From Suffix Trees to Suffix Vectors

Élise Prieur and Thierry Lecroq

ABISS
University of Rouen
76281 Mont-Saint-Aignan, France

e-mail: {elise.prieur,thierry.lecroq}@univ-rouen.fr

Abstract. We present a first formal setting for suffix vectors that are space economical alternative data structures to suffix trees. We give two linear algorithms for converting a suffix tree into a suffix vector and conversely. We enrich suffix vectors with formulas for counting the number of occurrences of repeated substrings. We also propose an alternative implementation for suffix vectors that should outperform the existing one.

Keywords: Suffix tree, suffix vector, repeats.

1 Introduction

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 in a suffix tree of y. Suffix vectors have been introduced by Monostori [4, 5, 6] in order to detect plagiarism. The suffix vector of the string y consists in a succession of boxes located at some positions on the string y. These boxes are equivalent to the nodes of the suffix tree of y. Monostori gave an "on-line" linear construction algorithm of an extended suffix vector and a linear algorithm to compact a vector.

We are the first to give a formal setting for suffix vectors. To do that we describe two linear algorithms to convert a suffix tree into a suffix vector and conversely. We also supply suffix vectors with counters of the number of occurrences of repeated substrings for a given length. From practical experiences, we propose an alternative physical implementation for the suffix vectors that should outperform the one proposed by Monostori. This article is organized as follows: Section 2 introduces the different notations and quickly recalls suffix trees; Section 3 introduces suffix vectors; Section 4 shows the conversion from a suffix tree to a suffix vector; Section 5 gives the conversion from a suffix vector to a suffix tree; Section 6 presents a method for counting the number of occurrences of repeated substrings in a string; Section 7 discusses the suffix vector implementation and finally Section 8 gives our conclusions and perspectives.

2 Notations

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 append to y the symbol \$ as a terminator which does not belong to A. From now on, y is a string of length $n+1$ finishing with \$.

The suffix tree $\mathcal{T}(y)$ of y is a linear size index structure that contains all the suffixes of y from the empty one to y itself. 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 letter) 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 factors $y[position..position + length - 1]$ of y. The terminator \$ ensures that no suffix of y is an internal factor of y and thus $\mathcal{T}(y)$ has exactly $n + 1$ leaves. Each internal node has at least two children, leading to at most n internal nodes and thus 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(a) presents $\mathcal{T}(\texttt{aatttatttatta\$})$.

There exist several linear time suffix tree construction algorithms [3, 7, 1] that extensively use the notion of suffix links.

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 Figure 1:

- node 7 in the tree is identified with $\texttt{atttatt}$,
- 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 $rpos(\texttt{att}, \texttt{aatttatttatta\$}) = 3$.

3 Suffix vectors

3.1 Extended suffix vectors

The suffix vector $\mathcal{V}(y)$ of y is a linear representation of the suffix tree $\mathcal{T}(y)$ consisting in 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 $\mathcal{V}(y)$.

Monostori did not give any formal definition of the suffix vectors, he only gave a linear time construction algorithm. We will now give a description of the suffix vectors.

There is a correspondence between the lines of the boxes of the suffix vector and the nodes of the suffix tree. 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 lines and 3 columns. The first column 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 $\textsc{Target}(p, y[j + 1]) = q$.

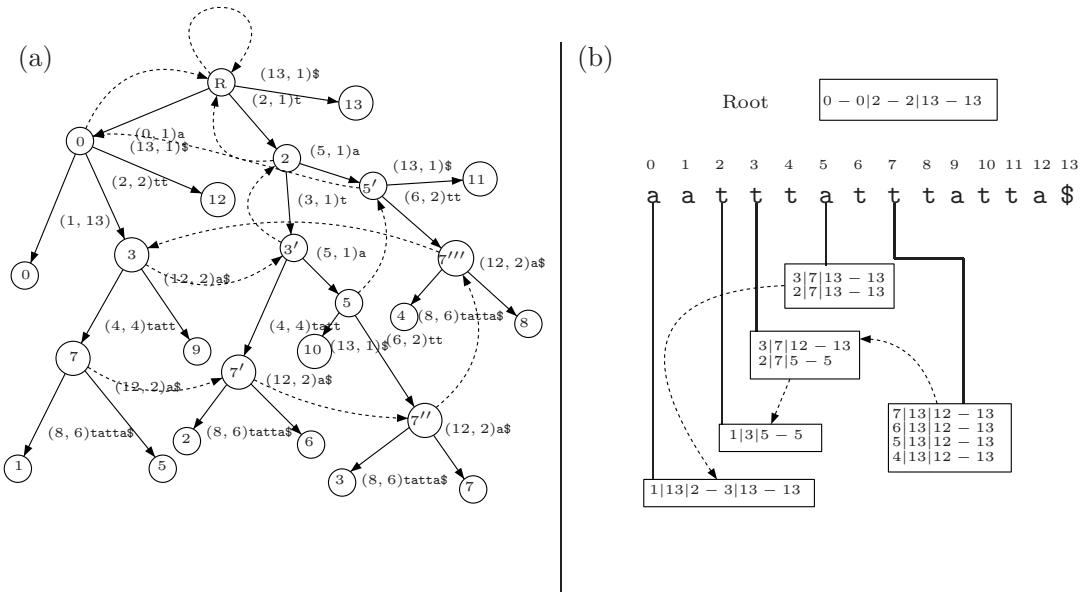

Figure 1: (a) Suffix tree of the string `aatttatttatta$`. The edges are labelled by pairs (*position*, *length*) and the substrings represented by the pairs. The label of the edge from node 0 to leaf 0 corresponds to the substring `atttatttatta$`. (b) Suffix vector of `aatttatttatta$`. Suffix links are represented by dashed arrows.

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

Inside a box, there are implicit suffix links from node represented by depth d to node represented by depth $d-1$. The depth of the deepest node is also stored in each box. Monostori pointed out in [4] that the depths in a box are continuous.

The root of the suffix tree is represented by a specific box in the suffix vector.

Example

In the box B_3 in the vector of Figure 1(b), the first line indicates that there exists a node representing a substring u of length 3 with $rpos(u, y) = 3$, so $u =$ `att`. Its natural edge is 7, this means that there is an edge from u such that $\textsc{Target}(u, y[4])$ is a node in B_7. The length of this edge is 4 (7-3), so this is the node of depth 7 in B_7 which recognizes `atttatt`.

The list of edges $B_3[1, 2]$ contains $12 - 13$. This means that there is one edge (different of the natural edge) going out from this node, its label begins at position 12 and ends at position 13. The end position is equal to the length of y, so this edge leads to a leaf.

1

We now present an example of utilization of a suffix vector. Let y be the string `aatttatttatta$` and x be the string `tatt`. We use the suffix vector of y (Figure 1(b)) to know whether x is a substring of y. In the edge list of the root, there is an edge labeled $(2, 2)$ and $y[2] =$ `t`, so we follow it and go to the box at position 2. This box

has only one line. As $y[3] \neq \mathtt{a}$, we do not follow the natural edge. The only edge in $B_2[0,2]$ begins at position 5, $y[5] = \mathtt{a}$ so we can follow it. It leads to the box B_5. As we have already read the prefix $\mathtt{ta}$ of x, we consider the line representing the node of depth 2. Since $y[6] = \mathtt{t}$, we follow the natural edge which leads to the box at position 7, so its length is 2. As $y[6..7] = \mathtt{tt}$, we have found one occurrence of x in y.

3.2 Compact suffix vectors

We introduce here the notion of 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 between the two lists. These boxes are called reduced boxes. They contain the number of nodes. To compact a suffix vector, Monostori established three rules (see [4]). These three compaction rules are:

Rule A the node with depth $d-1$ has the same number of edges as the node with depth d and these are the same edges. In this case we simply set their first edge pointers to the same position.

Rule B the node with depth $d-1$ has the same edges as the node with depth d plus some extra edges. In this case, the list of edges of the node with depth $d-1$ contains its own edges and a pointer to the list of edges of the node with depth d.

Rule C the node with depth $d-1$ has different edges to the node with depth d. In this case, all the edges must be represented in a separate list.

These rules are illustrated in Figure 2. Monostori gave a linear time algorithm for compacting an extended suffix vector.

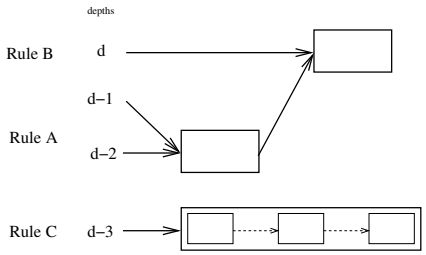

Figure 2: Representation of the compaction rules

Example

In the vector of Figure 1, we note that, in the boxes at positions 5 and 7, only the depths differ between the lines. So these boxes could be compacted storing only the first line and the number of lines. The result of the compaction of this suffix vector is shown Figure 3.

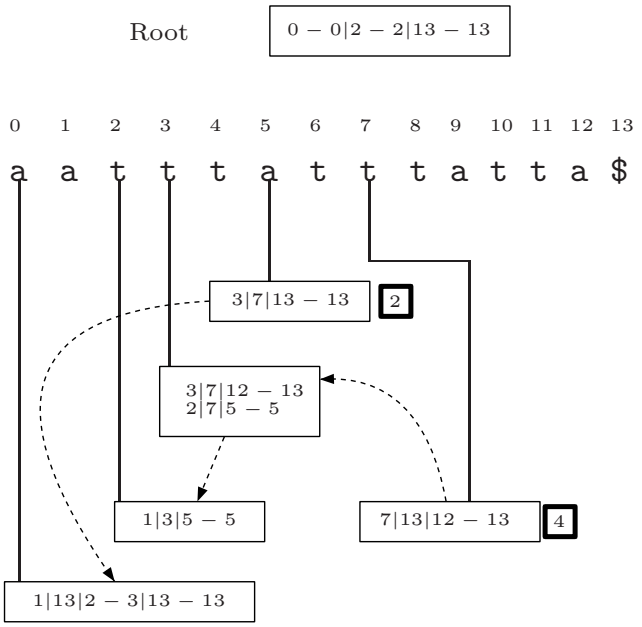

Figure 3: Compact suffix vector of the string `aatttatttatta$`.

4 Converting a suffix tree into a suffix vector

4.1 Method

We first outline the principle of the conversion of a suffix tree into an extended suffix vector by giving some propositions. The first one establishes the correspondence between an internal node in the suffix tree and a line in a box in the extended suffix vector.

Proposition 4.1. *Let p be an internal node of $\mathcal{T}(y)$ such that $y[j-d+1..j]$ is the first occurrence of the substring represented by p. This implies that there exists a box at position j in the suffix vector $\mathcal{V}(y)$ and a line h in B_j such that $B_j[h,0] = d$.*

Proof

Let $u \in \mathcal{T}(y)$ be a node of the suffix tree of the string y, u is a substring of y. When u is a node it means that it has at least two occurrences in y, this implies that u is represented in $\mathcal{V}(y)$ since all the repeated substrings of y are represented in $\mathcal{V}(y)$.

We denote by $j = rpos(u, y)$ the right position of the first occurrence of u in y. So, there exists a box B_j at position j in the vector. In this box, there exists a line h such that $B_j[h,0] = |u|$. If $u = y[j-d+1..j]$, we have $B_j[h,0] = d$.

Line h is such that among all the substrings $w \in \mathcal{T}(y)$ such that $rpos(w, y) = j$, u is the $(h+1)$-th longest one. ∎

Example

In the tree of Figure 1, node 5 can be identified with the substring $u =$ `tta` of y for which $rpos(u, y) = 5$. This node verifies Proposition 4.1 since the first line of the box B_5 in the vector represents a node of depth 3 ($B_5[0,0] = 3$ and $|$`tta`$| = 3$).

The next proposition establishes the correspondence between an edge in the suffix tree and either the natural edge or one edge in an edge list of the suffix vector.

Proposition 4.2. *Let (i, ℓ) be an edge of $\mathcal{T}(y)$ such that $\delta(p, (i, \ell)) = q$ where p and q are nodes of the suffix tree. Node p is such that $y[j-d+1..j]$ is the first occurrence of the substring represented by p. Two cases can arise:*

1. *(i, ℓ) is the natural edge of p ($y[i] = y[j+1]$), then $B_j[h, 1] = j + \ell$;*
2. *(i, ℓ) is an edge such that $y[i] \neq y[j+1]$ then there exists a pair (b, e) in $B_j[h, 2]$ such that $b = i$ and $e = i + \ell - 1$.*

Proof

Node p satisfies Proposition 4.1.

1. The natural edge:

   Considering the substring $u = y[j-d+1..j]$ the edge beginning with the letter $y[j+1]$ gives that there exists a node at position $rpos(\textsc{Target}(u, y[j+1]), y) = j + \ell$. The number $j + \ell$ is either the length of y (so the edge leads to a leaf) or the right position of the substring $u \cdot y[j+1..j+\ell]$. In the latter case, after Proposition 4.1, there exists a box at position $j + \ell$. Thus, $j + \ell$ is obtained following the natural edge of the node at line h in B_j. This implies that $B_j[h, 1] = j + \ell$.

2. The others edges:

   Considering $\textsc{Target}(u, y[i]) = q$ such that $rpos(\textsc{Target}(u, y[i]), y) = i + \ell - 1$ with $y[i] \neq y[j+1]$. The number $i + \ell - 1$ is either the length of y (so the edge leads to a leaf) or the right position of the substring $u \cdot y[i..i+\ell-1]$. In the latter case, after Proposition 4.1, there exists a box at position $i + \ell - 1$. Then, in box B_j there exists an edge $L[g] \in B_j[h, 2]$ such that $L[g].e = rpos(\textsc{Target}(u, y[i]), y) = i + \ell - 1$ and $L[g].b = L[g].e - |\textsc{Target}(u, y[i])| + 1 + |u| = i$.

■

Example

In the tree of Figure 1(a), there is an edge going out from node 5 beginning with $y[6] = \texttt{t}$ and labeled by $(6, 2)$. This node can be identified with the substring $u = \texttt{tta}$ of y for which $rpos(u, y) = 5$. It is represented by the first line of B_5. We have $rpos(\textsc{Target}(\texttt{tta}, y[6]), y) = rpos(\texttt{ttatt}, y) = 7$ so $B_5[0, 1] = 7$. This is the natural edge, this verifies Proposition 4.2 since $B_5[0, 1] = 5 + 2 = j + \ell$.

Node 5 in the suffix tree possesses only one edge beginning by a caracter in $A \setminus \{y[6]\}$ labeled by $(13, 1)$, $rpos(\textsc{Target}(\texttt{tta}, \$), y) = 13$ (this is a leaf) and $|\textsc{Target}(\texttt{tta}, \$)| - 1 - |\texttt{tta}| = 0$. The second part of Proposition 4.2 holds because in the box we have: $B_5[1, 2] = L$ such that L has one element defined by $L[0].e = 13$ and $L[0].b = 13$.

In the next proposition, the special case of the root is processed.

```
TREE2VECT(T(y))
    ▷ R is the root of the tree T(y)
 1  ADDROOT(R, V(y))
 2  for each child node p of R such that p is not a leaf do
 3  | PUSH(S, p)
 4  while not STACK-EMPTY(S) do
 5  | p ← POP(S)
 6  | ADDNODE(p, V(y))
 7  | for each child node q of p such that q is not a leaf do
 8  | | PUSH(S, q)
 9  return V(y)
```

Figure 4: Algorithm converting a suffix tree into a suffix vector.

Proposition 4.3. *Each edge (i, ℓ) going out from the root of the tree is represented by the pair $(i, i + \ell - 1)$ in the edge list of the specific box of the root of the suffix vector.*

Proof

Similar to the proof of Proposition 4.1.

■

The next two propositions show the correspondence for the suffix links.

Proposition 4.4 (Theorem 5.1 of [4]). *Let $s(u) = v$ be a suffix link in $\mathcal{T}(y)$ such that $rpos(u, y) = rpos(v, y)$ then u and v are represented in the same box of $\mathcal{V}(y)$.*

Proposition 4.5. *Let $s(u) = v$ be a suffix link in $\mathcal{T}(y)$ such that $i = rpos(u, y) \neq rpos(v, y) = j$ then $s(B_i) = B_j$.*

Proof

The suffix links are only defined from internal nodes to internal nodes. After Proposition 4.1, node u is represented in the box at position i and node v in the box at position j.

■

4.2 Algorithm

We now describe the algorithm to get a suffix vector from a suffix tree. For each node p of $\mathcal{T}(y)$, we need to know the value $rpos(p, y)$. This can be computed if each node p stores its length $|p|$ and the position of the first occurrence of p which corresponds to the number of the smallest leaf in the subtree rooted at p. This algorithm is based on a depth-first search of the suffix tree. It ensures the visit of all the nodes of the suffix tree. We use a stack S to visit the nodes (see Figure 4).

First, the algorithm processes the root because, in the vector, the root is not represented as the other nodes. The function ADDROOT, called line 1 in Figure 4, adds all the edges going out from the root of the tree in the root list of the suffix vector. It is described Figure 5.

```
AddRoot(R, V(y))
    ▷ LR is the list representing the root of V(y)
 1  LR ← ∅
 2  for each edge (i, ℓ) going out from R do
 3  ⌊ Insert((i, i + ℓ − 1), LR)
```

Figure 5: Algorithm adding the root of a suffix tree into a suffix vector.

```
AddNode(p, V(y))
 1  j ← rpos(p, y)
 2  if ∄B_j then
 3  | Create(B_j)
 4  | h ← 0
 5  else h ← k
    | ▷ k is the number of lines in B_j
 6  | k ← k + 1
 7  B_j[h, 0] ← |p|
 8  for each edge (i, ℓ) going out from p do
 9  | if y[i] = y[j + 1] then
    | | ▷ this is the natural edge
10  | | B_j[h, 1] ← j + ℓ
11  | else Insert((i, i + ℓ − 1), B_j[h, 2])
12  if j ≠ rpos(s(p), y) then
13  ⌊ s(B_j) ← B_{rpos(s(p),y)}
```

Figure 6: Algorithm adding a node of a suffix tree into a suffix vector.

Then, for each node p of the tree, we add its equivalent in the vector: we insert a line in a box at position $rpos(p, y)$ in the vector and if the box does not exist, we create it with the correct line. This function is detailed in Figure 6.

Theorem 4.1. *The algorithm* TREE2VECT($\mathcal{T}(y)$) *correctly computes* $\mathcal{V}(y)$ *in time* $O(|y|)$

Proof

The correctness of the algorithm comes from Propositions 4.1 to 4.5.

Each node and each edge of the suffix tree are processed only once. The operations per node and per edge take a constant time. Since the number of edges and nodes of the suffix tree is linear, the result on the running time follows.

■

5 Converting a suffix vector into a suffix tree

5.1 Method

We now show the conversion from an extended suffix vector to a suffix tree. The next proposition deals with the internal nodes.

Proposition 5.1. *Each line h of a box B_j in the suffix vector of y can be associated to an internal node of the suffix tree of y.*

Proof

Let u be the substring of y such that $u = y[j - B_j[h, 0] + 1..j]$. If there is a line h in a box B_j it means that $u \cdot y[j + 1..B_j[h, 1]]$ and $u \cdot y[L[0].e..L[0].b]$ are factors of y with $L[0] \in B_j[h, 2]$ and $y[j + 1] \neq y[L[0].e]$. This means that u has two occurrences in y followed by two different letters which implies that u represents an internal node in $\mathcal{T}(y)$. ■

Example

In the box B_5 of Figure 1(b), $B_5[0, 0] = 3$ indicates that the substring $u = y[5 - 3 + 1..5] = y[3..5]$ is represented in the first line of this box. This string is `tta`, it corresponds to node 5 in the suffix tree of y.

The three following propositions deal with the edges.

Proposition 5.2. *Each value $B_j[h, 1]$ of a line h of a box B_j in the suffix vector of y can be associated to an edge of the suffix tree of y.*

Proof

Let u be the substring of y such that $u = y[j - B_j[h, 0] + 1..j]$. There exists an edge in the tree such that $\delta(u, (j+1, B_j[h, 1]-j)) = y[j-B_j[h, 0]+1..B_j[h, 1]]$. Thus $y[j - B_j[h, 0] + 1..B_j[h, 1]]$ is in $\mathcal{T}(y)$, it can be an internal node or a leaf. ■

Example

In the box B_5 of Figure 1(b), the second column of the first line means that we can go to position 7 following an edge starting from position 6, this edge is $\delta(\texttt{tta}, (6, 2))$ in $\mathcal{T}(y)$.

Proposition 5.3. *Each pair (b, e) in a edge list of a line h of a box B_j in the suffix vector of y can be associated to an edge of the suffix tree of y.*

Proof

Let u be the substring of y such that $u = y[j - B_j[h, 0] + 1..j]$. There exists an edge in the tree such that $\delta(u, (b, e - b + 1)) = u \cdot y[e..b]$. Thus $u \cdot y[e..b]$ is in $\mathcal{T}(y)$, it can be an internal node or a leaf. ∎

Example

The third column of the first line of B_5 of Figure 1(b) has only one edge, $L[0].b = 13$ and $L[0].e = 13$ ($L[0].e = |y|$ means that this edge leads to a leaf). We have to verify that there exists an edge such that $\delta(\texttt{tta}, (L[0].b, L[0].e - L[0].b + 1)) = \delta(\texttt{tta}, (13, 1))$ in the tree. The node 5, which recognizes the same substring as the first line of B_5, has an edge labeled $(13, 1)$ going out to a leaf. We showed the equivalence between the node 5 in the tree and the first line of B_5 in the vector.

Proposition 5.4. *Each pair (b, e) in a edge list of the root the suffix vector of y can be associated to an edge of the suffix tree of y.*

Proof

Similar to Proposition 5.3. ∎

The next proposition deals with the leaves.

Proposition 5.5. *The leaves of the suffix tree $\mathcal{T}(y)$ can be retrieved from the suffix vector $\mathcal{V}(y)$.*

Proof

This is a direct consequence of Propositions 5.3 to 5.5 and the fact that there is exactly one edge leading to each leaf. ∎

The two following propositions deal with the suffix links.

Proposition 5.6 (Theorem 5.1 of [4]). *In a box B_j of k lines the suffix link of the node represented by the line h points to the node represented by the line $h + 1$ for $0 \leqslant h < k - 1$.*

Proposition 5.7. *In a box B_j of k lines the suffix link of the node represented by the line $k - 1$ points to $s(B_j)$.*

Proof

By construction. ∎

5.2 Algorithm

We give in this section an algorithm that computes a suffix tree from an extended suffix vector for a string y. It first processes the root box of the suffix vector and then processes sequentially each remaining box of the vector. For each box it sequentially processes each lines (see Figure 7).

```
VECT2TREE(V(y))
 1  R ← new node
 2  for each (b, e) in the edge list of the root box of V(y) do
 3  |  p ← new node at depth e − b + 1 at position e
 4  |  δ(R, (b, e − b + 1)) ← p
 5  for j ← 0 to n do
    |  ▷ k is the number of lines of the box B_j
    |  ▷ p is the node previously created at depth B_j[k − 1, 0] at position j
 6  |  q ← new node at depth B_j[k − 1, 0] + B_j[k − 1, 1] − j + 1 at position B_j[k − 1, 1]
 7  |  δ(p, (j + 1, B_j[h, 1] − j + 1)) ← q
 8  |  r ← p
 9  |  for each pair (b, e) ∈ B_j[k − 1, 2] do
10  |  |  q ← new node at depth B_j[k − 1, 0] + e − b + 1 at position e
11  |  |  δ(p, (b, e − b + 1)) ← q
12  |  s(p) ← s(B_j)
13  |  for h ← k − 2 to 0 do
    |  |  ▷ p is the node previously created at depth B_j[h, 0] at position j
14  |  |  q ← new node at depth B_j[h, 0] + B_j[h, 1] − j + 1 at position B_j[h, 1]
15  |  |  δ(p, (j + 1, B_j[h, 1] − j + 1)) ← q
16  |  |  for each pair (b, e) ∈ B_j[h, 2] do
17  |  |  |  q ← new node at depth B_j[h, 0] + e − b + 1 at position e
18  |  |  |  δ(p, (b, e − b + 1)) ← q
19  |  |  s(r) ← p
20  |  |  r ← p
21  return T(y)
```

Figure 7: Algorithm converting a suffix vector into a suffix tree.

Theorem 5.1. *The algorithm* VECT2TREE($\mathcal{V}(y)$) *correctly computes* $\mathcal{T}(y)$ *in time* $O(|y|)$.

Proof

The correctness of the algorithm comes from Proposition 5.1 to 5.6.

The algorithm processes each pairs of each lines of each boxes of the suffix vector which correspond to the edges and the nodes of the suffix tree whose quantity is linear. The only difficulty consists in retrieving a node at depth d for position j. This can be realized by storing the largest depth in each box. All the other operations take constant time. The result on the running time follows. ∎

6 Repeats

Before counting the number of repeats of the substrings of y, we explain some notions for the counting of the number of occurrences.

6.1 Counting the number of occurrences

As mentioned before, each line of a box of $\mathcal{V}(y)$ is associated to a node u in $\mathcal{T}(y)$ and thus to a substring of y. Let B_j be a box of $\mathcal{V}(y)$, let h be a line of B_j, the line h is

associated to the substring $u = y[j - B_j[h,0] + 1..j]$. Let $nbOcc(u)$ be the number of occurrences of the substring u. Let $nbL(t)$ be the number of leaves in the subtree rooted at the end of any edge t .

Then

$$nbL(t) = \begin{cases} 1 & \text{if } t = |y|; \\ nbOcc(v) & \text{otherwise} \end{cases}$$

where v is the node in the box B_t such that t is the end position of the edge going to node v.

We then deduce that

$$nbOcc(u) = nbL(ne) + \sum_{L[g] \in B_j[h,2]} nbL(L[g].e).$$

With this expression, it is easy to obtain a linear algorithm which adds the value $nbOcc(u)$ on each line of the vector. This algorithm visits the boxes of the suffix vector from right to left and completes the lines with $nbOcc(u)$.

6.2 Counting the number of repeats

The method described in this section allows to compute for each substring of y with a given length $lg < n$, its number of occurrences in y. Let $lpocc(lg)$ be the list of pairs $(rpos(u,y), nbOcc(u))$ for all substrings u of y of length lg.

The principle is to visit all the boxes of the suffix vector and for each line h in a box at position j such that $B_j[h,0] \geqslant lg$ to update $lpocc(lg)$ using $nbOcc$(u) where u is the substring represented by this line.

First, we test if the depth of the deepest node of the box we are visiting is larger than lg. In the contrary case, the visit of the box stops. For reduced boxes we only have to take into account the deepest node, whereas in the other boxes we have to process all the nodes whose depth is larger than lg. We now explain the two different cases.

Reduced box Let us assume that we are processing the reduced box B_j and $B_j[0,0] = d \geqslant lg$. This implies this line represents $u = y[j - d + 1..j]$. Let j' be a position such that $j - d + 1 \leqslant j' \leqslant d - k + 1$ and $|y[j'..j]| \geqslant lg$ (k is the number of lines represented in the box). For each possible j', let v be $y[j'..j' + lg - 1]$, either we add the pair $(rpos(v,y), nbOcc(u))$ in the list or we update $nbOcc(v)$ with $nbOcc(u)$ if v is already present in $lpocc$.

Extended box For each line h of the extended box B_j such that $B_j[h,0] = d \geqslant lg$. This implies this line represents $u = y[j - d + 1..j]$. Let v be the prefix of length lg of u, either we add the pair $(rpos(v,y), nbOcc(u))$ in the list or we update $nbOcc(v)$ with $nbOcc(u)$ if v is already present in $lpocc$.

After that, the list $lpocc(lg)$ gives the number of occurrences of repeated substring of y of length lg.

7 Implementation

7.1 Monostori's implementation

We now explain the representation used by Monostori to store compact suffix vectors (section 5.4 of [4]). Each box contains the following information:

Deepest node The deepest node value is usually small so Monostori proposed to store it in 1 or 4 bytes. The first bit is used to denote the number of bytes needed to store the value, so the deepest node value is represented with 7 or 31 bits.

Number of nodes In a box, we also need the number of nodes value which is smaller than the deepest node value. The number of nodes can fit into one byte when the deepest node value is stored into one byte. So, it is not necessary to use another bit to flag as for the depth.

Suffix link The next information stored in a box is the suffix link. If the number of nodes is equal to the depth of the deepest node, this means that the smallest depth in the box is 1. So the suffix link of the box is implicit to the root. In this representation, the suffix link is stored anyway because its first bit is used to indicate if the box is a reduced one and its second one is used to indicate if the values of the natural edges will need 1 or 4 bytes.

Natural edges Then the natural edges are stored in an array called array of next node pointers. We can save space by storing the length of an edge rather than the end position. If there is one of the lengths of the natural edges of the box which need to be stored in more than one byte, all of them are stored in 4 bytes.

Edges At last, we have to consider the representation of the edges. An edge is represented with its start position and its length. The first edge pointer of a node gives the memory address of the list of edges going out from this node. The first bit of a start position of an edge indicates if this edge leads to a leaf. In this case, the length of the edge is not stored. The next bit flags whether this is the last edge of the list. The third one is used to indicate the number of bytes (1 or 4) required to store the length of the edge.

7.2 Counting

Table 1 compares the space required by the suffix vector with the space required by the suffix tree implemented with Kurtz's method [2]. It is extracted from Table 5.1 in [4]. Here, we give the results for four files which are two English texts (`book2` and `bible`), one C program (`progc`) and one DNA sequence (`ecoli`). The results are given in bytes per symbol of the input sequences.

The measures done by Monostori show that its implementation of the suffix vectors is less efficient for DNA sequences than for large alphabets. Therefore we performed experiments on several DNA sequences. Tables 2 to 6 give the results for five of them:

- chromosome 4 of *S. cerevisiae* (of length 1,531,931);

Table 1: Comparison of space requirements of suffix vectors and suffix trees.

File name	File size (in bytes)	Bytes/symbol Compact Suffix Vector	Bytes/symbol Kurtz
book2	610,857	8.61	9.67
bible	4,047,393	8.53	7.27
progc	39,612	8.63	9.59
ecoli	4,638,691	12.51	12.56

- chromosome 3 of *E. coli* (of length 13,783,270);
- chromosome 5 of *E. coli* (of length 20,922,241);
- chromosome 2 of *A. thaliana* (of length 19,847,294);
- chromosome 4 of *A. thaliana* (of length 17,790,892).

For each sequence, we build its extended suffix vector and reported for each box:

- the number of nodes;
- the depth;
- the length of the natural edge minus 1 (since it is always at least equal to 1);

and for each edge list of each box:

- the next position;
- the difference between the next position and the position of the box minus 2 (since it is always at least equal to 2);
- the length of the edge minus 1 (since it is always at least equal to 1).

For all the values we counted the number of them that can fit between:

- 1 and 6 bits;
- 7 and 14 bits;
- 15 and 22 bits;
- 23 and 30 bits.

The idea is, instead of using only one flag bit and use 1 or 4 bytes for representing the different objects, to use two flag bits and 1, 2, 3 or 4 bytes for representing them. The tables clearly show that this approach will save a large number of bytes in all cases. Of course, storing the difference between the next position and the position of the box rather than the next position always enables to save storage space. The actual total gain is not yet completely measurable since, to keep a direct access to any node in a box, all the natural edges in a box are stored with the space necessary for the largest natural edge. We can now present an alternative implementation.

Table 2: Counts for chromosome 4 of *S. cerevisiae*. It contains 1,531,931 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	501,378	129		
Depth	875,852	13,924		
Natural edge	369,501	6497	513,778	
Next position	55	18,794	1,555,245	
Difference	2539	128,375	1,443,180	
Edge length	642,254	9143	922,697	

Table 3: Counts for chromosome 3 of *E. coli*. It contains 13,783,270 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	4,182,237	2283		
Depth	7,978,520	172,622		
Natural edge	3,697,663	32,220		4,421,259
Next position	2	18,527	4,529,723	9,302,660
Difference	36,315	439,978	11,548,079	1,826,540
Edge length	5,633,779	35,474		8,181,659

Table 4: Counts for chromosome 5 of *E. coli*. It contains 20,922,241 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	6,395,182	3224	4	
Depth	11,998,929	288,722	40,428	
Natural edge	5,456,836	121,698	5156	6,744,389
Next position	8	18,606	4,579,468	16,428,416
Difference	42,261	485,515	14,355,158	6,143,564
Edge length	8,583,613	111,606	2344	12,328,935

Table 5: Counts for chromosome 2 of *A. thaliana*. It contains 19,847,294 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	6,429,030	2192		
Depth	11,499,363	137,572	183,616	
Natural edge	5,353,725	32,671		6,434,155
Next position	61	18,470	4,630,471	15,624,486
Difference	37,770	426,015	14,124,810	5,691,893
Edge length	8,186,302	23,318		12,063,868

Table 6: Counts for chromosome 4 of *A. thaliana*. It contains 17,790,892 base pairs.

	1 – 6 bits	7 – 14 bits	15 – 22 bits	23 – 30 bits
Number of nodes	5,809,708	1869		
Depth	10,203,150	136,807	224,650	
Natural edge	4,767,148	33,801		5,763,658
Next position	61	18,713	4,578,990	13,529,684
Difference	26,541	474,155	13,353,092	4,273,660
Edge length	7,325,278	33,235		10,768,935

7.3 An alternative implementation

Here, we explain how to use the idea explained in section 7.2 to reduce the space. Each box contains the following information:

Deepest node Instead of storing in 1 or 4 bytes, we could store the depth of the deepest node in 1, 2, 3 or 4 bytes. This means that we have to use the two first bits to indicate how many bytes we need. So the deepest node value is stored in 6, 14, 22 or 30 bits.

Number of nodes As mentioned in Section 7.1, the number of bytes needed to store the number of nodes depends on the number of bytes of the deepest node value. Then, if we need 6, 14, 22 or 30 bits to store this depth, we could use respectively 1, 2, 3 or 4 bytes to store the number of nodes.

Suffix link Similar to Section 7.1.

Natural edges We can use two flag bits and then 1, 2, 3, or 4 bytes for all the values. We store the length of the natural edge minus 1 since it is always larger than 1.

Edges Instead of storing the start position of an edge, we could store the difference between the start position and the position of the box. For a box at position j and an edge starting at $L[g].b > j$, we store $L[g].b - j - 2$. We can use the same idea as for the deepest node value to store $L[g].b - j - 2$ and the length of $L[g] - 1$ using 1, 2, 3 or 4 bytes.

The main idea is to reduced the space required with Monstori's implementation for DNA sequences by storing the data with 2 or 3 bytes instead of 4 when it is possible. To do that we use 2 bits for the needed number of bytes. Tables 2 to 6 show that we can reduce the space in many cases.

8 Conclusions and perspectives

We presented a first formal setting for suffix vectors that are space economical alternative data structures to suffix trees. We gave two linear algorithms for converting a suffix tree into a suffix vector and conversely. We enriched suffix vectors with formulas for counting the number of occurrences of repeated substrings. We finally proposed an alternative implementation for suffix vectors that should outperform the one proposed by Monostori specially for small alphabets and large sequences.

In order to really take advantage of this implementation we are studying an "online" linear algorithm for directly building a compact suffix vector. This should allow to deal efficiently with huge sequences such as human chromosomes.

References

[1] 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.

[2] S. Kurtz. Reducing the space requirements of suffix trees. *Software Practice & Experience*, 29(13):1149–1171, 1999.

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

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

[5] 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, Melbourne, 2002. Australian Computer Society, Inc.

[6] K. Monostori, A. Zaslavsky, and I. Vajk. Suffix vector: A space-efficient suffix tree representation. In *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.

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