Theoretical Computer Science 407 (2008) 290–301

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On-line construction of compact suffix vectors and maximal repeats[☆]

Élise Prieur, Thierry Lecroq [*]

University of Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France

ARTICLE INFO

Article history:
Received 24 November 2006
Received in revised form 19 May 2008
Accepted 12 June 2008
Communicated by A. Apostolico

Keywords:
Suffix vectors
Suffix trees
Compact suffix automata
Design of analysis
Maximal repeats

ABSTRACT

A suffix vector of a string is an index data structure equivalent to a suffix tree. It was first introduced by Monostori et al. in 2001 [K. Monostori, Efficient computational approach to identifying overlapping documents in large digital collections, Ph.D. Thesis, Monash University, 2002; K. Monostori, A. Zaslavsky, H. Schmidt, Suffix vector: Space-and-time-efficient alternative to suffix trees, in: CRPITS'02: Proceedings of the 25th Australasian Computer Science Conference, vol. 4, Darlinghurst, Australia, 2002, Australian Computer Society, Inc. pp 157–166; K. Monostori, A. Zaslavsky, I. Vajk, Suffix vector: A space-efficient suffix tree representation, in: P. Eades, T. Takaoka (Eds.), Proceedings of the 12th International Symposium on Algorithms and Computation, in: Lecture Notes in Computer Science, vol. 2223, Springer-Verlag, Berlin, 2001, pp. 707–718. Christchurch, New Zealand]. They proposed a linear construction algorithm of an extended suffix vector, then another linear algorithm to transform an extended suffix vector into a more space-economical compact suffix vector. We propose an on-line linear algorithm for directly constructing a compact suffix vector. Not only do we show that it is possible to directly build a compact suffix vector but we will also show that this on-line construction can be faster than the construction of the extended suffix vector. Then we formalize the relation between suffix vectors and compact suffix automata which leads to an efficient method for computing maximal repeats using suffix vectors.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Indexes are data structures that are extensively used in pattern matching. An index for a string y contains all the substrings of y. The most famous index data structure is the suffix tree [2]. A suffix vector is an alternative data structure to a suffix tree. A suffix vector, for a string y, can store, in a reduced space, the same information as a suffix tree of y. Suffix vectors have been introduced by Monostori et al. [12–14] in order to detect plagiarism. The extended suffix vector of the string y consists of a succession of boxes located at some positions on the string y. These boxes are equivalent to the internal nodes (so called forks) of the suffix tree of y. Extended suffix vectors can be compacted. Compact suffix vectors can save up to 33% of the nodes compared to extended vectors [12]. Unfortunately, Monostori et al. only gave an on-line linear construction algorithm of an extended suffix vector and a linear algorithm to transform an extended suffix vector into a compact suffix vector.

In [16], we show the correspondence between suffix trees and suffix vectors. In [1], the authors show the correspondence between suffix trees and suffix arrays. Then compact suffix vectors could probably be built from suffix arrays but not in an on-line way.

☆ This work has been partially supported by the project "Informatique Génomique" of the program "MathStic" of the French CNRS.

* Corresponding address: Université de Rouen, LITIS, Faculté des Sciences et des Techniques, Place Emile Blondel, 76821 Mont-Saint-Aignan Cedex, France. Tel.: +33 235146581.

E-mail addresses: Elise.Prieur@univ-rouen.fr (É. Prieur), Thierry.Lecroq@univ-rouen.fr (T. Lecroq).

URL: http://monge.univ-mlv.fr/~lecroq (T. Lecroq).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.06.017

In this article, we propose an on-line linear algorithm for directly building a compact suffix vector. This is based on the notion of groups of nodes (nodes having the same transitions). This algorithm allows to deal with longer strings. Moreover, we show that this construction can be done faster than the construction of the extended suffix vector.

Then we formalize the relation between suffix vectors and compact suffix automata by showing that the groups of nodes of the suffix vector for a string y correspond exactly to the states of the compact suffix automaton for y.

The main advantage of the suffix vector compared to the suffix tree resides in the linear location of the boxes. We use this fact to present an efficient linear method for computing maximal repeats using compact suffix vectors.

This article is organized as follows: Section 2 introduces the different notations, quickly recalls suffix trees and defines suffix vectors; Section 3 presents the new on-line construction algorithm of a compact suffix vector; Section 5 explicits the correspondence between suffix vectors and compact suffix automata and Section 6 gives a simple linear method for computing maximal repeats with suffix vectors; Section 7 contains our conclusions and perspectives.

2. Notations and definitions

Let A be a finite alphabet. Throughout the article we will consider a string $y \in A^*$ of length n: $y = y[0..n-1]$. We assume without lost of generality that the symbol $y[n-1]$ does not occur in $y[0..n-2]$.

2.1. Suffix tree

2.1.1. Definition

The suffix tree $\mathcal{T}(y)$ of y is a well-known linear size index structure that contains all the suffixes of y. It can be constructed by considering the suffix trie of y (tree containing all the suffixes of y, the edges of which are labeled by exactly one symbol) where all internal nodes with only one child are removed and where remaining successive edge labels are concatenated. The leaves of the suffix tree contain the starting position of the suffix they represent.

The total length of all the suffixes of y can be quadratic, the linear size of the suffix tree is thus obtained by representing edge labels by pairs (*position*, *length*) referencing substrings $y[position..position + length - 1]$ of y. The terminator $y[n-1]$ ensures that no suffix of y is an internal substring of y and thus $\mathcal{T}(y)$ has exactly n leaves. Each internal node has at least two children, leading to at most $n-1$ internal nodes and thus to a linear number of nodes overall. This also gives a linear number of edges. Each edge requires a constant space. Altogether the suffix tree $\mathcal{T}(y)$ of y can be stored in linear size. Fig. 1 presents $\mathcal{T}(\texttt{aatttatttatta\$})$.

There exist several suffix tree construction algorithms. The first algorithm was designed by Weiner [19]. For a string y built on an alphabet A, two algorithms run in time $O(|y| \times \log |A|)$ [11,18] that extensively use the notion of suffix links. One algorithm runs in time $O(|y|)$ [6] when the alphabet is considered as a set of integers.

Each node p of the tree is identified with the substring obtained by concatenating the labels on the unique path from the root to the node p. We represent the existence of the edge from node p to node q with label (i, ℓ) by $\delta(p, (i, \ell)) = q$. We also consider $\textsc{Target}(p, a)$ which can be defined as $\delta(p, (i, \ell))$ for $y[i] = a$ and $\ell \geqslant 1$. For $a \in A$ and $u \in A^*$, if au is a node of $\mathcal{T}(y)$ then $s(au) = u$ is the suffix link of the node au.

For instance, in Fig. 1:

- node 7 in the tree is identified with atttatt,
- the edge going from node 3 to node 7 is $\delta(\texttt{att}, (4, 4)) = \texttt{atttatt}$,
- and $\textsc{Target}(\texttt{att}, \texttt{t}) = \delta(\texttt{att}, (4, 4)) = \texttt{atttatt}$.

The right position of the first occurrence of the string u in y is denoted by $rpos(u, y)$, for instance $rpos(\texttt{att}, \texttt{aatttatttatta\$}) = 3$.

2.1.2. Ukkonen's algorithm

Ukkonen's algorithm [18] is a linear on-line algorithm for constructing the suffix tree of a string. The suffix tree is initialized with $y[0]$ the first symbol of the string. The algorithm consists then of $n-1$ phases. Basically phase i consists in building the suffix tree of $y[0..i]$ from the suffix tree of $y[0..i-1]$. A phase is split in extensions. During the extension j of the phase i, the suffix $y[j+1..i]$ of $y[0..i]$ is inserted in the tree. The last substring inserted in the tree is denoted by $w = y[j+1..i-1]$. The algorithm is based on the three following rules.

Rule 1 In the tree, if the path corresponding to w leads to a leaf, it is sufficient to extend the label of the path with $y[i]$.

Rule 2 There exists a path in the tree labeled by w which does not lead to a leaf and such that it is impossible to continue it with $y[i]$. In this case, we have to create an edge labeled by $y[i]$ going out from w, if w does not correspond to a node it has to be added in the tree.

Rule 3 There exists a path in the tree labeled by w which does not lead to a leaf and such that it can be continued with $y[i]$, this means that $y[j+1..i]$ is already in the tree.

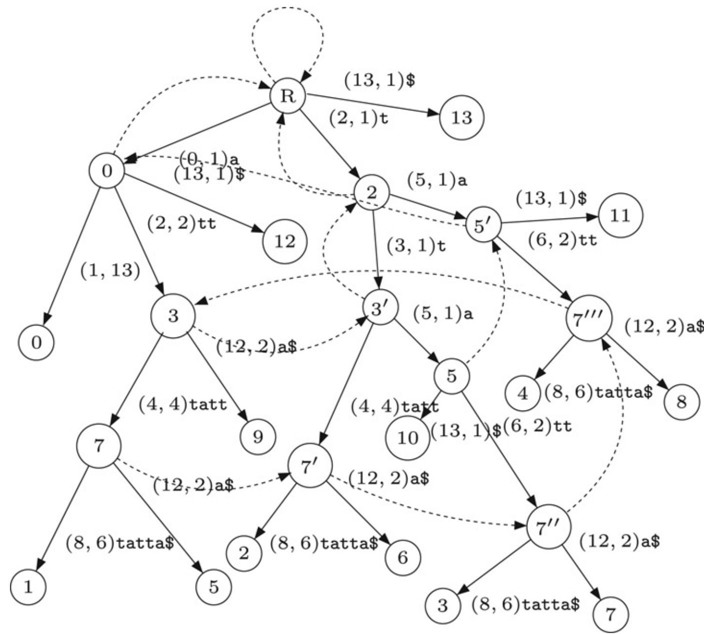

Fig. 1. Suffix tree of the string `aatttatttatta$`. All the labels of the edges are given here in both forms: (p, ℓ) and $y[p..p+\ell-1]$ in order to help the reader except for the label of the edge from node 0 to leaf 0 which corresponds to the string `atttatttatta$`. Suffix links are represented by dashed arrows.

Once a leaf is created, it always remains a leaf, so Rule 1 does not require any processing. Let j_ℓ be the number of the last created leaf corresponding to the suffix $y[j_\ell..n-1]$ of y. Phase i of the Ukkonen's algorithm consists of extensions from $j_\ell+1$ to the smallest $j > j_\ell$ such that Rule 3 applies, since if $y[j+1..i]$ is already in the tree so are all its suffixes.

There exist already several adaptations of Ukkonen's algorithm for different kind of suffix trees [4,9]. We will show how to adapt it for directly constructing compact suffix vector.

2.2. *Suffix vector*

2.2.1. *Extended suffix vector*

The suffix vector $\mathcal{V}(y)$ of y is a linear representation of the suffix tree $\mathcal{T}(y)$ consisting of a succession of boxes. These boxes contain the same information as the internal nodes of the tree, so that all the repeated substrings of y are represented in $\mathcal{V}(y)$. We will now give a description of the suffix vectors.

Let B_j be the box of the suffix vector at position j of the string y. The box B_j is considered as an array with $k \geqslant 1$ lines and 3 columns. It has been shown in [16] that each line of a box in the suffix vector is equivalent to an internal node in the suffix tree. In the present article, we will use the term *node* indifferently for a node of the tree and for a line of the vector. Each node p of the vector is identified with the substring obtained by concatenating the labels on the unique path from the root to the node p.

The first column of a box B_j contains the depth of the node, the second one contains the natural edge. The natural edge of a node p in a box B_j is the length $\ell = i - j$ such that i is the position of the box containing the node q such that $\textsc{Target}(p, y[j+1]) = q$ or q is a leaf.

The third column contains the edge lists L. Each edge of L is stored as a pair (b, ℓ) where b is the beginning of the edge (the position of the first symbol) and ℓ is the length of the edge (the position of the box containing the target node – if it is not a leaf – is thus $b+\ell-1$). So a box B is characterized by: $B[h, 0] = depth$, $B[h, 1] = ne$, $B[h, 2] = L$ for each $0 \leqslant h \leqslant k-1$.

Inside a box, there are implicit suffix links from the node represented by depth d to the node represented by depth $d-1$. Monostori pointed out in [12] that the depths in a box are continuous.

The root of the suffix tree is represented by a specific box in the suffix vector. The extended suffix vector of `aatttatttatta$` is given in Fig. 2.

2.2.2. *Compact suffix vector*

A suffix vector can be compacted when, for lines h_1 and h_2 of the box at position j, the edge list of line h_1 is included in the edge list of line h_2: $B_j[h_1, 2] \subseteq B_j[h_2, 2]$. In this case, we just need to store the list of the line h_2 and create a link from line h_1 to line h_2. When the edge lists of each line of the box B_j are the same, B_j is called a reduced box. In a reduced box, we store the deepest node and the number of nodes represented in the box. The compaction method is presented in Fig. 3. Nodes are ordered from the largest depth to the smallest depth. Nodes of a box are partitioned into groups: two nodes are in the same group if they have exactly the same edges. Fig. 4 presents the compact version of the suffix vector of Fig. 2.

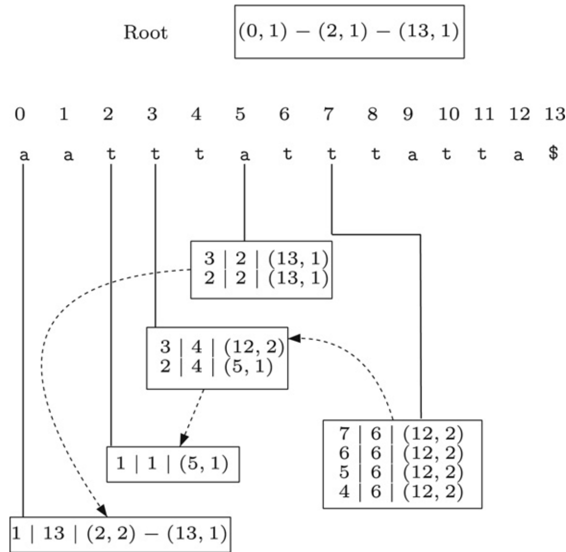

Fig. 2. Suffix vector of aatttatttatta$. Suffix links are represented by dashed arrows.

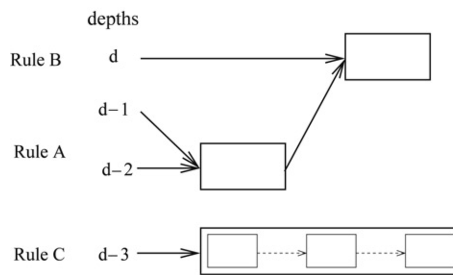

Fig. 3. Rules of compaction of a box. Rule A shows that nodes with exactly the same edge list can be merged into the same group of nodes. Rule B shows that some consecutive nodes can share some edges.

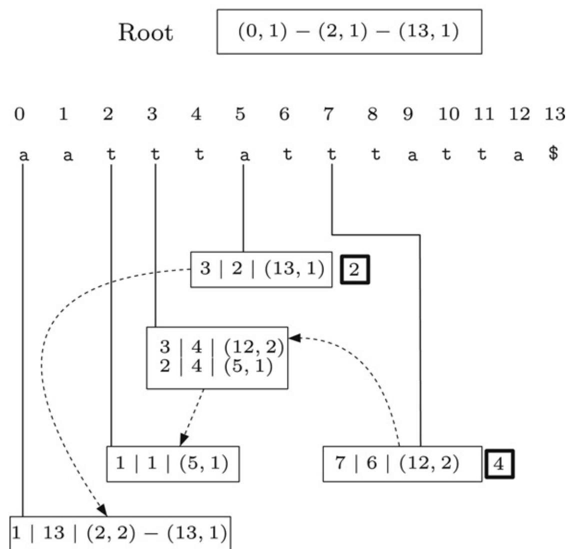

Fig. 4. Compact suffix vector of the string aatttatttatta$. Boxes at position 5 and 7 are reduced. The number of nodes in these two reduced boxes is indicated at the right of the boxes.

3. On-line construction of a compact suffix vector

The construction algorithm of the suffix vector given by Monostori *et al.* is based on Ukkonen's algorithm for suffix trees (cf. 2.1.2). The main difference in our algorithm is that we are able to skip some extensions when we add an edge to a box. The following proposition explains the general situation. Let B_p be the box at position p, let $y[p-d+1..p]$ be the substring representing the node of depth d in B_p to which the edge will be added, let D be the depth of the deepest node in B_p and let nb be the number of nodes in B_p, $1 \leqslant nb \leqslant D$.

Proposition 3.1 allows to add the edge beginning by $y[i]$ only once for a group of nodes.

Proposition 3.1. *If an edge is added to the node h of depth d in a box B_p, this edge will be added to all the nodes of depth smaller than d in the group of nodes of node h.*

Proof. Let d' be the smallest depth of the nodes in the group of nodes of h. All the nodes of depth within d and d' have exactly the same edges. Considering the box at position p, we know that p is the end position of the first occurrence of each substring $y[p'..p]$ such that $p-d+1 \leqslant p' \leqslant p-d'+1$, the length of these substrings are smaller or equal to d.

We assume that during the phase i, we add an edge labeled by $y[i]$ to the node $y[p-d+1..p]$. This means that i is the position of the first occurrence of the substring $y[p-d+1..p]y[i]$ and that $y[p-d+1..p]$ has no edge beginning by $y[i]$. Therefore, we can say that i is the position of the first occurrences of all the substrings $y[p'..p]y[i]$ with $p-d+1 \leqslant p' \leqslant p-d'+1$. So the edge beginning by $y[i]$ will be added to each node of depth within d and d' of B_p during consecutive extensions of the phase i. □

When an edge is added to the node h of depth d in a box B_p, if h is not the deepest node of its group, the group has to be split into two. In this case if B_p is a reduced box, it has to be extended. The edge labeled by $y[i]$ is added only at the beginning of the edge list of the group node of h.

The next corollary allows to add the edge beginning by $y[i]$ only once for a reduced box and to jump from extension $k-D+1$ to extension $k-D+nb$ during the phase i.

Corollary 3.2. *If an edge is added to the deepest node of a reduced box, this edge will be added for each node of the box, then the box is still reduced.*

The algorithm BuildSV(y) presented in Fig. 5 builds on-line the compact suffix vector of y. It is an adaptation of Ukkonen's algorithm for suffix vectors. The on-line construction of compact suffix vectors is similar to the on-line construction of suffix trees except for the algorithm AddEdge (see Fig. 11) called by algorithm BuildSV(y). The algorithm AddEdge implements the results of Proposition 3.1 and Corollary 3.2.

All the algorithms use the following global variables whose meaning is shown in Fig. 6:

- k is the length of $w = y[j_\ell + 1..i-1]$;
- q is the position of the node representing u which is the longest prefix of w corresponding to a node;
- r is the beginning position of v which is such that $w = uv$;
- p is the end position of the first occurrence of w.

We now describe briefly the functions used in the construction algorithm given in Fig. 5. The function InitRoot initializes the root with an edge labeled by $y[0]$. The function FastScan is the same function as in the construction algorithms of the suffix tree (McCreight [11], Ukkonen[18]). The only difference is that it looks through a suffix vector instead of a suffix tree. The function AddToRoot(i) (see Fig. 7) tests if there is an edge by $y[i]$ going out from the root. If it does not exist, it adds it and increments j_ℓ by 1. The function AddNode(pos, w, i) (see Fig. 8) tests if there is a box at position p in the vector with a node representing w. If there is not, it creates this box with the correct node. If the box at position p exists and is a reduced box then if the length of w is larger than the depth of the deepest node in B_p, it has to extend the box before adding the node, otherwise it just has to increment the number of nodes in the box. If the box is extended, it adds the node. Adding a node in an extended box means adding a line in it. The function AddNode calls the function BreakEdge given Fig. 9. In the function UpdateSL($fork$) (see Fig. 10), the suffix link of the fork becomes the node w and then the fork becomes w where w is the last created or modified node. All these functions are adapted to suffix vectors and behave as in the Ukkonen's algorithm.

The function AddEdge in Fig. 11 implements the results of Proposition 3.1 and Corollary 3.2. It adds the edge beginning by $y[i]$ to the edge list of the node w of depth k in B_p. If B_p is a reduced box and its deepest node is w, Corollary 3.2 allows B_p to remain reduced and to add only once the edge (lines 4 and 10 of the algorithm in Fig. 11). If B_p is reduced and w is not its deepest node then B_p has to be extended and the edge has to be added once for all the nodes in the group of w. If B_p is not reduced, then the edge has to be added once for all the nodes in the group of w. In all cases, α is set with the number of nodes in the group of w, j_ℓ is incremented by α enabling to skip the $\alpha - 1$ next extensions. This requires to follow α suffix links (line 11). The function AddEdgeNodes(i) adds the edge to all the nodes of depth smaller or equal to d in the group of w, this group is split if necessary. The function AddEdgeBox(i) adds the edge to the edge list of the reduced box at position p.

We can then give the following result.

Theorem 3.3. *The algorithm* BuildSV(y) *builds on-line the compact suffix vector of a string y in linear time.*

Proof. The correctness and the time complexity of the algorithm come from the fact that the construction is based on Ukkonen's algorithm and on Proposition 3.1 and Corollary 3.2. □

```
BuildSV(y)
 1  (k, q, r, j_ℓ, p) ← (0, −1, −1, 0, −1)
 2  InitRoot()
 3  for i ← 1 to n − 1 do
        ▷ Phase i
 4      fork ← −1
 5      while j_ℓ < i do
            ▷ Extension j_ℓ
 6          FastScan()
 7          if k = 0 then
                ▷ j_ℓ = i − 1
 8              AddToRoot(i)
 9          else if (y[i] ≠ y[p + 1]) and
                (∄ node w in B_p or (∃ node w and ∄ edge by y[i])) then
                ▷ Rule 2
10              if ∄ node w in B_pos then
11                  AddNode(i)
12              else AddEdge(i)
13              fork ← updateSL(fork)
14          else Rule 3
15              if y[i] = y[p + 1] then
16                  p ← p + 1
17              else p ← beginning position of the edge by y[i]
18              fork ← updateSL(fork)
19              break
```

Fig. 5. On-line construction algorithm of the compact suffix vector of the string y. The variable j_ℓ is incremented in functions AddToRoot, AddNode and AddEdge.

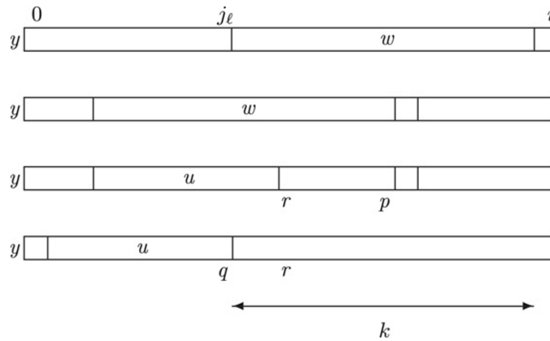

Fig. 6. Global variables used in the extension j_ℓ of phase i of algorithms BuildSV and AddEdge.

```
AddToRoot(i)
 1  if ∄ edge from the root by y[i] then
 2      Root ← Root ∪ (i, ∞)
 3      j_ℓ ← j_ℓ + 1
```

Fig. 7. AddToRoot.

```
AddNode(pos, w, i)
 1 if B_pos does not exist then
 2   CreateBox(pos, w, i)
 3 else if B_pos is a reduced box then
 4     if w > B_pos.D then
 5       ExtendBox(B_pos)
 6       AddNodeEB(pos, w, i)
 7     else AddNodeRB(pos)
 8   else AddNodeEB(pos, w, i)
 9 BreakEdge(u, v, pos)
10 j_ℓ ← j_ℓ + 1
```

Fig. 8. AddNode.

```
BreakEdge(u, v, pos)
1 if u = root then
2   BreakEdgeRoot(root, v, pos, B_pos.ne)
3 else if B_pos is a reduced box then
4     if w = B_pos.D then
5       BreakEdgeRB(u, v, pos, B_pos.ne)
6     else ExtendBox(B_pos)
7       BreakEdgeEB(u, v, pos, B_pos.ne)
8   else BreakEdgeEB(u, v, pos, B_pos.ne)
```

Fig. 9. BreakEdge.

```
UpdateSL(fork, pos)
1 s(fork) ← pos
2 fork ← pos
3 return fork
```

Fig. 10. Update suffix link.

4. Implementation

We will show experimental results on the space and the time needed for the construction of the compact suffix vectors. First, we introduce the data structure that we specially designed for that purpose.

4.1. Data structures

We describe here the structures used in our implementation of the compact suffix vector. Integer values are stored with 1, 2, 3 or 4 bytes, to do that we used 2 indicator bits. Thus only 6 bits are available on the first byte.

We add to the structure of the box a data byte which reduces the space used for the values to be stored. Let us give the role of each bit in this byte. The first bit is used to indicate if the box is a reduced one, the second one is set if the suffix link is implicit (*i.e.* the smallest depth of the box is 1) and the third one indicates if the natural edge leads to a leaf. Two bits are used to give the number of bytes needed to store the greatest depth of the box, and two others for the number of bytes used to store the number of nodes in the box. Fig. 12 presents the data byte of a box.

The suffix link and the natural edge are stored on 1, 2, 3 or 4 bytes (using each two indicator bits) only if they are not implicit. The box also contains the address of the first edge in the list and, if it is an extended box, the address of the first group of nodes. Fig. 13 shows the details of the information in a box.

The structure used for the groups of nodes is nearly the same. The suffix link is always implicit for a node so it is not considered in its data byte. The first bit is now used to indicate if the group of nodes is the last one of the list. Two bits are used to indicate the size of the natural edge. Fig. 14 presents the data byte of a group of nodes.

AddEdge(i)

1 **if** B_p is reduced **then**
2   **if** k = depth of the deepest node in B_p **then**
3   AddEdgeBox(i)
4     $\alpha \leftarrow$ number of nodes in B_p
5   **else** ExtendBox(B_p)
6   AddEdgeNodes(i)
7     $\alpha \leftarrow$ number of nodes between w and the node of smallest depth in the group of w
8 **else** AddEdgeNodes(i)
9   $\alpha \leftarrow$ number of nodes between w and the node of smallest depth in the group of w
10 $\bar{j_\ell} \leftarrow j_\ell + \alpha$
11 $q \leftarrow s^\alpha(q)$

Fig. 11. Algorithm AddEdge that implements the result of Proposition 3.1 and Corollary 3.2. It enables to skip some extensions.

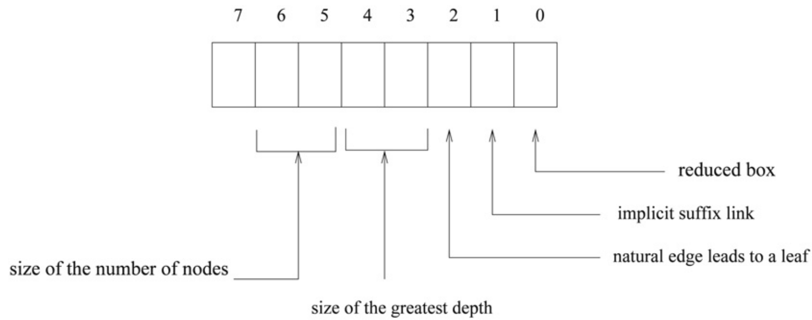

Fig. 12. Data byte of a box.

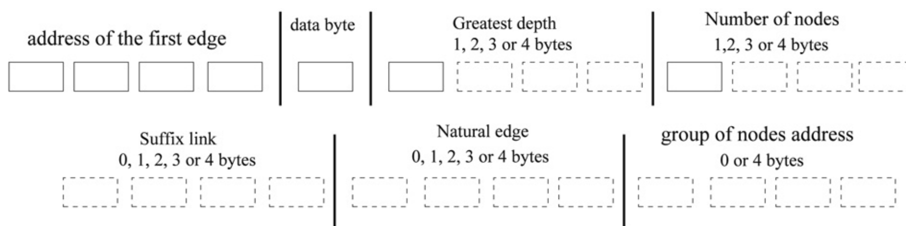

Fig. 13. Representation of a box.

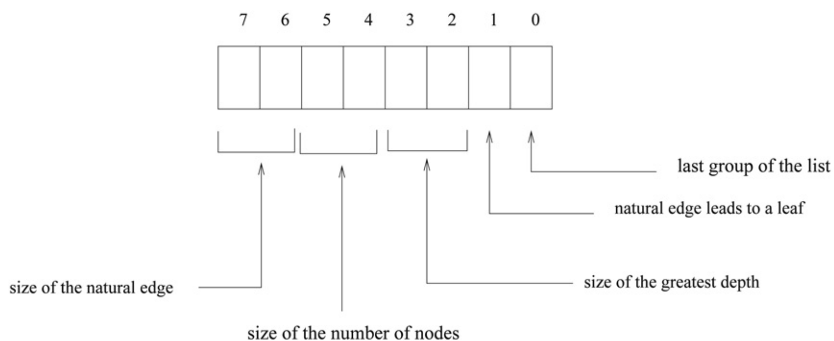

Fig. 14. Representation of the data byte of a group of nodes.

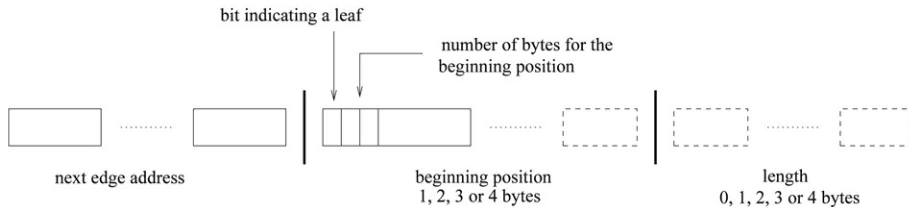

Fig. 15. Representation of an edge.

Table 1
Number of nodes in the suffix vectors

Files	File size (bytes)	Number of nodes	Number of groups of nodes
paper1	53,162	29,037	15,286
paper2	82,200	43,210	25,517
bible	4,047,392	2,239,836	1,099,794
world192	2,473,401	1,337,299	592,694
alice29	152,090	80,856	48,252
progc	39,612	21,174	10,480
E.coli	4,638,691	2,978,795	2,633,146

Table 2
Compact suffix vector size

File name	File size (bytes)	bytes/symbol (Monostori)	Compact suffix vector size (Prieur, Lecroq)	bytes/symbol (Prieur, Lecroq)
paper1	53,162	8.82	428,795	**8.06**
paper2	82,200	9.10	714,307	**8.69**
bible	4,047,392	8.53	31,526,310	**7.79**
world192	2,473,401	7.68	16,635,165	**6.73**
alice29	152,090	9.15	1,355,955	**8.92**
progc	39,612	8.63	297,882	**7.52**
E.coli	4,638,691	**12.51**	68,376,273	14.74

An edge contains 3 types of information, its beginning position, its length and the address of the next edge. The beginning position is stored on 1, 2, 3 or 4 bytes using two indicator bits. Another bit is used to indicate if it leads to a leaf. In this case, the length is not stored. Otherwise, it is stored on 1, 2, 3 or 4 bytes. The representation of an edge is given in Fig. 15.

4.2. Experimental results

The tests presented in this section were done on a personal computer with a 3.2 GHz processor and 2GB of RAM. Files come from the Calgary Corpus and Canterbury Corpus.

Table 1 contains the number of nodes and the number of groups of nodes for several files. It shows the gain obtained by storing the groups instead of the nodes.

Table 2 gives the compact suffix vector size with our implementation and with Monostori's one (see [12]). This table shows that our implementation is more space-economical than Monostori's except for *Escherichia coli*. All the experiments that we conducted on DNA sequences leads to a compact suffix vector using within 14 and 15 bytes per symbol. Nevertheless, we directly construct the compact suffix vector whereas Monostori constructs first the extended suffix vector. The size of the extended suffix vector is much bigger than the size of the compact suffix vector. We are thus able to deal with larger sequences.

Regarding the time needed for the construction, Fig. 16 shows the linearity of our algorithm. The strings were generated randomly on an alphabet of size 4. The given time is an average time obtained with 100 strings of each length.

5. Equivalence between a compact suffix automaton and a compact suffix vector

In this section, we show the equivalence between the compact suffix automaton [5] (also called CDAWG for Compact Directed Acyclic Word Graph) and the compact suffix vector by giving the equivalence between the states of the compact suffix automaton and the group of nodes of the compact suffix vector. Let us denote by $Endpos_y(x) = \{k \mid y = zxy[k+1..n-1]\}$ the set of end positions of the substring x in y.

Definition 5.1. Let x_1 and x_2 be two strings on A^*, the equivalence relation $\mathcal{R}_y$ is defined by $x_1 \,\mathcal{R}_y\, x_2 \iff Endpos_y(x_1) = Endpos_y(x_2)$.

Definition 5.2. The equivalence class $Cl_{\mathcal{R}_y}(x)$ of the string x for relation $\mathcal{R}_y$ is defined by $Cl_{\mathcal{R}_y}(x) = \{x' \mid x' \,\mathcal{R}_y\, x\}$.

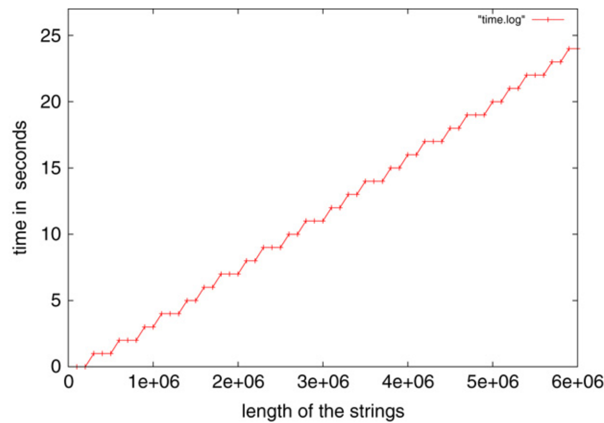

Fig. 16. Time of the direct construction algorithm of the compact suffix vector for random strings. Time is given depending on the length of the strings.

Definition 5.3. Let u be a substring in a $Cl_{\mathcal{R}_y}(x)$, if there exist at least two different characters a and b such that ua and ub are substrings of y, the class $Cl_{\mathcal{R}_y}(x)$ is called a strict class.

Except for final states, each state of a compact suffix automaton represents a strict class.

The two following propositions show that each substring of a strict class is represented in the same box.

Proposition 5.4. *Let u be a substring of y, if there exists a box in $\mathcal{V}(y)$ with a node representing u then u has at least two occurrences in y.*

Proof. During the construction of the compact suffix vector, the node representing a substring u is created during the extension j of a phase i. It is added into the box at position $rpos(u) = p < i$ such that $u = y[p-i+j+2..p] = y[j+1..i-1]$ and $y[i] \neq y[p+1]$. □

The next proposition shows the relation between the equivalence classes and the boxes of the suffix vector.

Proposition 5.5. *Each substring of a class $Cl_{\mathcal{R}_y}(x)$ is represented in the same box of the suffix vector. This is the box at position p such that $p = min\{k \mid k \in Endpos_y(x)\}$.*

Proof. All the substrings of an equivalence class have the same end positions. A repeated substring is represented in a box of the suffix vector at position p such that p is the end position of its first occurrence. Each substring of a class have the same first end position. □

In Section 2.2.2, we define a group of nodes as a set of nodes which are in the same box and have exactly the same edges. The next proposition shows the equivalence between a group of nodes and a state of the compact suffix automaton.

Proposition 5.6. *Substrings of y are represented in the same box and are in the same strict equivalence class if and only if they are in the same group of nodes.*

Proof. $\Longrightarrow$ Suppose that u and v are substrings of y representing nodes in the same box such that $Endpos_y(u) = Endpos_y(v)$ and the nodes are not in the same group of nodes. If they are not in the same group of nodes, they do not have the same edges, so there occurrences have different positions. This is in contradiction with the fact that $Endpos_y(u) = Endpos_y(v)$. So they must be in the same group.

$\Longleftarrow$ Suppose that u and v are substrings of y representing nodes in the same group of nodes and which are not in the same equivalence class. As they are not in the same equivalence class, $Endpos_y(u) \neq Endpos_y(v)$. So the nodes representing u and v do not have the same edges, so they cannot be in the same group of nodes. This is in contradiction with their definitions, so they must be in the same class. □

The usual schema on indexing structures can be updated as in Fig. 17. The suffix trie can be compacted which gives the suffix tree, or minimized (in the sense of automata theory) which gives the suffix automaton. Minimizing the suffix tree or compacting the suffix automaton leads to exactly the same structure known as compact suffix automaton. The suffix vector can be seen as an even more compacted representation of the compact suffix automaton.

6. Maximal repeats

The problem of detecting repeated substrings is important in many fields such as computational biology. A maximal repeat is a repeat which cannot be extended to the left nor the right. We will first recall some previous results on maximal repeats and then show the relation between maximal repeats and suffix vectors.

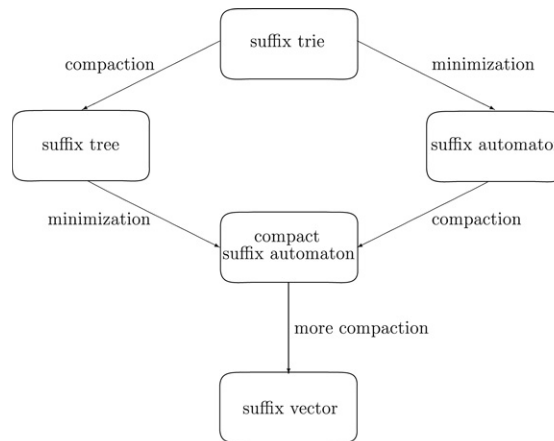

Fig. 17. Usual schema on indexing structures.

6.1. Definitions

Let us assume w.l.o.g. that $y[0]$ is unique in y.

Definition 6.1. A maximal repeat in a string y is a substring u such that there exist at least two occurrences of u, a_1ub_1 and a_2ub_2 with $a_1, a_2, b_1, b_2 \in A$, $a_1 \neq a_2$ and $b_1 \neq b_2$.

Raffinot has shown in [17] the next theorem.

Theorem 6.2 ([17]). *A substring is a maximal repeat if and only if it is the longest string in a strict equivalence class.*

The next two results make the link between a maximal repeat and the corresponding line in the suffix vector. They show that a node represents a maximal repeat if and only if this is the deepest node of a group of nodes.

Proposition 6.3. *In a box, if consecutive nodes have exactly the same edges, only the deepest one represents a maximal repeat.*

Proof. If some nodes are in the same box with exactly the same edges, this means that they have exactly the same number of occurrences and end positions, so they are in the same equivalence class. Applying Theorem 6.2 and Proposition 5.5, we can affirm that only the deepest of these nodes represents a maximal repeat. □

Corollary 6.4. *The substring represented by the deepest node of a reduced box is a maximal repeat and it is the only one in this box.*

Proof. Each substring represented in a reduced box have exactly the same edges. By Proposition 6.3, we can say that the substring represented by the deepest node of the reduced box is a maximal repeat and it is the only one in the box. □

Raffinot [17] has demonstrated Proposition 6.5 for the compact suffix automaton. A proof of this proposition for the suffix trees is also given in [7]. Here, we show it for the compact suffix vector.

Proposition 6.5. *The number of maximal repeats of a string y of length n is within* 0 *and* $n - 2$.

Proof. A suffix tree can have at most $n - 1$ internal nodes including the root. As lines of a suffix vector are equivalent to internal nodes of a suffix tree, it can have at most $n - 2$ lines in boxes. A maximal repeat must be represented by a node. So, there are at most $n - 2$ maximal repeats in a string of length n. □

6.2. Computing maximal repeats with suffix vector

Proposition 6.3 and Corollary 6.4 give a method for computing the maximal repeats in a string y with its suffix vector. For $0 \leq j < n$, each box B_j of the vector is looked up only once and the deepest node of each box represents a maximal repeat. If B_j is a reduced box, it is the only one, in the other case each deepest node of a group of nodes is a maximal repeat.

Example 1. In the suffix vector of Fig. 4, the boxes at positions 0, 2, 5 and 7 are reduced boxes so only their deepest nodes are maximal repeats: `a`, `t`, `tta` and `atttatt`. The box at position 3 is extended and the nodes have different edges, so the two nodes represent maximal repeats: `att`, `tt`.

Example 2. The compact suffix automaton of `gtagtaaac` is given in [17] and Fig. 18. The suffix vector of `gtagtaaac` is given in Fig. 18. The box at position 2 has two nodes with the same edges, therefore they are in the same equivalence class $Cl_{\mathcal{R}_y}(\texttt{gta})$, so `gta` is a maximal repeat. The other line of B_2 and the box at position 6 give two equivalence classes $Cl_{\mathcal{R}_y}(\texttt{a})$ and $Cl_{\mathcal{R}_y}(\texttt{aa})$, so `a` and `aa` are maximal repeats.

It should be noted that the presented method allows to compute maximal repeats but not directly the maximal repeats in pairs.

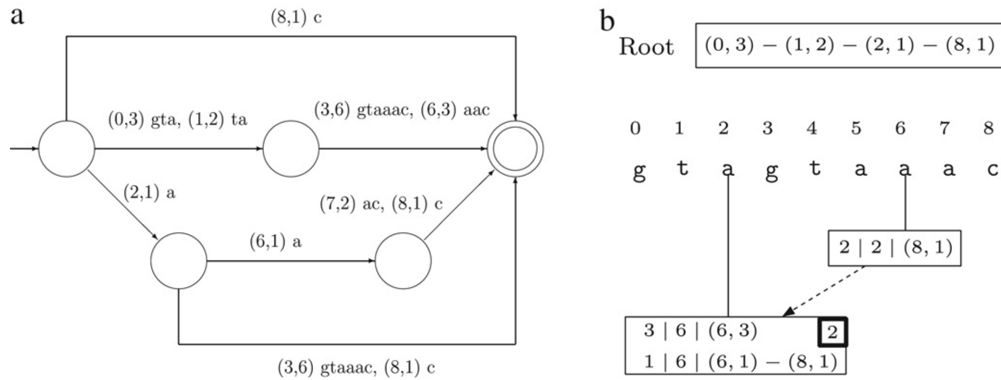

Fig. 18. (a) Compact suffix automaton of gtagtaaac. All the labels of the edges are given here in both forms: (p, ℓ) and $y[p..p + \ell - 1]$ in order to help the reader. (b) Compact suffix vector of gtagtaaac, the first line of the box at position 2 is a group of nodes containing 2 nodes.

7. Conclusion and perspectives

This article presents an on-line linear algorithm for building the compact suffix vector for a string based on Ukkonen's algorithm for building suffix trees. This avoids the construction of the extended suffix vector which is more space consuming. Moreover, the on-line construction of the compact suffix vector enable to skip some extensions. This structure is very helpful for computing maximal repeats in strings.

A practical study for measuring the space and time performances of compact suffix vectors remains to be done in order to compare them to suffix trees [10,15], CDAWGs [8] and LZ-indexes [3].

References

[1] M. Abouelhoda, S. Kurtz, E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, Journal of Discrete Algorithms 2 (1) (2004) 53–86.
[2] A. Apostolico, The myriad virtues of suffix trees, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, in: NATO Advanced Science Institutes, Series F, vol. 12, Springer-Verlag, Berlin, 1985, pp. 85–96.
[3] D. Arroyuelo, G. Navarro, Space-efficient construction of LZ-index, in: X. Deng, D.-Z. Du (Eds.), Proceedings of the 16th International Symposium on Algorithms and Computation, in: Lecture Notes in Computer Science, vol. 3827, Springer-Verlag, Berlin, 2005, pp. 1143–1152. Sanya, Hainan, China.
[4] R. Clifford, M. Sergot, Distributed and paged suffix trees for large genetic databases, in: R.A. Baeza-Yates, E. Chávez, M. Crochemore (Eds.), Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 2676, Springer-Verlag, Berlin, 2003, pp. 70–82. Morelia, Michocán, Mexico.
[5] M. Crochemore, R. Vérin, Direct construction of compact directed acyclic word graphs, in: A. Apostolico, J. Hein (Eds.), Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 1264, Springer-Verlag, Berlin, 1997, pp. 116–129. Aarhus, Denmark.
[6] M. Farach, Optimal suffix tree construction with large alphabets, in: Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science, Miami Beach, FL, 1997, pp 137–143.
[7] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[8] J. Holub, M. Crochemore, On the implementation of compact DAWG's, in: J.-M. Champarnaud, D. Maurel (Eds.), Proceedings of the 7th International Conference on Implementation and Application of Automata 2002, in: Lecture Notes in Computer Science, vol. 2608, Springer-Verlag, Berlin, 2003, pp. 289–294. Tours, France.
[9] S. Inenaga, M. Takeda, On-line linear-time construction of word suffix trees, in: M. Lewenstein, G. Valiente (Eds.), Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 4009, Springer-Verlag, Berlin, 2006, pp. 60–71. Barcelona, Spain.
[10] S. Kurtz, Reducing the space requirements of suffix trees, Software — Practice & Experience 29 (13) (1999).
[11] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of Algorithms 23 (2) (1976) 262–272.
[12] K. Monostori, Efficient computational approach to identifying overlapping documents in large digital collections, Ph.D. Thesis, Monash University, 2002.
[13] K. Monostori, A. Zaslavsky, H. Schmidt, Suffix vector: Space-and-time-efficient alternative to suffix trees, in: CRPITS'02: Proceedings of the 25th Australasian Computer Science Conference, vol. 4, Australian Computer Society, Inc., Darlinghurst, Australia, 2002, pp. 157–166.
[14] K. Monostori, A. Zaslavsky, I. Vajk, Suffix vector: A space-efficient suffix tree representation, in: P. Eades, T. Takaoka (Eds.), Proceedings of the 12th International Symposium on Algorithms and Computation, in: Lecture Notes in Computer Science, vol. 2223, Springer-Verlag, Berlin, 2001, pp. 707–718. Christchurch, New Zealand.
[15] J. Ian Munro, Venkatesh Raman, S. Srinivasa Rao, Space efficient suffix trees, in: V. Arvind, R. Ramanujam (Eds.), Proceedings of the Foundations of Software Technology and Theoretical Computer Science, in: Lecture Notes in Computer Science, vol. 1530, Springer-Verlag, Berlin, 1998, pp. 186–196. Chennai, India.
[16] É Prieur, T. Lecroq, From suffix trees to suffix vectors, International Journal of Foundations of Computer Science 17 (6) (2006) 1385–1402.
[17] M. Raffinot, On maximal repeats in strings, Information Processing Letters 80 (3) (2001) 165–169.
[18] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[19] P. Weiner, Linear pattern matching algorithm, in: Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory, Washington, DC, 1973, pp. 1–11.