Pattern Matching on Sparse Suffix Trees

Roman Kolpakov, Gregory Kucherov, Tatiana Starikovskaya

Pattern matching on sparse suffix trees

Roman Kolpakov
Moscow State University
Moscow, Russia
foroman@mail.ru

Gregory Kucherov
Laboratoire d'Informatique Gaspard Monge
Université Paris-Est & CNRS
Marne-la-Vallée, France
Gregory.Kucherov@univ-mlv.fr

Tatiana Starikovskaya
Moscow State University
Moscow, Russia
tat.starikovskaya@gmail.com

***Abstract*—We consider a compact text index based on evenly spaced sparse suffix trees of a text [9]. Such a tree is defined by partitioning the text into blocks of equal size and constructing the suffix tree only for those suffixes that start at block boundaries. We propose a new pattern matching algorithm on this structure. The algorithm is based on a notion of suffix links different from that of [9] and on the packing of several letters into one computer word.**

I. Introduction

Many application areas, such as genomics or computer security for example, face a sharp growth of volumes of available data. Even with the spectacular development of hardware capacities, data size often remains a bottleneck for its efficient processing, which requires new algorithmic solutions allowing for both a compact representation and efficient querying of data.

Suffix trees remain a very popular and commonly used data structure for text indexing, that are known, however, to be rather space-consuming in practice. A way to save space, that we study in this paper, is to partition the text T into blocks of r characters and to use a suffix tree which stores only those suffixes that start at the block boundaries. Then the suffix tree has no more than n/r leaves and then no more than n/r internal nodes, where n is the length of T. Such a suffix tree, called *an evenly spaced sparse suffix tree* (hereafter *sparse suffix tree* for short), has been first introduced in [9]. Figure 1 provides an example of a sparse suffix tree. The definition of sparse suffix tree has been generalized to an arbitrary text partition into *words* [1], and a corresponding notion of suffix arrays on words has also been studied [4].

A sparse suffix tree allows one to easily search for the occurrences of a pattern P that start at block boundaries. However, efficiently identifying occurrences of P starting inside blocks is a more complicated task. The first solution to it has been proposed in the original paper [9], which takes time $O(rn)$ in the worst case to compute all occurrences of P in T.

Recently, the idea of sparse suffix tree has been used in [3] to build a *succinct index* of a text. Indeed, if r is of order $\log_\sigma n$, where σ is the alphabet size, then the sparse suffix tree takes space $O(n/\log_\sigma n)$ which is the minimal space needed to store the text itself. On the other hand, if $r < \log_\sigma n$ then it is possible to compress T into a string T' by packing each block into a machine word, and to use a regular suffix tree for T' instead of a sparse suffix tree for T.

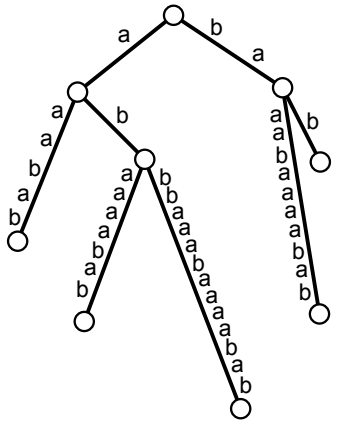

Figure 1. A sparse suffix tree for string $T = abb\ baa\ aba\ aaa\ bab$ and $r = 3$.

Based on these ideas, the algorithm of [3] computes all occurrences of a pattern in a text in time $O(m + (\log n)(\log_\sigma n) + occ \log n)$, where m is the pattern length and occ the number of occurrences of the pattern in the text. The idea of using a suffix tree for a compressed string has been further developed in [6] to provide entropy-efficient indexes.

In this paper, we turn to the original approach based on the sparse suffix tree for the input text T. To make an efficient pattern matching possible, we augment the tree with suffix links defined differently from those of [9]. Our algorithm computes all occurrences of a pattern of length $m \geq r$ in a text in $O(m \cdot \max\{1, \frac{r \log \sigma}{w}\} + \max\{occ, r\} \cdot \log n / \log\log n))$ time and $O(\frac{n}{r})$ space. Similar to [3], [6] we assume a unit-cost RAM model and take advantage of unit-cost comparisons of blocks of letters to speed up the algorithm.

Using sparse suffix trees over the alphabet of letters (as opposed to the meta-alphabet of blocks of letters stored in

a computer word) has several advantages. Our construction works for any r and we avoid the use of perfect hashing in navigating over the suffix tree (under the assumption of constant-size alphabet). Furthermore, time and space bounds of [3] and [6] (for the index for internal memory model) can be obtained from our results by an appropriate choice of the block size r without any additional compressed text index data structure. Finally, our suffix links allow to locate all pattern suffixes $P[k+1..m]$, $k = 0..r-1$, occurring at block boundaries in a single traversal of the suffix tree, rather than locating them independently (as it is done in the previous works), which we believe to be more elegant.

The paper is organized as follows. We first define sparse suffix trees and associated suffix links, then explain how to compute occurrences of a pattern using this data structure and finally describe how this data structure can be constructed.

II. Evenly spaced sparse suffix tree

Let Σ denote an alphabet, i.e. a set of *letters* or *characters*, of cardinality σ. We assume a lexicographic order $<$ on Σ, naturally extended to the set of all strings over Σ. Positions in strings are numbered from 1. For a string α, $\alpha[i..j]$ denotes substring $\alpha[i]\dots\alpha[j]$, and $\alpha[j..]$ is a shorthand for the suffix $\alpha[j..|\alpha|]$ respectively.

We consider *evenly spaced sparse suffix trees* as defined in [9]. Consider a string $T[1..n]$. Let Suf_r be the set of suffixes $\{T[rj+1..]|j=0,1,\dots,\frac{n}{r}-1\}$ (assume for simplicity that n is a multiple of r).

An *r-spaced suffix tree* of T, denoted ST_r, is a compacted trie for the set Suf_r. For $r=1$, the r-spaced suffix tree is the usual suffix tree. Edges of an *r-spaced suffix tree* are labelled by substrings $T[i..j]$ of T, represented by a pair (i,j). We define *explicit* and *implicit* nodes of ST_r in the same way as for the regular suffix trees. An implicit node will be specified by a pair (v,ℓ), where v is the closest explicit ancestor node and ℓ is the offset with reference to v. Note that by definition of the tree, the labels of the outgoing edges of any explicit node have different first letters.

Assuming that the last letter of T is unique, ST_r has $\frac{n}{r}$ leaves and then no more than $\frac{n}{r}$ explicit internal nodes. Therefore, ST_r takes $O(\frac{n}{r})$ space.

By default, a *node* may refer to either an explicit or an implicit node. A string α is *represented* in ST_r if α is a prefix of one of the suffixes of Suf_r, i.e. if α is a substring of T starting at a position $rj+1$ for some j. In this case, α is the label of some node v of ST_r, and we say that α *is represented by* v, and $|\alpha|$ is the *string depth* of v.

Consider the lexicographic order on suffixes Suf_r. Note that each leaf of the tree ST_r represents some suffix of Suf_r, and we call the *rank* of a leaf v the rank of the suffix represented by v in the lexicographic order on Suf_r.

For a node v, we define $MinRank(v)$ and $MaxRank(v)$ to be respectively the minimal and the maximal rank of leaves in a subtree of ST_r rooted at v. The ranks of all leaves of the subtree rooted at v form the *rank interval* $[MinRank(v), MaxRank(v)]$. If α is a word corresponding to v, then the ranks of suffixes of Suf_r starting with α are specified by the interval $[MinRank(v), MaxRank(v)]$.

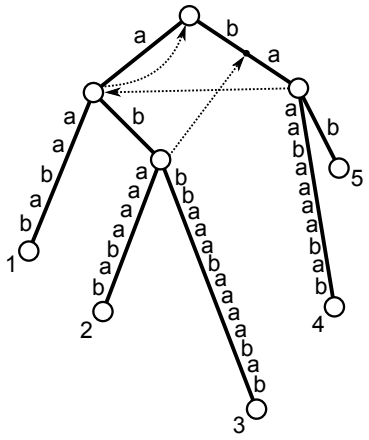

Figure 2. A sparse suffix tree for string $T = abb\ baa\ aba\ aaa\ bab$ and $r = 3$ with suffix links and ranks of leaves shown.

We assume that for each explicit node v of ST_r, $MinRank(v)$ and $MaxRank(v)$, as well as its string depth $d(v)$ can be recovered in constant time. This can be trivially achieved by post-processing the tree and storing this information explicitly.

We extend the r-spaced suffix tree ST_r with *suffix links*: for each explicit node v representing a string α, a suffix link $s(v)$ maps v to a (not necessarily explicit) node labelled with the longest proper suffix $\alpha[i+1..]$ represented in the tree (see Fig. 2). Offset i will be called the *type* of the suffix link. It follows easily that $1 \le i \le r$. Note that this definition of suffix links is different from that of [9] which sets $s(v)$ to be the node, always explicit, representing the suffix $\alpha[r+1..]$.

For each explicit node v of ST_r, we store the target node $s(v)$ together with the type of the suffix link.

III. Pattern matching algorithm

Consider a pattern $P[1..m]$, where $m \ge r$. To find the occurrences of P in T we use the following general idea. Based on the sparse suffix tree, we first locate all occurrences of the pattern suffixes $P[1..], P[2..], \dots, P[r..]$ starting at block boundaries with the procedure RightSearch that we describe in Section III-A. Secondly, we locate all occurrences of $P[1..k]$, for $k = 1..r-1$, ending at block boundaries using another procedure LeftSearch. Finally, procedure Selection computes those boundaries that are both preceded by $P[1..k]$ and followed by $P[k+1..]$ for the same k, and thus correspond to occurrences of the entire pattern. Procedure Selection is essentially the same as in [6], but we still provide its description in Section III-B for the sake of completeness.

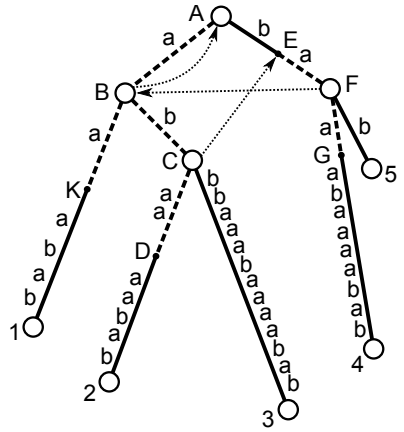

Figure 3. Right search for pattern $P = abaa$ in string $T = abb\ baa\ aba\ aaa\ bab$, $r = 3$. During the procedure the algorithm goes along the path is $ABCDCEFGFBK$ (the path is shown by dashes)

We use bit-vector operations to speed up the algorithms and we assume that both strings P and T are given in a packed form, namely that they are divided into blocks of $t = \frac{w}{\log \sigma}$ characters, where w is the size of a machine word, and each block is packed into one machine word.

A. RIGHTSEARCH

The procedure RIGHTSEARCH (Algorithm 1) proceeds by navigating through ST_r trying to locate all nodes representing $P[1..], P[2..], \ldots, P[r..]$. Starting at the root with $P[1..]$, RIGHTSEARCH follows down the current suffix $P[k+1..]$ in the tree as long as possible.

When following an edge in the tree, its label $T[i..j]$ is divided into blocks of t letters, except for possibly a smaller last block, and each block is compared by a single operation.

Assume that RIGHTSEARCH arrives at some (generally implicit) node (v, ℓ) reaching the end of $P[k+1..m]$ (line 14 of Algorithm 1). Then the algorithm retrieves the rank interval $[MinRank(v'), MaxRank(v')]$, where v' is the closest explicit descendant node, which specifies all the occurrences of $P[k+1..]$ at block boundaries. After that, the traversal jumps to $s(v)$ and proceeds with the prefix $P[k+i+1, m-\ell+1]$ of the current suffix $P[k+i+1..]$, where i is the type of suffix link $s(v)$ (lines 20-22).

Assume now that RIGHTSEARCH reaches a mismatch while processing current suffix $P[k+1..]$ (line 8). Assume that the mismatch occurred when visiting a node (v, ℓ) and processing a prefix $P[k+1..p]$ of $P[k+1..]$. Similarly to the previous case, the algorithm jumps to $s(v)$ and proceeds with the prefix $P[k+i+1, p-\ell+1]$ of the new current suffix $P[k+i+1..]$, where i is the type of suffix link $s(v)$. The procedure is illustrated on Fig. 3.

Importantly, the described procedure does not miss any occurrences:

Algorithm 1 RIGHTSEARCH

```
1:  k ← 1
2:  p ← 1
3:  Node ← root
4:  NodeOffset ← 0
5:  while k ≤ r do
6:    while p ≤ m do
7:      starting from position p in P, follow down the
        current edge of ST_r by comparing blocks of up
        to t characters at once
8:      if mismatch occurred then
9:        break the while-loop
10:     else
11:       update Node, NodeOffset, p
12:     end if
13:   end while
14:   if p = m then
15:     if NodeOffset ≠ 0 then
16:       Descendant ← closest explicit descendant for
          (Node, NodeOffset)
17:     end if
18:     output k,MinRank(Descendant),
        MaxRank(Descendant)
19:   end if
20:   p ← p − NodeOffset + 1
21:   (Node, NodeOffset) ← s(Node)
22:   k ← k + type of the suffix link (Node, s(Node))
23: end while
```

Lemma 1. RIGHTSEARCH *correctly identifies all suffixes* $P[k+1..]$, $0 \le k \le r-1$, *occurring at block boundaries of* T.

Proof: It is easy to see by induction that once a suffix $P[k+1..]$ is found (line 11 of Algorithm 1), it is represented in the tree and therefore occurs starting at a block boundary.

A key point is that the procedure does not miss any such suffixes. This is due to the definition of suffix links: when following a suffix link (lines 20-22), the algorithm switches from processing the suffix $P[k+1..]$ to the suffix $P[k+i+1..]$, where i is the type of the suffix link. It follows that no suffix $P[k+i'+1..]$ for $i' < i$ can be represented in the tree. This is because the suffix link points to the *longest* suffix represented in the tree. ■

Let us now turn to the analysis of the running time of RIGHTSEARCH. The algorithm navigates over the suffix tree ST_r by following edges downwards. We analyse separately the traversal of two types of edges: completely traversed edges (hereafter *traversed edges*), and incompletely traversed edges (hereafter *dead-end edges*), either due to a mismatch or due to a found suffix.

The number of dead-end edges is at most r, as each of

them terminates the processing of some suffix $P[k+1..]$. On each such edge, the algorithms makes no more that m/t block comparisons. Therefore, the whole time spent on dead-end edges is $O(\frac{mr}{t})$.

The number of all comparisons made along the traversed edges is bounded by m, as these comparisons compare different portions of the pattern. In other words, the sequence of these comparisons can be associated with moving a pointer in the pattern left-to-right by blocks of letters. The whole time spent on these comparisons is thus $O(m)$.

Theorem 1. RIGHTSEARCH *computes the rank intervals of all suffixes* $P[k+1..]$, $0 \leq k \leq r-1$, *occurring at block boundaries of* T *in time* $O(m \cdot \max\{1, \frac{r}{t}\}) = O(m \cdot \max\{1, \frac{r \log \sigma}{w}\})$.

B. LEFTSEARCH *and* SELECTION

LEFTSEARCH locates all occurrences of $P[1..k]$ ending at block boundaries of T. Let $RevBlocks_r$ be a set of the *reversed* blocks $T[r(j-1)+1..rj]$, for $j = 1, \ldots, \frac{n}{r}$. We build a compacted trie for $RevBlocks_r$. Each leaf v of the trie is associated with the indices of the blocks it represents, namely with the set $\{j | 1 \leq j \leq \frac{n}{r} - 1$ and v represents the reversal of $T[r(j-1)+1..rj]\}$.

We augment the trie with suffix links and rank intervals defined in the same way as in RIGHTSEARCH. Then we run the procedure described in RIGHTSEARCH, but we use the trie instead of the sparse suffix tree and a pattern $P[r-1]P[r-2]\ldots P[1]$.

So, for each $k = 1, \ldots, r-1$, LEFTSEARCH locates the closest explicit descendant of the node representing $P[k]P[k-1]\ldots P[1]$ and retrieves the rank interval I^k_{left} corresponding to that node. Obviously, I^k_{left} contains starting positions of reversed blocks starting with $P[k]P[k-1]\ldots P[1]$. All the intervals I^k_{left}, for $k = 1, \ldots, r-1$, can be computed in $O(r \max\{1, \frac{r \log \sigma}{w}\})$ time (time analysis is exactly the same as in RIGHTSEARCH).

From Section IV it is obvious that the trie for $RevBlocks_r$ and suffix links for it can be constructed in $O(nr)$ time.

We now show how SELECTION can be reduced to 2D range reporting problem. Consider a rank interval I^k_{right} output by RIGHTSEARCH for some k (line 18 of Algorithm 1). We have to compute all $j, 1 \leq j \leq \frac{n}{r}$, such that the rank of $T[rj+1..]$ in the lexicographical order on Suf_r belongs to I^k_{right} and the rank of the reversal of $T[r(j-1)+1..rj]$ in lexicographical order on $RevBlocks_r$ belongs to I^k_{left}. Each such j will correspond to an occurrence of $P[1..m]$ starting at position $rj-k+1$ in T.

Consider a set Q of $\frac{n}{r}$ points, where a point j has the first coordinate equal to the rank of $T[jr+1..]$ in the lexicographical order on Suf_r, and the second coordinate equal to the rank of the reversal of $T[r(j-1)+1..rj]$ in lexicographical order on $RevBlocks_r$. It can be easily seen that Q can be constructed in linear time. Then the desired output is the points of Q which lie inside the rectangle $I^k_{left} \times I^k_{right}$ (see Fig. 4).

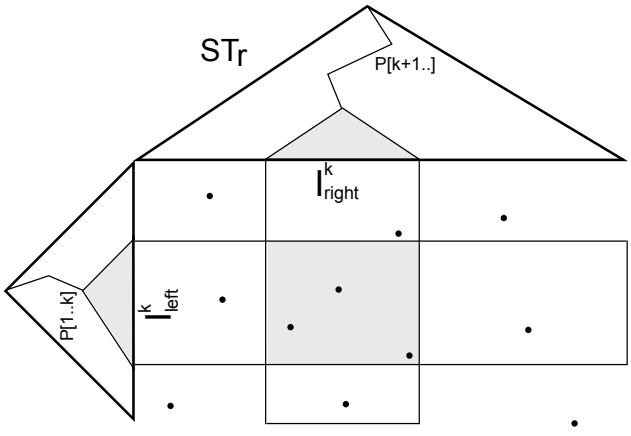

Figure 4. Selection on Q.

The problem of selecting points lying inside a given rectangle within some larger set of points has been studied in the literature under the name of 2D range reporting problem. We can use several linear-space solutions to this problem that have different trade-offs between preprocessing and query time. Let occ_k be the number of points inside the rectangle $I^k_{left} \times I^k_{right}$ (or, alternatively, the number of occurrences of P in T crossing a block border at position $k+1$).

The most recent and efficient linear 2D range reporting data structure [2] has $O(\max\{occ_k, 1\} \log^{\epsilon} n)$ query time, where ϵ is an arbitrary positive constant, however, no bound for construction time of the data structure has been given in the paper. The data structure of [11] has $O(\log n + occ_k \log^{\epsilon} n)$ query time and $O(\frac{n}{r} \log^3 n)$ construction time in our case.

In our solution we use generalized wavelet trees ([5], [10], [7], [13]), which have $O(\frac{n}{r} \log n)$ construction time in our case. With wavelet trees, SELECTION can be solved in $O(\frac{n}{r})$ space and in $O((occ_k + 1) \log n / \log \log n)$ time.

C. Resulting bound

We summarize the complexity bounds in the following theorem.

Theorem 2. *Identifying all occurrences of a pattern* P *of length* $m \geq r$ *in* T *takes* $O(m \cdot \max\{1, \frac{r \log \sigma}{w}\} + \max\{occ, r\} \cdot \log n / \log \log n))$ *time and* $O(\frac{n}{r})$ *space, where* occ *is the total number of output occurrences.*

Proof: Time taken by RIGHTSEARCH is $O(m \cdot \max\{1, \frac{r \log \sigma}{w}\})$. Time taken by LEFTSEARCH is $O(r \max\{1, \frac{r \log \sigma}{w}\})$. SELECTION takes $O((occ_k + 1) \log n / \log \log n)$ time for each k, and therefore $O(\max\{occ, r\} \cdot \log n / \log \log n)$ time overall.

Sparse suffix tree for T and suffix links take $O(\frac{n}{r})$ space, and same holds for the trie for $RevBlocks_r$. The wavelet

tree for Q takes $O(\frac{n}{r})$ space. ■

Note that in the case when $w = \Theta(\log n)$ (natural assumption under the RAM model) and $r = \Theta(\log_\sigma n)$, we obtain a fully linear pattern matching algorithm with respect to the pattern length running in $O(\frac{n}{r})$ space, which is the same performance as for the algorithm presented in [3]. When $r = log^2 n$, we achieve query time and additional space of the index for the internal memory model describe in [6].

Note for completeness that we have always assumed that the pattern length m is larger than r and, therefore, must cross at least one block boundary. In case $m < r < \log_\sigma n$, all occurrences of P located inside blocks can be reported in $O(m+occ)$ time and $O(\frac{n}{r})$ space with the method of [3].

IV. Construction of ST_r

Ukkonen and Kärkkäinen [9] described how to construct the sparse suffix tree in time $O(n)$ and space $O(n/r)$ (see also [1], [8], [12]). Denote by $l(v)$ the string represented by a node v in ST_r. The construction of [9] uses suffix links defined as follows: for an explicit node v of ST_r representing a string $\alpha = l(v)$, the suffix link of v points to the node representing $\alpha[r+1..]$. We call such suffix links *r-suffix links*. The definition is well-founded, as if a string α, $|\alpha| > r$, is represented in ST_r, then the string $\alpha[r+1..]$ is represented in ST_r too. Moreover, if α is represented by an explicit node, then so is $\alpha[r+1..]$.

Since our construction of sparse suffix tree differs from that of [9] only in the definition of suffix links, we assume that we already have the tree constructed by the algorithm of [9] equipped by r-suffix links and we have to set the suffix links as defined in Section II. We will consecutively set suffix links of type $1, 2, \ldots, r$.

For each explicit node v of ST_r, we fix an arbitrary occurrence of $l(v)$ in T starting at a block boundary. We then compile an array A of $\frac{n}{r}$ lists of nodes of ST_r. A node v belongs to the i-th list iff the fixed occurrence of $l(v)$ starts at position $ir+1$ in T. We assume that nodes in each list of A occur in the increasing order of string depths. A can be compiled by one breadth-first traversal of ST_r in $O(\frac{n}{r})$ time.

Consider some i, $0 \le i \le r-1$. Let β_i^j, where $0 \le j \le \frac{n}{r} - 1$ be the longest prefix of $T[rj+i+1..]$ represented in ST_r. First, the algorithm locates the (possibly implicit) nodes $v_0, v_1, \ldots, v_{\frac{n}{r}-1}$ of ST_r representing $\beta_i^0, \beta_i^1, \ldots, \beta_i^{r-1}$ respectively.

Locating $v_0, v_1, \ldots, v_{\frac{n}{r}-1}$ is done by simultaneously traversing ST_r and comparing characters of T from left to right, similarly to the algorithm of RightSearch. Suppose that we reach a (generally implicit) node v_j representing β_i^j. If $|\beta_i^j| \le r$, then we start over from the root and continue to follow T from character $T[r(j+1)+i+1]$. Otherwise we move up to the closest explicit ancestor v of v_j representing $\beta_i^j[..\ell]$ and follow the r-suffix link to get to a node u representing $\beta_i^j[r+1..\ell]$. As previously noted, this node exists and is explicit. We then proceed moving down from node u and comparing only the first characters of labels of edges with corresponding characters of $\beta_i^j[\ell+1..]$. When no move is possible any more, we have arrived at the node representing the longest prefix of $T[1+i+rj+r..]$ represented in ST_r, which is β_i^{j+1}.

The following lemma can be proved:

Lemma 2. *The algorithm above correctly locates the nodes $v_0, v_1, \ldots, v_{\frac{n}{r}-1}$ in time $O(n)$.*

Secondly, the algorithm builds suffix links of type i using the nodes $v_0, v_1, \ldots, v_{\frac{n}{r}-1}$.

Lemma 3. *Let u and v be two explicit nodes such that u is an ancestor of v (that is, $l(u)$ is a prefix of $l(v)$). Then the type of the suffix link of u is not larger than the type of the suffix link of v.*

The Lemma will insure that all nodes with suffix links of type i occur consecutively in the initial part of lists $A[j]$ (note that by induction, the nodes with suffix links of type smaller than i have been deleted from lists $A[j]$, see below) and if the head element of some $A[j]$ does not have a suffix link of type i, then no other element of $A[j]$ has one. Note also that a suffix link of type i of some node v in $A[j]$ must point to a node on the path from the root to v_j.

Hence, the main idea is to maintain a stack of nodes on the path from the root of ST_r to v_j to compute suffix links of type i for nodes of $A[j]$. Note that v_j's are implicit nodes in general, therefore some additional care is needed for this procedure.

In more details, we traverse ST_r depth-first and maintain a stack V (implemented as an array, i.e. allowing access to all stored elements) of size $O(\frac{n}{r})$ storing explicit nodes on the path from the root to the the current node of ST_r.

Assume that we are in a node v_j representing β_i^j, $0 \le j \le \frac{n}{r} - 1$. We check the head element v of the list $A[j]$. If the string depth $d(v)$ is less than $d(v_j)$, then the type of a suffix link from v is i. We find the first node u on the path from the root of ST_r to v_j with string depth bigger than $d(v)$ by a binary search on the elements of V. Obviously, the target node $s(v)$ is a (possibly implicit) node $(u, d(u) - d(v))$. After computing $s(v)$, v is deleted from $A[j]$. We repeat this procedure while string depth of the head element is less than $d(v_j)$ and then continue the tree traversal.

Let us now turn to time and space analysis. Recall that ST_r with r-suffix links is computed in $O(\frac{n}{r})$ time. To locate nodes $v_0, v_1, \ldots, v_{\frac{n}{r}-1}$ we need $O(n)$ time for a fixed i, and therefore $O(nr)$ time altogether. To compute all suffix links, we need $O(\frac{n}{r} \cdot \log \frac{n}{r} + n)$ time. Finally, to store V and A during tree traversals we need $O(\frac{n}{r})$ space.

V. Concluding remarks

In this paper, we introduced a new definition of suffix links in evenly spaced sparse suffix trees. Based on this structure, we proposed a new pattern matching algorithm that applies to any partitioning of the text into blocks of equal size. Assuming that a computer word is $\Theta(\log n)$ bits, we obtain essentially the same time and space bounds as those of [3], [6].

We believe that our definition of suffix links could bring further improvements to the pattern matching algorithm. In particular, we conjecture that one could completely avoid using an "external" data structure for orthogonal range queries and design an efficient algorithm based on the sparse suffix tree alone. Another challenging problem for future research is to get rid of the multiplicative factor depending on n in the *occ* term of the time bound (see Theorem 2).

Acknowledgment: The authors are grateful to Djamal Belazzougui for very helpful discussions. R.Kolpakov and T.Starikovskaya were partly supported respectively by grants 11-01-00508 and 10-01-93109-CNRS-a of the Russian Foundation for Basic Research.

References

[1] A. Andersson, J. Larsson, and K. Swanson. Suffix trees on words. In *Proc. of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, volume 1075 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 1996.

[2] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the 27th annual ACM symposium on Computational geometry*, SoCG '11, pages 1–10, New York, NY, USA, 2011. ACM.

[3] Y.-F. Chien, W.-H. Hon, R. Shah, and J. Vitter. Geometric burrows-wheeler transform: Linking range searching and text indexing. In *Proc. Data Compression Conference (DCC 2008)*, pages 252–261. IEEE Computer Society, 2008.

[4] Paolo Ferragina and Johannes Fischer. Suffix arrays on words. In *In Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, volume 4580 of LNCS*, pages 328–339. Springer, 2007.

[5] Paolo Ferragina, Giovanni Manzini, Veli Makinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3:2007, 2007.

[6] W.-K. Hon, R. Shah, Sh. Thankachan, and V. Vitter. On entropy-compressed text indexing in external memory. In *Proc. 16th Int. Symp. String Processing and Information Retrieval (SPIRE'09)*, volume 5721 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2009.

[7] W.-K. Hon, R. Shah, and J. Vitter. Ordered pattern matching: Towards full-text retrieval. Technical Report 06-008, Purdue University, March 2006.

[8] Shunsuke Inenaga and Masayuki Takeda. On-line linear-time construction of word suffix trees. In *in Proc. 17th Ann. Symp. on Combinatorial Pattern Matching (CPM06*, pages 60–71. Springer-Verlag, 2006.

[9] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Computing and Combinatorics Conference (COCOON'96)*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer Verlag, 1996.

[10] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN)*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer Verlag, 2006.

[11] Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Comput. Geom. Theory Appl.*, 42:342–351, May 2009.

[12] Takashi Uemura and Hiroki Arimura. Sparse and truncated suffix trees on variable-length codes. In *CPM*, pages 246–260. Springer-Verlag, 2011.

[13] C.-C. Yu, W.-K. Hon, and B.-F. Wang. Efficient data structures for the orthogonal range successor problem. In *Proceedings of the 15th Annual International Conference on Computing and Combinatorics*, COCOON '09, pages 96–105, Berlin, Heidelberg, 2009. Springer-Verlag.