Pattern Matching on Sparse Suffix Trees
Roman Kolpakov, Gregory Kucherov, Tatiana Starikovskaya

To cite this version:
Roman Kolpakov, Gregory Kucherov, Tatiana Starikovskaya. Pattern Matching on Sparse Suffix Trees. Data Compression, Communications and Processing (CCP), University of Salerno, Jun 2011, Palinuro, Italy. pp.92-97, 10.1109/CCP.2011.45. hal-00681844

HAL Id: hal-00681844
https://hal-upec-upem.archives-ouvertes.fr/hal-00681844
Submitted on 22 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 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 \( \sigma \) 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 \( T \).

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 \( O(m \cdot \max\{1, r \log_\sigma n\} + \max\{occ, r\} \cdot \log n/\log_\sigma n) \) 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 $\Sigma$ denote an alphabet, i.e. a set of letters or characters, of cardinality $\sigma$. We assume a lexicographic order $<_\Sigma$ on $\Sigma$, naturally extended to the set of all strings over $\Sigma$. Positions in strings are numbered from 1. For a string $\alpha$, $\alpha[i..j]$ denotes substring $\alpha[i] \cdots \alpha[j]$, and $\alpha[j..]$ is a shorthand for the suffix $\alpha[j..\alpha|\alpha|]$ 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 $T[rj + 1..]|j = 0, 1, \ldots, \frac{n}{r} − 1$ (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 $(v, \ell)$, where $v$ is the closest explicit ancestor node and $\ell$ is the offset with reference to $v$. Note that the leaves 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 $\alpha$ is represented in $ST_r$ if $\alpha$ is a prefix of one of the suffixes of $Suf_r$, i.e. if $\alpha$ is a substring of $T$ starting at a position $rj + 1$ for some $j$. In this case, $\alpha$ is the label of some node $v$ of $ST_r$, and we say that $\alpha$ 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 $\text{MinRank}(v)$ and $\text{MaxRank}(v)$ 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 $[\text{MinRank}(v), \text{MaxRank}(v)]$. If $\alpha$ is a word corresponding to $v$, then the ranks of suffixes of $Suf_r$ starting with $\alpha$ are specified by the interval $[\text{MinRank}(v), \text{MaxRank}(v)]$.

We assume that for each explicit node $v$ of $ST_r$, $\text{MinRank}(v)$ and $\text{MaxRank}(v)$, 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 $\alpha$, 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 $1 \leq i \leq r$. 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 $m \geq r$. 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 $P[1..], P[2..], \ldots, P[r..]$ 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.
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 \( (v_{\ell}, k) \) reaching the end of \( P[k+1..m] \) (line 14 of Algorithm 1). Then the algorithm retrieves the rank interval \([\text{MinRank}(v'), \text{MaxRank}(v')]\), 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 \( (v_{\ell}, k) \) 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

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

Lemma 1. RightSearch correctly identifies all suffixes \( P[k+1..], 0 \leq k \leq r-1, \) 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 to the suffix \( P[k+i+1..] \) 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 \( O \left( \frac{m}{t} \right) \).

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

### B. LEFTSEARCH and Selection

\( \text{LEFTSEARCH} \) locates all occurrences of \( P[1..k] \) ending at block boundaries of \( T \). Let \( \text{RevBlocks}_r \) be a set of the reversed blocks \( T[r(j-1) + 1..rj] \), for \( j = 1, \ldots , \sigma \). We build a compacted trie for \( \text{RevBlocks}_r \). Each leaf \( v \) of the trie is associated with the indices of the blocks it represents, namely with the set \( \{ j | 1 \leq j \leq \frac{r}{n} - 1 \} \) 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 \( \text{RIGHTSEARCH} \). Then we run the procedure described in \( \text{RIGHTSEARCH} \), but we use the trie instead of the sparse suffix tree and a pattern \( P[r-1]P[r-2]..P[1] \).

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

From Section IV it is obvious that the trie for \( \text{RevBlocks}_r \) and suffix links for it can be constructed in \( O(nr) \) time.

We now show how \( \text{SELECTION} \) can be reduced to 2D range reporting problem. Consider a rank interval \( I^k_{right} \) output by \( \text{RIGHTSEARCH} \) for some \( k \) (line 18 of Algorithm 1). We have to compute all \( j, 1 \leq j \leq \frac{r}{n} - 1 \), 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 \( \text{RevBlocks}_r \) 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 \( \frac{r}{n} \) points, where a point \( j \) has the first coordinate equal to the rank of \( T[rj + 1..] \) 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 \( \text{RevBlocks}_r \). 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).

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 \cite{2} has \( O(\max \{occ_k, 1 \} \log^4 n) \) query time, where \( \epsilon \) 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 \cite{11} has \( O(\log \log n + occ_k \log^3 n) \) query time and \( O(\frac{1}{\epsilon} \log n) \) construction time in our case.

In our solution we use generalized wavelet trees \cite{5}, \cite{10}, \cite{7}, \cite{13}, which have \( O(\frac{1}{\epsilon} \log n) \) construction time in our case. With wavelet trees, \( \text{SELECTION} \) can be solved in \( O(\frac{n}{\epsilon}) \) 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 \( O( \max \{1, \frac{r \log \sigma}{w} \}) + \max \{occ, r \} \cdot \log n / \log \log n \) time and \( O(\frac{n}{\epsilon}) \) space, where \( occ \) is the total number of output occurrences.

**Proof:** Time taken by \( \text{RIGHTSEARCH} \) is \( O( \max \{1, \frac{r \log \sigma}{w} \}) \). Time taken by \( \text{LEFTSEARCH} \) is \( O( \max \{1, \frac{r \log \sigma}{w} \}) \). \( \text{SELECTION} \) 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 \( O(\frac{n}{\epsilon}) \) space, and same holds for the trie for \( \text{RevBlocks}_r \). The wavelet
tree for $Q$ takes $O(\frac{n}{\sigma})$ space.

Note that in the case when $w = \Theta(\log n)$ (natural assumption under the RAM model) and $r = \Theta(\log n)$, we obtain a fully linear pattern matching algorithm with respect to the pattern length running in $O(\frac{n}{\sigma})$ 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 described 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 $m < r < \log_2 n$, all occurrences of $P$ located inside blocks can be reported in $O(m + occ)$ time and $O(\frac{n}{\sigma})$ 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 $\alpha$, a suffix link as defined in Section II, we assume node, then so is $\alpha[r+1..]$. Let compile an array $v_0, v_1, \ldots, v_{\frac{n}{r}-1}$ in time $O(n)$. Moreover, if $\alpha$ is represented by an explicit node, then so is $\alpha[r+1..]$. If 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 $O(\frac{n}{\sigma})$ time.

Consider some $i$, $0 \leq i < r - 1$. Let $\beta_i^r$, where $0 \leq j \leq \frac{n}{r} - 1$ 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 $\beta_0, \beta_1, \ldots, \beta_{r-1}$ 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 $\beta_j^r$. If $|\beta_j^r| \leq r$, 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 $\beta_j^r[..\ell]$ and follow the $r$-suffix link to get to a node $u$ representing $\beta_j^r[r + 1..\ell]$. 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 $\beta_j^r[\ell + 1..]$. 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 $\beta_j^{r+1}$. 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 $O(\frac{n}{\sigma})$ storing explicit nodes on the path from the root to the the current node of $ST_r$.

Assume that we are at a node $v_j$ representing $\beta_j^r$, $0 \leq j \leq \frac{n}{r} - 1$. 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 $O(\frac{n}{\sigma})$ 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 $O(\frac{n^2}{\sigma} \cdot \log \frac{n}{\sigma} + n)$ time. Finally, to store $V$ and $A$ during tree traversals we need $O(\frac{n}{\sigma})$ space.
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 \( \text{occ} \) term of the time bound (see Theorem 2).

Acknowledgment: The authors are grateful to Djamaal 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


