages 565–565, 2000.

[21] National Center for Biotechnology Information *http://www.ncbi.nih.gov/*

[22] Sagot M.-F. Spelling approximate repeated or common motifs using a suffix tree. In *Proceedings of the Third Latin American Symposium*, Lucchesi C.L., Moura A.V. Springer–Verlag: Berlin, pages 374–390, 1998.

[23] Szpankowski W. A Generalized Suffix Tree and Its (Un)expected Asymptotic Behaviors. *SIAM J. Computing*, 22:1176–1198, 1993.

[24] Ukkonen E. On-line construction of suffix trees *Algorithmica*, 14:249–260, 1995.

[25] Vanet A., Marsan L. Sagot M.-F. Promoter sequences and algorithmical methods for identifying them. *Research in Microbiology*, 150:779–799, 1998.

[26] Weiner P. Linear pattern matching algorithm. In *Proc. of the 14th Annual IEEE Symposium on Switching and Automata Theory*, Washington DC, pages 1–11, 1973.