On-line Construction of compact suffix vectors and maximal repeats *

Élise Prieur and Thierry Lecroq

LITIS, University of Rouen, France,
{Elise.Prieur,Thierry.Lecroq}@univ-rouen.fr

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 [9, 10, 11]. 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 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. Finally, we present an efficient method for computing maximal repeats using suffix vectors.

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. 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.* ([9, 10, 11]) 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 [9]. 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 [13], we show the correspondance between suffix trees and suffix vectors. In [1], the authors show the correspondance between suffix trees and suffix arrays. Then compact suffix vectors could probably be built from suffix arrays but not in an on-line way.

In this article, we propose an on-line linear algorithm for directly building a compact suffix vector. This algorithm allows to deal with longer strings. Moreover, we show that this construction can be done faster than the construction of

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

the extended suffix vector. 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 4 gives a linear method for computing maximal repeats with suffix vectors; Section 5 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 which edges 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 $(\mathit{position}, \mathit{length})$ referencing substrings $y[\mathit{position}..\mathit{position} + \mathit{length} - 1]$ 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. Figure 1 presents $\mathcal{T}(\texttt{aatttatttatta\$})$.

There exist several suffix tree construction algorithms. For a string y built on an alphabet A, two algorithms run in time $O(|y| \times \log |A|)$ [8, 15] that extensively use the notion of suffix links. One algorithm runs in time $O(|y|)$ [4] 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 is 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.

2.1.2 Ukkonen's algorithm

Ukkonen's algorithm ([15]) 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, phase i consists in building the suffix tree of $y[0..i]$ from the suffix tree of $y[0..i-1]$. A phase is

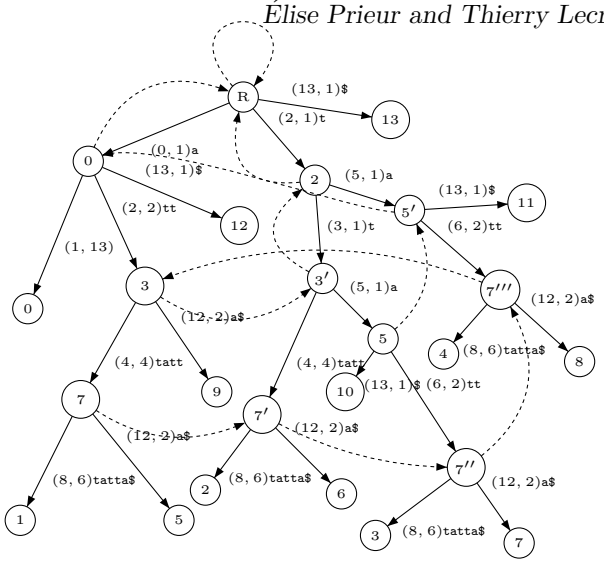

Figure 1: Suffix tree of the string `aatttatttatta$`. All the labels of the edges are given here in both forms: (p, ℓ) and $y[p..p+\ell-1]$ in order to help the reader except for the label of the edge from node 0 to leaf 0 which corresponds to the substring `atttatttatta`$. Suffix links are represented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

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 If, in the tree, 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 we can continue it with $y[i]$, this means that $y[j+1..i]$ is already in the tree.

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 [3, 6].

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 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 [13] 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 *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 position of the box containing the node q such that $\textsc{Target}(p, y[j+1]) = q$.

The third column contains the edge lists L. Each edge of L is stored as a pair (b, e) where b is the beginning of the edge (the position of the first symbol) and e the end of the edge (the position of the box containing the target node). So a box is characterized by: $B[h, 0] = \mathit{depth}, B[h, 1] = \mathit{ne}, B[h, 2] = L$ for each $0 \leqslant h \leqslant k - 1$.

Inside a box, there are implicit suffix links from node represented by depth d to node represented by depth $d - 1$. Monostori pointed out in [9] 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 Figure 2.

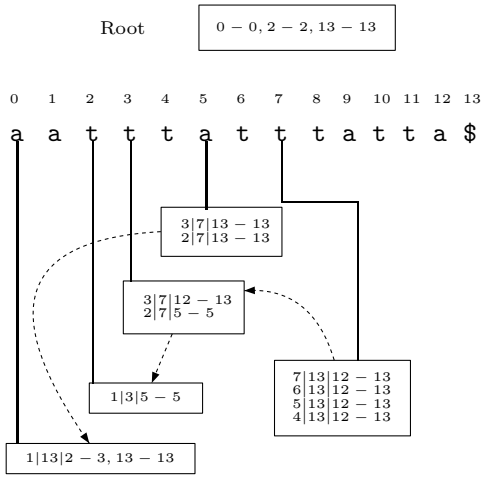

Figure 2: Suffix vector of `aatttatttatta$`. Suffix links are represented by dashed arrows.

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 Figure 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. Figure 4 presents the compact version of the suffix vector of Figure 2.

3 On-line construction of a compact suffix vector

The construction algorithm of the suffix vector 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 1 allows to add the edge beginning by $y[i]$ only once for a group of nodes.

Proposition 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. □

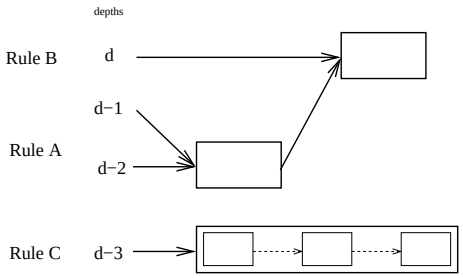

Figure 3: Rules of compaction of a box. Rule A shows that nodes with exactly the same edge list can be merged into the same group of nodes. Rule B shows that some consecutive nodes can share some edges.

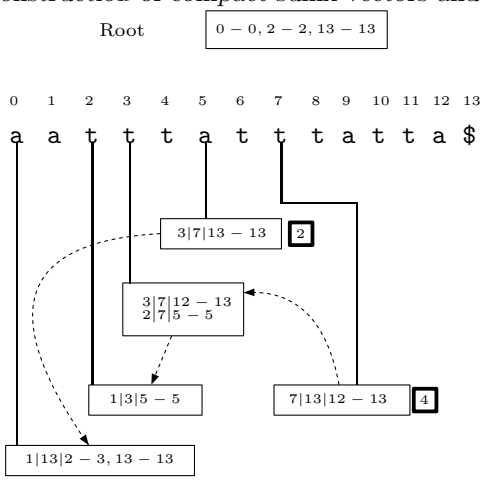

Figure 4: Compact suffix vector of the string `aatttatttatta`\$.

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 the case where 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 1 *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 Figure 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 Figure 6) called by algorithm BuildSV(y). The algorithm AddEdge implements the results of Prop. 1 and Coro. 1.

All the algorithms use the following global variables whose meaning is shown in Figure 7:

- 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 Figure 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 [8], Ukkonen[15]). The only difference is that we look through a suffix vector instead of a suffix tree. The function AddToRoot tests if there is an edge by $y[i]$ going out from the root.

```
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 ℓ = 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
```

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

If it does not exist, we add it and increment j_ℓ by 1. The function AddNode tests if there is a box at position p in the vector with a node representing w. If there is not, we create 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, we have to extend the box before adding the node, otherwise we just have to increment the number of nodes in the box. If the box is extended, we add the node. Adding a node in an extended box means adding a line in it. In the function UpdateSL, 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 Figure 6 implements the results of Prop. 1 and Coro. 1. 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, Coro. 1 allows B_p to remain reduced and to add only once the edge (lines 4 and 10 of the algorithm in Figure 6). 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.

```
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 |  |  α ← number of nodes in B_p
 5 |  else ExtendBox(B_p)
 6 |  |  AddEdgeNodes(i)
 7 |  |  α ← number of nodes between w and
   |  |    the node of smallest depth in the group of w
 8 else AddEdgeNodes(i)
 9 |  α ← number of nodes between w and
   |    the node of smallest depth in the group of w
10 j_ℓ ← j_ℓ + α
11 q ← s^α(q)
```

Figure 6: Algorithm AddEdge that implements the result of Prop. 1 and Coro. 1. it enables to skip some extensions.

Theorem 1 *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 Prop. 1 and Coro. 1. □

4 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

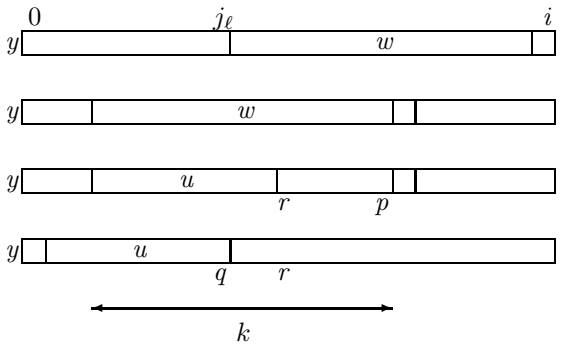

Figure 7: Global variables used in the extension j_ℓ of phase i of algorithms BuildSV and AddEdge.

to the left or the right. We will first recall some previous results on maximal repeats and then show the relation between maximal repeats and suffix vectors.

4.1 Definitions

Definition 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 $b_1 \neq b_2$.*

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 2 *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 3 *The equivalence class $Cl_{\mathcal{R}_y}(x)$ of the string x for relation $\mathcal{R}_y$ is defined by $Cl_{\mathcal{R}_y}(x) = \{x' \mid x' \mathcal{R}_y x\}$.*

Raffinot has shown in [14] the next theorem.

Theorem 2 ([14]) *A substring is a maximal repeat if and only if it is the longest string in an equivalence class.*

4.2 Suffix vectors and maximal repeats

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

Proposition 2 *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. □

The next two results make the link between a maximal repeat and the corresponding line in the suffix vector.

Proposition 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 2 and Prop. 2, we can affirm that only the deepest of these nodes represents a maximal repeat. □

Corollary 2 *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 Prop. 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 [14] has demonstrated Prop. 4 for the compact suffix automaton. A proof of this proposition for the suffix trees is also given in [5]. Here, we show it for the compact suffix vector.

Proposition 4 *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. □

4.3 Computing maximal repeats with suffix vector

Proposition 3 and Coro. 2 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 Figure 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 [14]. The suffix vector of `gtagtaaac$` is given in Figure 8. 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.

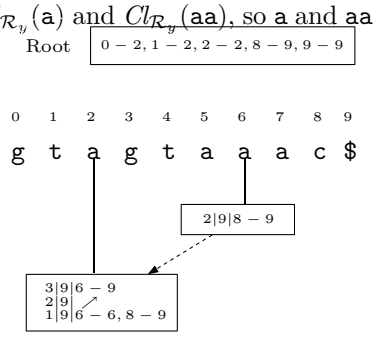

Figure 8: Suffix vector of `gtagtaaac$`, the second line of the box at position 2 has the same edge as the first line.

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

5 Conclusion and perspectives

This article presents an on-line linear algorithm for building the compact suffix vector for a string. 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 [7, 12] and LZ-indexes [2].

References

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

[2] D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In X. Deng and D.-Z. Du, editors, *Proceedings of the 16th International Symposium on Algorithms and Computation*, volume 3827 of *Lecture Notes in Computer Science*, pages 1143–1152, Sanya, Hainan, China, 2005. Springer Verlag.

[3] R. Clifford and M. Sergot. Distributed and paged suffix trees for large genetic databases. In R. A. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676, pages 70–82, Morelia, Michocán, Mexico, 2003. Springer Verlag.

[4] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science*, pages 137–143, Miami Beach, FL, 1997.

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

[6] S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, Barcelona, Spain, 2006. to appear.

[7] S. Kurtz. Reducing the space requirements of suffix trees. *Software – Practice & Experience*, 29(13), 1999.

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

[9] K. Monostori. *Efficient Computational Approach to Identifying Overlapping Documents in Large Digital Collections.* PhD thesis, Monash University, 2002.

[10] K. Monostori, A. Zaslavsky, and H. Schmidt. Suffix vector: Space-and-time-efficient alternative to suffix trees. In *CRPITS '02: Proceedings of the 25th Australasian Computer Science Conference*, volume 4, pages 157–166, Darlinghurst, Australia, 2002. Australian Computer Society, Inc.

[11] K. Monostori, A. Zaslavsky, and I. Vajk. Suffix vector: A space-efficient suffix tree representation. In P. Eades and T. Takaoka, editors, *Proceedings of the 12th International Symposium on Algorithms and Computation*, volume 2223 of *Lecture Notes in Computer Science*, pages 707–718, Christchurch, New Zealand, 2001. Springer Verlag.

[12] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. In V. Arvind and R. Ramanujam, editors, *Proceedings of the Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 186–196, Chennai, India, 1998. Springer Verlag.

[13] É. Prieur and T. Lecroq. From suffix trees to suffix vectors. In J. Holub and M. Šimanek, editors, *Proceedings of the Prague Stringology Conference '05*, pages 37–53, Prague, Czech Republic, 2005.

[14] M. Raffinot. On maximal repeats in strings. *Information Processing Letters*, 80(3):165–169, 2001.

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

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