Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 302 (2014) 73–94

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# A Compressed Suffix Tree Based Implementation With Low Peak Memory Usage[1]

Daniel Saad Nogueira Nunes[2] Mauricio Ayala-Rincón[3]

*Instituto de Ciências Exatas*
*Departamentos de Ciência da Computação e Matemática*
*Universidade de Brasília*
*Brasília, Brazil*

Abstract

Suffix trees ($\mathcal{ST}$s) and suffix arrays are well known indices which demand too much space for large inputs. Recently, several works explore a data structure called compressed suffix tree ($\mathcal{CST}$), which offers the same functionality than suffix trees and is based on compressed suffix arrays, compressed longest common prefix information and navigational operations. In this paper, the implementation of a $\mathcal{CST}$ based on range-minimum-queries and nearest smaller value queries, which requires roughly more than the space needed to represent the index during the construction, is presented. Experiments show that this index is useful for many applications since, on the one side, one can execute complex traversals such as *suffix links* and *longest common ancestor* queries that are essential to deal with several questions about the combinatorial structure of sequences; and, on the other side, the structure results of practical interest for applications using computational environments in which the amount of available memory is restricted, because it fits in main memory of ordinary computers.

*Keywords:* suffix trees, suffix arrays, longest common prefix, compressed suffix trees, String Processing.

## 1 Introduction

Suffix trees ($\mathcal{ST}$s) are very versatile data structures well known by their wide range of applications in several areas of Computer Science such as Computational Biology, as shown in [14]. The $\mathcal{ST}$ of a text encodes all its suffixes in a compact way in comparison with other data structures such as tries.

[1] Work supported by a PRONEX grant from the Federal District Research Foundation FAPDF and a travel grant from the Foundation for Scientific and Technological Enterprises FINATEC.

[2] Email: daniel.saad.nunes@gmail.com
Author supported by a graduate scholarship from the Coordination for Enhancement of Higher Education Personnel CAPES.

[3] Email: ayala@unb.br
Author partially supported by a grant from the National Counsel of Technological and Scientific Development CNPq.

http://dx.doi.org/10.1016/j.entcs.2014.01.021

Several suffix tree operations as suffix link and lowest common ancestor queries ($LCA$) are essential to solve relevant problems. Thus, in practice, a $\mathcal{ST}$ with full functionality, supporting many complex traversals, is often required.

The major problem of $\mathcal{ST}$s is the space consumption in practice. Although the $\mathcal{ST}$ of a text of length $n$ asymptotically requires $\Theta(n \log n)$ bits, in practice one needs a factor of $\approx \times 15$ over the size of the input [19], which results in 45GB of main memory for building the structure for the human genome, for instance.

Suffix arrays were proposed in order to solve string processing problems using less space than $\mathcal{ST}$s [23]. However, the information of suffix arrays correspond only to leaves of $\mathcal{ST}$s in lexicographical order. Nevertheless, suffix arrays can be enhanced with longest common prefix information ($LCP$), which implicitly encodes the topology of the associated $\mathcal{ST}$. Hence, a suffix array can replace entirely a $\mathcal{ST}$ if provided with $LCP$ information and navigational operations over it [2].

Although less space consuming, suffix arrays still require $\Theta(n \log n)$ bits, usually 4 bytes per input symbol, or 8 bytes if enhanced with the $LCP$ information. The amount of space required for the structure is still huge, not allowing their use for moderately large sized texts.

Recently, a great amount of work has been done in order to compress the suffix array data structure as shown in [12] and [8]. The compressed structure can achieve $\Theta(nH_0)$ bits of space, where $H_0$ stands for the zero order entropy. Consequently, one needs less space in practice to represent the suffix array information, although the compression increases the query time over the structure. Generally, the factor of compression is very large. One can represent the structure within few bits per input symbol, as shown in [16].

It's also possible to represent the $LCP$ information in a compressed way using $\Theta(n)$ bits [30]. Hence, using compressed data structures for navigational operation in $LCP$, it's possible to emulate a $\mathcal{ST}$ using $\Theta(nH_0)$ bits of space, which represent a huge economy of space in practice.

This compressed data structure composed by compressed suffix array, compressed $LCP$ information and compressed navigational information is called compressed suffix tree ($\mathcal{CST}$), and was introduced by K. Sadakane in [31]. Despite a few additional amount of bits being necessary to represent the $\mathcal{CST}$ with support to complex traversals , more time is required to answer queries. Usually a factor of $\Theta(\log^{\epsilon} n)$, $\epsilon > 0$, is present in each suffix array, $LCP$, or navigation operations. The trade-off between space and time is highly acceptable when high quantities of memory are not available.

Our contribution is an implementation of a $\mathcal{CST}$ with low memory peak usage if compared with other state-of-art indices. It uses a compressed data structure based on [5] for range-minimum-queries ($RMQ$), previous smaller value queries ($PSV$) and next smaller value queries ($NSV$), to navigate in the topology encoded by the compressed $LCP$ information. In order to reduce the peak memory required during the construction of the structure, instead of compressing a raw suffix array, an incremental algorithm to build the compressed suffix array within $\Theta(nH_0)$ working space was used [15]. This incremental algorithm avoids building a raw suffix array and then compressing it due to the online approach used.

In order to evaluate the proposed implementation, experiments were performed with two indices. The first one is based on a uncompressed suffix array provided by [25] and uncompressed $LCP$ information calculated with the method from [18]. The second index is a $\mathcal{CST}$ proposed by the Succinct Data Structures group (SuDS) from the university of Helsinki [34] [33], which is based on [31].

The paper is organized as follows. All necessary basic concepts are given in Section 2; compressed suffix arrays and $LCP$ are presented in Section 3 and then, in Section 4, compressed suffix trees are introduzed. Then, in Sections 5 and 6 related work and experimental results are presented, before concluding and presenting future work in Section 7.

## 2 Basic Concepts

An alphabet $\Sigma$ is a finite set of symbols $\{\alpha_0, \alpha_1, \ldots, \alpha_{\sigma-1}\}$ with a total order $\alpha_0 < \alpha_1 < \ldots < \alpha_{\sigma-1}$, where $|\Sigma| = \sigma$, the cardinality or size of the alphabet. Texts are finite strings of symbols of $\Sigma$. The $i^{th}$ symbol of a text $T$ of length $n$ is denoted by $T[i]$ and its substring from the $i^{th}$ until the $j^{th}$ symbol as $T[i, j]$, $0 \leq i \leq j \leq n-1$. The suffix $T[i, n-1]$ is denoted by $T_i$. It is assumed that the last character of any text $T$ is the symbol \$, which belongs to $\Sigma$ and is lexicographically smaller than any other symbol of $\Sigma$ and that occurs only in the last position of $T$. Given to texts $R$ and $S$, it is said that $R =^k S$ whenever the first $k$ symbols of both strings are equal.

According to [14], a $\mathcal{ST}$ for $T$ is a rooted directed tree with exactly $n$ leaves numbered from 0 to $n-1$. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of $T$. Edges out of a node cannot have edge-labels beginning with the same character. The key feature of $\mathcal{ST}$s is that the concatenation of the edge-labels on the path from the root to any leaf $i$ spells $T_i$. The Figure 1 shows a suffix tree for the text $T = acaaacatat\$$.

A suffix array $\mathcal{A}$ is an array of integers containing the starting position of suffixes of $T$ in the lexicographical order induced by the order of the symbols. Thus, $T_{\mathcal{A}[0]} < T_{\mathcal{A}[1]} < \ldots < T_{\mathcal{A}[n-1]}$.

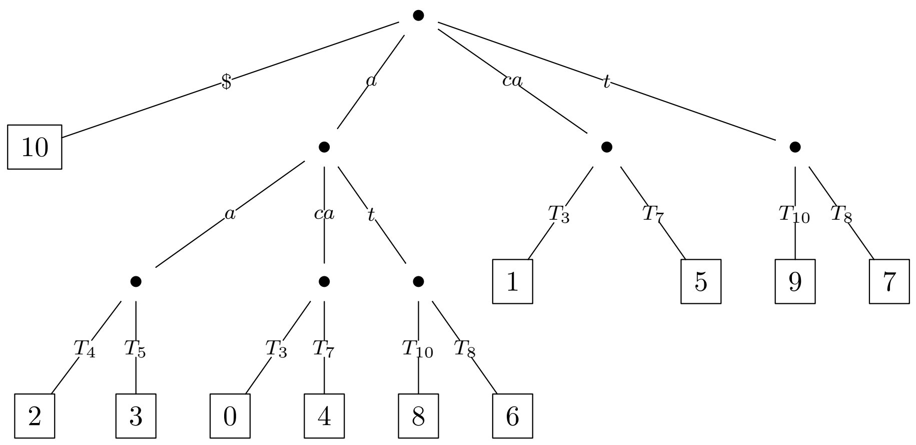

Figure 1. Suffix Tree for $T = acaaacatat\$$.

Table 1
Suffix array for $acaaacatat\$$ and $LCP$ information.

| $i$ | $T_{\mathcal{A}[i]}$ | $\mathcal{A}[i]$ | $LCP[i]$ |
|---|---|---|---|
| 0 | $\$$ | 10 | 0 |
| 1 | $aaacatat\$$ | 2 | 2 |
| 2 | $aacatat\$$ | 3 | 1 |
| 3 | $acaaacatat\$$ | 0 | 3 |
| 4 | $acatat\$$ | 4 | 1 |
| 5 | $at\$$ | 8 | 2 |
| 6 | $atat\$$ | 6 | 0 |
| 7 | $caaacatat\$$ | 1 | 2 |
| 8 | $catat\$$ | 5 | 0 |
| 9 | $t\$$ | 9 | 1 |
| 10 | $tat\$$ | 7 | 0 |

The $LCP$ information holds the length of the longest common prefix between two adjacent entries of a suffix array. Formally, it is defined as:

$$LCP[i] = \begin{cases} \max\{k \mid T_{\mathcal{A}[i]} =_k T_{\mathcal{A}[i+1]}\}, & i < n-1 \\ 0, & i = n-1 \end{cases} \tag{1}$$

An example of suffix array and $LCP$ information for $T = acaaacatat\$$ is shown in Table 1.

## 3 Compressing Suffix Arrays and *LCP*

The compression of a suffix array can be achieved by sampling only few entries of this data structure in such a way that non sampled entries are recoverable by computation. If the sample factor is $\mathcal{K} \in \Theta(\log n)$ one would need only $\Theta(n)$ bits to represent the sampled entries.

The core of a compressed suffix array data structure is the $\Psi$ function, which allows us to navigate through the suffix array. This function is defined as:

$$\Psi(i) = j, \quad \mathcal{A}[j] = (\mathcal{A}[i] + 1) \mod n \tag{2}$$

In other words, if $\mathcal{A}[i] = k$, then $\Psi(i)$ gives us the lexicographical order of suffix $T_{(k+1) \mod n}$ among all other suffixes.

Naively, $\Psi$ would be represented using $\Theta(n \log n)$ bits. However it is possible to represent $\Psi$ within $\Theta(nH_0)$ bits. Since the array is in lexicographical order, $\Psi$ holds an increasing sequence for suffixes starting with the same symbol, as illustrated in Table 2. Therefore, it is possible to encode each increasing sequence in $\Theta(n)$ bits using Rice coding while allowing $\Theta(1)$ time query to any position of the $\Psi$ values using *Rank* and *Select* queries on bitvectors, as shown in [12]. Although there are $\Theta(1)$ solutions for *Rank* and *Select* queries, it is often obtained an $\omega(1)$ result in practice because of better performance and less space consumption, as shown in [11]. The access time for $\Psi$ is denoted by $t_\Psi$.

Table 2
$\Psi$ function walks through the suffix array.

| $i$ | $T_{\mathcal{A}[i]}$ | $\mathcal{A}[i]$ | $\Psi[i]$ | Flag |
|---|---|---|---|---|
| 0 | \$ | 10 | 3 | 1 |
| 1 | *aaacatat*\$ | 2 | 2 | 0 |
| 2 | *aacatat*\$ | 3 | 4 | 0 |
| 3 | *acaaacatat*\$ | 0 | 7 | 0 |
| 4 | *acatat*\$ | 4 | 8 | 0 |
| 5 | *at*\$ | 8 | 9 | 1 |
| 6 | *atat*\$ | 6 | 10 | 0 |
| 7 | *caaacatat*\$ | 1 | 1 | 0 |
| 8 | *catat*\$ | 5 | 6 | 0 |
| 9 | *t*\$ | 9 | 0 | 0 |
| 10 | *tat*\$ | 7 | 5 | 1 |

Once the $\Psi$ data structure is built, one can recover non sampled suffix arrays entries by walking through the suffix array using this function. Using Table 2 as an example, $\mathcal{A}[2]$ is not sampled, therefore the $\Psi$ function is called 4 times until a sampled value is found; hence, $A[2] = A[10] - 4 = 7 - 4 = 3$. On average, an access to $\mathcal{A}[i]$, denoted by $t_\mathcal{A}$, takes $O(t_\Psi \mathcal{K})$, as shown in [16].

The proposed $\mathcal{CST}$ builds the compressed suffix array using an incremental algorithm from [15]. The algorithm only $\Theta(nH_0)$ bits of working space and $\Theta(n \log n)$

Table 3
$LCP[i] + \mathcal{A}[i]$ holds an increasing sequence if suffixes are examined in the text order.

| $i$ | $T_{\mathcal{A}[i]}$ | $\mathcal{A}[i]$ | $LCP[i]$ | $\mathcal{A}[i] + LCP[i]$ |
|---|---|---|---|---|
| 0 | \$ | 10 | 0 | 10 |
| 1 | *aaacatat*\$ | 2 | 2 | 4 |
| 2 | *aacatat*\$ | 3 | 1 | 4 |
| 3 | *acaaacatat*\$ | 0 | 3 | 3 |
| 4 | *acatat*\$ | 4 | 1 | 5 |
| 5 | *at*\$ | 8 | 2 | 10 |
| 6 | *atat*\$ | 6 | 0 | 6 |
| 7 | *caaacatat*\$ | 1 | 2 | 3 |
| 8 | *catat*\$ | 5 | 0 | 5 |
| 9 | *t*\$ | 9 | 1 | 10 |
| 10 | *tat*\$ | 7 | 0 | 7 |

time, and thus, creates the compressed suffix array without building the raw suffix array first. The idea is to compute $\Psi$ incrementally from the end to the beginning of the text. For each block of size $\mathcal{B} \in \Theta(\frac{n}{\log n})$, one needs to calculate the position of the new added suffixes among them and the position of each new added suffix among the existing ones, and finally use this new information to build the $\Psi$ function for the added suffixes. There are $\mathcal{B} \in \Theta(\log n)$ blocks overall.

It was shown in [30] that $LCP$ information can be compressed using at most $2n$ bits for its representation, while fast access is allowed. This is possible because $LCP[i] + \mathcal{A}[i]$ values hold an increasing sequence if the entries are examined in decreasing length of suffixes. Thus, it can be represented by the same methods used to represent $\Psi$. For example, in Table 3, there is an increasing sequence $(3, 3, 4, 4, 5, 5, 6, 7, 10, 10, 10)$ for $\mathcal{A}[i]+LCP[i]$ values when the entries are examined in decreasing order of suffix lengths. The methodology from [18] computes the $LCP$ information based on this order, so was the method of choice of the proposed $\mathcal{CST}$. Since the encoded information is not stored in lexicographical order, one needs an additional access to $\mathcal{A}[i]$ to retrieve $LCP[i]$, thus, an access to $LCP[i]$ takes $t_{LCP} \in \Theta(t_{\mathcal{A}})$.

## 4 Compressed Suffix Trees

Once the $LCP$ information implicitly encodes the associated suffix tree topology, one needs to navigate through this information in order to make complex tree traversals. But in first place, it is necessary to identify the topology in the $LCP$ information. And this can be done using the concept of $\ell$-interval, which owns a one-to-one correspondence with the internal nodes of the associated $\mathcal{ST}$ [2]. An interval $[i, j]$ is an $\ell$-interval, denoted by $\ell - [i, j]$, if:

$$\begin{aligned} & i = 0 \vee LCP[i-1] < \ell \\ & LCP[k] \geq \ell,\ i \leq k < j \\ & LCP[k] = \ell,\ \text{for one } i \leq k < j \\ & j = n-1 \vee LCP[j] < \ell \end{aligned} \tag{3}$$

The positions in an $\ell - [i, j]$ interval with $LCP$ value equal to $\ell$ are called $\ell$-indices.

Another useful concept is the concept of child interval. Given that $\ell - [i, j]$ is an interval, $\ell' - [i', j']$ is embedded in an $\ell - [i, j]$ interval if it is a sub-interval of $[i, j]$. It is said that $\ell - [i, j]$ encloses $\ell' - [i', j']$. If $\ell - [i, j]$ encloses $\ell' - [i', j']$ and there is no other interval embedded in $\ell - [i, j]$ that also encloses $\ell' - [i', j']$, then $\ell' - [i', j']$ is called a child interval of $\ell - [i, j]$.

These intervals correspond to internal nodes in $\mathcal{ST}$s because they are maximal (can neither be extended to the left nor to the right) and every suffix shares a prefix of length $\ell$. Analogously, an internal node of $\mathcal{ST}$ can not be extended, because every edge starts with a different symbol and the leaves which are below the node, share a prefix of length equal to the string depth of the internal node. Besides, the parent-child-intervals represent a relationship of parent and child in the associated $\mathcal{ST}$. The leaves are represented by the suffix array $\mathcal{A}$ itself [2].

Once the relation of the $LCP$ with the topology of the tree is understood, navigational operations are necessary to emulate suffix tree traversals. The proposed compressed $\mathcal{ST}$ is based on $RMQ$, $NSV$ and $PSV$ queries over the $LCP$ information to navigate in the tree. Formally they are defined as:

$$RMQ(i, j) = \min\{\arg \min_{i \leq k \leq j} \{LCP[k]\}\} \tag{4}$$

$$PSV(i) = -1 \vee \max_{0 \leq k < i} \{LCP[k] < LCP[i]\} \tag{5}$$

$$NSV(i) = (n-1) \vee \min_{i < k \leq n-1} \{LCP[k] < LCP[i]\} \tag{6}$$

With a single data structure proposed in [5], one can handle these three queries. The data structure is a tree whose leaves have the same height and correspond to the $LCP$ information. Let $\mathcal{L}$ be the number of siblings allowed for any node in this tree. Sibling leaves correspond logically to a block. Each parent in this tree is labeled with the leftmost position considering all the positions of minimum $LCP$ value of its children. An example of this structure corresponding to the Table 3 is presented in Figure 2. In this example, the leaves correspond to the $LCP$ information $(0, 2, 1, 3, 1, 2, 0, 2, 0, 1, 0)$ and the parent of the leaves labeled with $(3, 1, 2)$ is labeled with the position 4, which is the leftmost position in the $LCP$ array that occurs the minimum value 1 among $(3, 1, 2)$. Following this idea, the root is labeled with 0 because the leftmost position in which the minimum $LCP$ value occurs between the positions $(0, 10)$ of the $LCP$ array is the position 0.

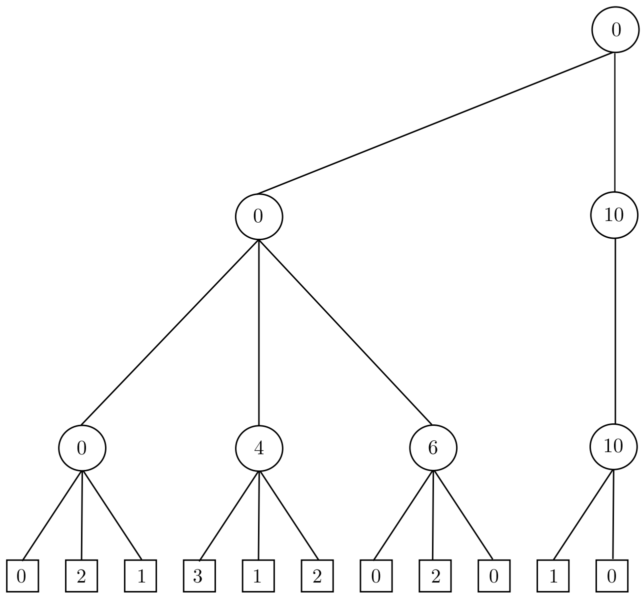

Figure 2. Data structure introduced by [5] for Table 3.

To answer $NSV(i)$ queries, one must scan the $i$'s block and if the value is not present, a bottom-up search is done until a node with a $LCP$ value lesser than $LCP(i)$ is found. Then, a top-down search is executed in order to find the block in which the $NSV$ query will be answered. $PSV$ queries work in a symmetric way.

For $RMQ(i, j)$ queries, one must scan $i$'s block and $j$'s block to find a local answer and then, execute a bottom-up search to calculate the $RMQ$ between $i$'s block and $j$'s until a common ancestor of the underlying leaves is achieved.

Since both queries need a top-down and/or a bottom-up traversals of the tree of height $O(\log_{\mathcal{L}}(n)) = O(\log(n/\mathcal{L}))$ and either an analysis of the block or the $\mathcal{L}$ children, the running time for $RMQ$, $NSV$ and $PSV$ queries is $O(t_{LCP} \cdot \mathcal{L} \log(n/\mathcal{L}))$. The space used by the structure belongs to $o(n)$ bits if $\mathcal{L} = \omega(\log n)$. Running time for $RMQ$ queries is denoted by $t_{RMQ}$. For $PSV$ and $NSV$ queries $t_{PNSV}$ is used.

Traversals and navigational operations supported by the $\mathcal{CST}$ implementation are shown in Table 4.

Most of these operations were implemented as suggested by [10] with some slight differences. The methodology to execute each navigational operation is described below. Assume the node $u$ is identified by $\ell - [i, j]$ if it is not a leaf and node $v$ is identified by $\ell' - [i', j']$ if it is not a leaf either.

- Root: returns the root node of the $\mathcal{ST}$. This can be done returning the $0 - [0, n-1]$ interval in $O(1)$ time.
- Leaf$(u)$: returns true if $u$ is a leaf, false otherwise. One just have to check if the interval $[i, j]$ from $u$ is a singleton interval, i.e, $i = j$. It is possible to check the

Table 4
Supported operations of the proposed implementation

| Operation | Description | Complexity |
|---|---|---|
| Root | The root of the $\mathcal{ST}$ | $O(1)$ |
| Leaf$(u)$ | True if $u$ is a leaf, false otherwise | $O(1)$ |
| Locate$(u)$ | The $A[u]$ value | $O(t_{\mathcal{A}})$ |
| Parent$(u)$ | The parent of node $u$ | $O(t_{PNSV})$ |
| Children$(u)$ | The children of node $u$ | $O(\sigma \cdot t_{RMQ})$ |
| Edge$(u, v)$ | Edge label between nodes u and v | $O(t_{\mathcal{A}} + \|S\|)$ |
| Child$(v, c)$ | Child following an specific edge label | $O(\sigma \cdot t_{RMQ})$ |
| Depth$(v)$ | String depth from the root to $u$ | $O(t_{RMQ})$ |
| LCA$(u, v)$ | $LCA$ between $u$ and $v$ | $O(t_{RMQ})$ |
| SLink$(v)$ | Suffix link of node $u$ | $O(t_{RMQ})$ |

equality in $O(1)$ time.

- Locate$(u)$: returns the position of the suffix associated with a $\mathcal{ST}$ leaf. Node $u$ must be a leaf, hence one just have to lookup $A[i]$.
- Parent$(u)$: returns the parent of the $u$ node. The string depth of Parent$(u)$ must be the greatest among $(LCP[i-1], LCP[j])$. Let $k$ be the position which occurs the maximum between $LCP[i-1]$ and $LCP[j]$. Thus, the parent interval is given by $[PSV(k)+1, NSV(k)]$. The operation has $O(t_{PNSV})$ total cost. This operation is illustrated by Figure 3.

$$k = \arg\max\{LCP(i-1), LCP(j)\}$$

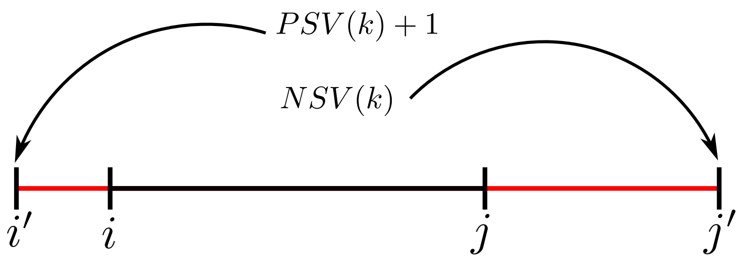

Figure 3. Parent operation.

- Children$(u)$: returns the children of the node $u$. If $u$ is a leaf, it returns NULL. The child intervals of a node u, are determined by its $\ell$-indices which are denoted by $\{\imath_1, \imath_2, \ldots, \imath_k\}$. The $\ell$-indices can be recovered through $RMQ$ queries. So the first child is determined by $[i, \imath_1 = RMQ(i, j-1)]$, the second by $[\imath_1 + 1, RMQ(\imath_1, j-1)]$, and so on, until the last child is determined by $[\imath_k, j]$. Since at most $\sigma$ children are possible, the complexity of this operation is $O(\sigma \cdot t_{RMQ})$.

Figure 4 shows the Children$(u)$ operation.

$$\imath_1 = RMQ(i, j-1)$$
$$\imath_2 = RMQ(\imath_1 + 1, j-1)$$

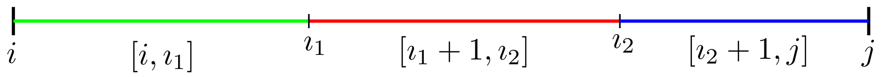

Figure 4. Children operation.

- Edge$(u, v)$: returns the label $S$ of the edge between $u$ and $v$. This is given by $T[\mathcal{A}[i'] + \ell, \mathcal{A}[i'] + \ell + |S| - 1]$. Hence, one needs $O(t_{\mathcal{A}} + |S|)$ time to recover the desired information.
- Child$(u, c)$ returns the child of $u$ which is linked by an edge that begins with the symbol $c$. This can be done collecting the children of $u$ and checking their first symbol using the technique from Edge operation. Since a node has at most $\sigma$ children, the operation takes $O(\sigma \cdot t_{RMQ})$ time.
- Depth$(u)$: returns the string depth of node $u$. If $u$ is a Leaf, it returns $\mathcal{A}[u]$; otherwise, returns $\ell = LCP[RMQ(i, j-1)]$. Therefore, the operation requires $O(t_{RMQ})$ time.
- LCA$(u, v)$: returns the lowest common ancestor of $u$ and $v$ .The $LCA$ of $u$ and $v$ is an $\ell'' - [i'', j'']$ interval. Let $k = RMQ(j, i' - 1)$. The common prefix (and string depth) $\ell''$ can be identified as $LCP[RMQ(j, i' - 1)]$ and the boundaries as $[PSV(k) + 1, NSV(k)]$. Hence, the operation needs $O(t_{RMQ})$ time. This operations is ilustrated by Figure 5.

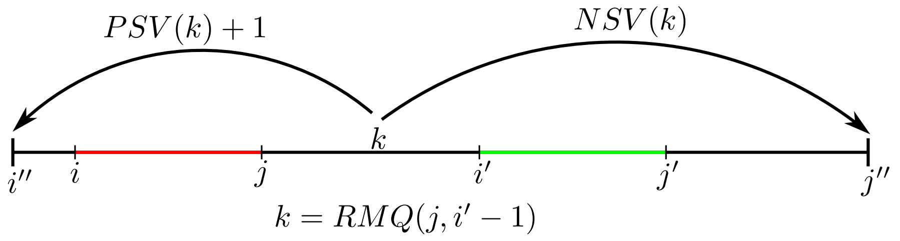

Figure 5. $LCA$ operation.

- SLink$(u)$: returns the node $v$ which is linked with $u$ by a suffix-link. If $u$ is a leaf, it returns $\Psi(i)$; otherwise, it follows the suffix-links for $\Psi(i)$ and $\Psi(j)$, finds the $\ell$ value by setting $k = LCP[RMQ(i, j-1)]$ and then, finds the boundaries with $[PSV(k) + 1, NSV(k) + 1]$. The operation takes $O(t_{RMQ})$ time since it is dominated by the $RMQ$ query. Figure 6 illustrates the suffix link operation.

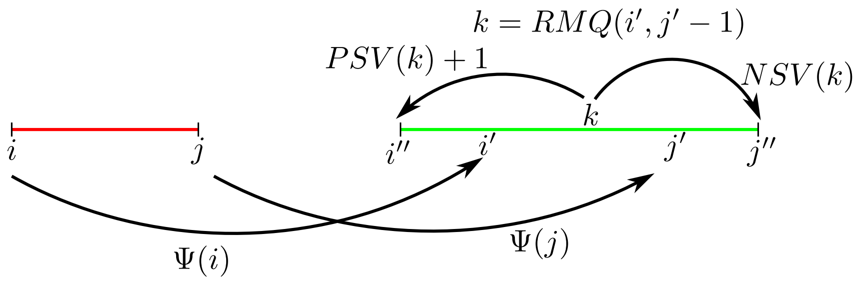

Figure 6. Suffix link operation.

## 5 Related Work

The Figure 7 presents a chronology of related work including relevant works on compressed and uncompressed $\mathcal{ST}$ data structures. It helps one to establish a connection between our results and the others.

Initially, suffix trees were introduced in the 1970's decade by Weiner [35], giving birth to several works in which the expressive power and computational capabilities of this structure were better explained and thoroughly understood, such as the outstanding works of McCreight's [24], Apostolico's [3], Ukkonen's [32] and Farach [6].

In the begging of the 1990's decade, Udi Manber and Gene Myers pointed out that suffix trees had an excessive space consumption in practice. Thus, they proposed the suffix arrays [23]. In their work the authors also introduced techniques to deal with the notion of longest common prefix.

Still in the ambit of suffix arrays, Kasai *et. al* proposed a $O(n)$ method to compute the $LCP$ information [18]. Abouelhoda *et.al* showed many applications that have arisen from the suffix array data structure and $LCP$ information [1]. The same authors, in another work, showed that every problem solvable with the help of a suffix tree is also solvable by replacing suffix trees by enhanced suffix arrays with the same time complexity [2]. Later, Puglisi *et. al* introduced a taxonomy for the suffix arrays construction algorithms algorithms [28].

In the 2000's decade, Grossi and Vitter introduced a data structure that is more space-efficient than suffix arrays. They called this data structure compressed suffix arrays [13]. This structure achieved an $O(nH_0) \subseteq o(n \log n)$ bits space requirement result for its final representation, but its construction required $\Theta(n \log n)$ bits. Simultaneously, Ferragina and Mazini described the FM-Index [7], which essentially corresponds to compressed suffix arrays. Still in 2000, Sadakane proposed a full index based on the compressed suffix array [29]. Shortly afterwards, the same author introduced an algorithm for building the compressed $LCP$ information in $\Theta(n)$ bits [29]. Later in the same decade, variations of the compressed suffix array, as for example the RLFM index [22], were introduced. Concerned about the space consumption during the compressed suffix arrays construction, Hon *et. al* developed an incremental algorithm which builds the compressed suffix array within $\Theta(nH_0)$ bits of working space [15].

Since it was possible to compress both the suffix array and the $LCP$ information,

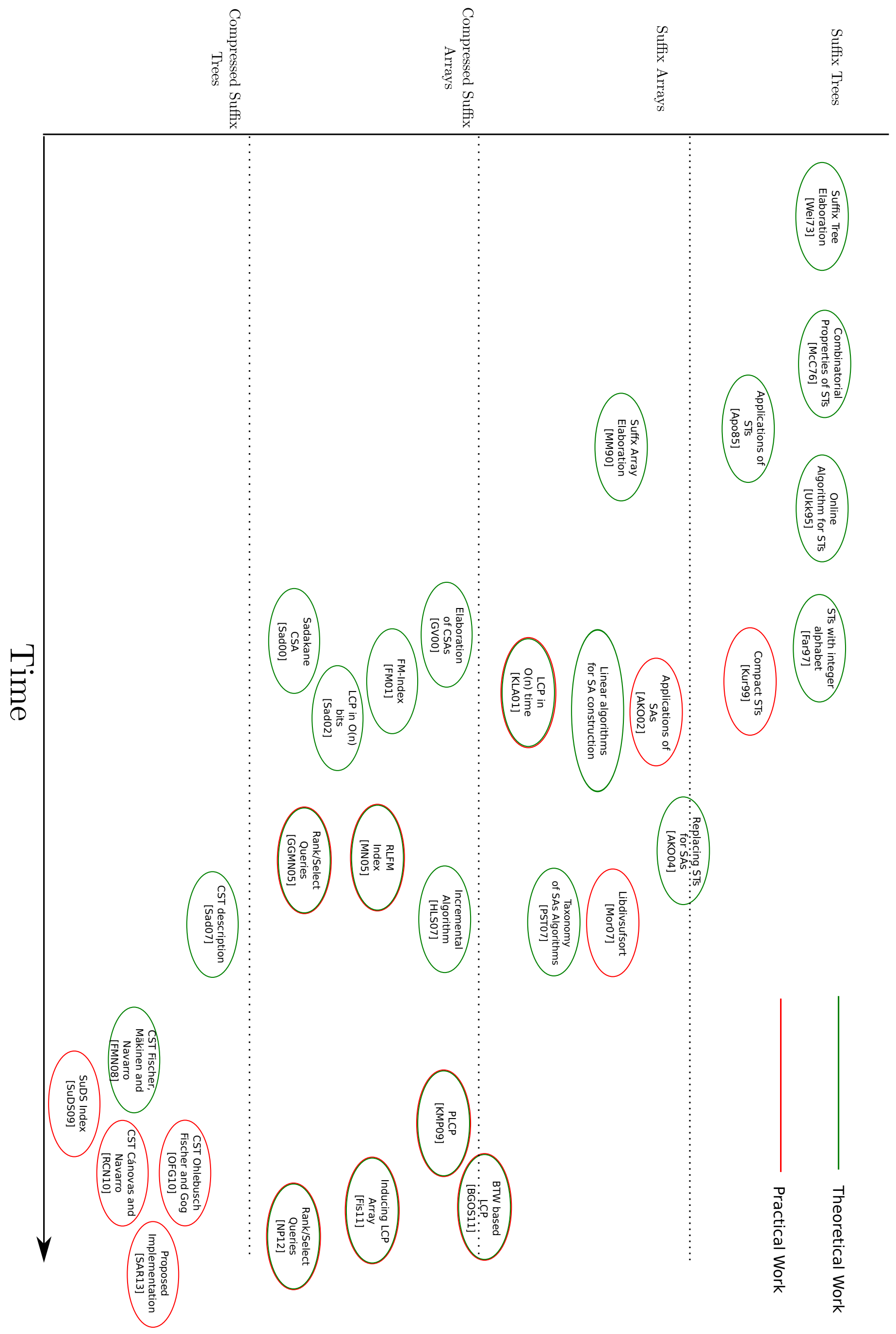
Suffix Trees
Suffix Arrays
Compressed Suffix Arrays
Compressed Suffix Trees
Time
Theoretical Work
Practical Work
Suffix Tree Elaboration [Wei73]
Combinatorial Properties of STs [McC76]
Applications of STs [Apo85]
Online Algorithm for STs [Ukk95]
STs with integer alphabet [Far97]
Compact STs [Kur99]
Suffix Array Elaboration [MM90]
Applications of SAs [AKO02]
Linear algorithms for SA construction
LCP in O(n) time [KLA01]
Replacing STs for SAs [AKO04]
Libdivsufsort [Mor07]
Taxonomy of SAs Algorithms [PST07]
Elaboration of CSAs [GV00]
FM-Index [FM01]
Sadakane CSA [Sad00]
LCP in O(n) bits [Sad02]
RLFM Index [MN05]
Rank/Select Queries [GGMN05]
Incremental Algorithm [HLS07]
PLCP [KMP09]
BTW based LCP [BGOS11]
Inducing LCP Array [Fis11]
Rank/Select Queries [NP12]
CST description [Sad07]
CST Fischer, Mäkinen and Navarro [FMN08]
SuDS Index [SuDS09]
CST Cánovas and Navarro [RCN10]
CST Ohlebusch Fischer and Gog [OFG10]
Proposed Implementation [SAR13]

Figure 7. Related Work.

the first works about compressed suffix trees appeared; among them, we can mention Sadakane's work [31]. After this work, other works such as Fischer's *et. al* [10], Välimäki's *et. al* [34], Cánovas and Navarro's [5] and Ohlebusch's *et. al* [27] have come up with both theoretical and practical variations of compressed suffix trees.

Finally, our work also aims to contribute to the improvement of this space-efficient data structure by using:

- the incremental algorithm of Hon *et. al* [15] in the compressed suffix array construction;
- the $\Theta(n)$ bits $LCP$ result by Sadakane [30] and
- the $NSV$, $PSV$ and $RMQ$ queries for navigational operations on the $\mathcal{CST}$, which also appeared in [10] and [5].

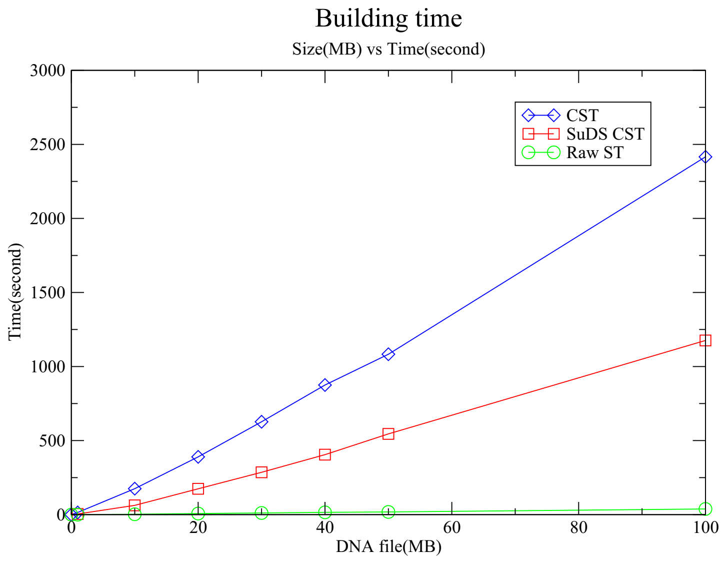

Figure 8. Building time for $\mathcal{K}=10$ and $\mathcal{L}=8$.

## 6 Experimental Results

In order to evaluate the implementation of the proposed $\mathcal{CST}$, a comparison with a raw $\mathcal{ST}$ based on suffix array and $LCP$ information was carried out. Comparisons with the $\mathcal{CST}$ provided by the SuDS group in [33] were performed as well.

Initially, the suffix array of the uncompressed structure was built with the library `libdivsufsort` [25]. It can build a suffix array very quickly using $5n + O(1)$ bytes per input symbol and $\Theta(n \log n)$ time. Afterwards, the $LCP$ information was calculated through Kasai *et. al*'s method in [18] without compression as well. Finally, the data structure chosen for the $RMQ$ and $PSV$ queries was based on the one in [5], that is the same used in the proposed implementation.

Experiments were carried out with several random DNA texts with different sizes generated with a simple C++ program. The sizes were under 100MB because the implementation supports only 32-bits operations.

The experiments were run under a 64-bit Linux operating system in a core i7-3770k processor.

Different scenarios were created by modifying the parameters $\mathcal{K}$ and $\mathcal{L}$, the sample factors for the compressed suffix array and for the data structure proposed as proposed in [5], respectively. The parameters change the space/time requirements over the inputs. Hence, there is a time-space trade-off within the indices.

- The first scenario sets $\mathcal{K} = 10$ and $\mathcal{L} = 8$ and it is the worst space-efficient scenario; therefore, it is the fastest one.
- The second scenario has an intermediary trade-off between time and space, the parameters were set as $\mathcal{K} = 20$ and $\mathcal{L} = 16$.
- The last scenario, with parameters $\mathcal{K} = 20$ and $\mathcal{L} = 32$, is the most space-efficient and therefore the slowest one.

The parameter $\mathcal{B} = 60$ was chosen empirically and it was fixed in all scenarios because it represented a good trade-off between memory peak and building time.

The first subject of interest for the analysis is the time required for the construction of the data structures. While the raw data structure needs an almost negligible building time, requiring only few seconds, the proposed one and the SuDS $\mathcal{CST}$ require a time in the order of thousand of seconds, to build the index in a compressed way. See building times for the different scenarios in the Figures 8, 9 and 10.

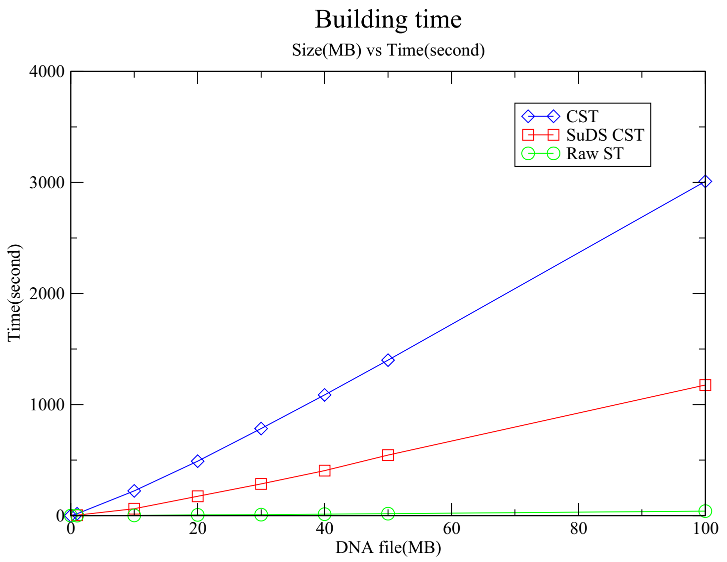

Figure 9. Building time for $\mathcal{K}=20$ and $\mathcal{L}=16$.

Comparing the compressed indices, the proposed implementation has a factor of $\approx 2\times$ over the SuDS index in the fastest scenario, as given in the Figure 8. this is a reasonable factor since the memory peak of the proposed implementation is much lower. In the slowest scenario, given in the Figure 10, the factor is of $\approx 2.5\times$ over the SuDS index.

Despite of being an important resource, the building time is not crucial for applications for which the index needs to be built only once. A typical example is the problem of mapping fragments of DNA to a reference genome. In this application, the index for the genome needs to be built only once in order to map several billions of fragments.

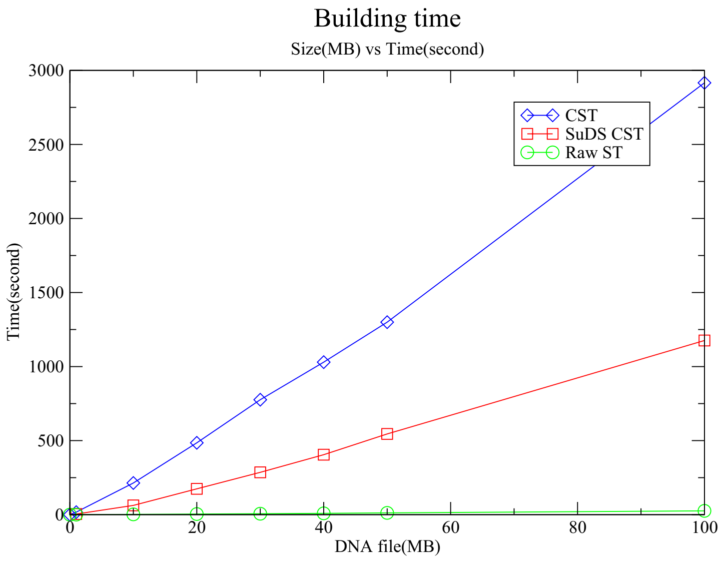

Figure 10. Building time for $\mathcal{K}=20$ and $\mathcal{L}=32$.

The high building time is payed off with low space usage after the structure is built, and with low memory peak usage. This can be observed in the Figures 11 , 12 and 13. Also, it has been shown that the proposed index does not use much more space in its construction than the space required to represent the final structure. The main reason for this, is the use of the incremental algorithm introduced by [15], which allows the construction of the compressed suffix array within $\Theta(nH_0)$ bits.

In comparison with the SuDS index, the proposed implementation is highly competitive in practice when considering the memory peak. On the one hand, the memory peak of the SuDS index has a factor of $\approx 4.5\times$ over the input. Hence, for a 100MB DNA text, for instance, one would require 450MB of main memory in the construction of this structure. On the other hand, the proposed implementation has a low peak memory. The fastest variant, whose space usage is depicted in the Figure 11, needs a factor of $\approx 2.5\times$ over the input whereas the slowest variants, in the Figures 12 and 13 need only factors of $\approx 2\times$ and $\approx 1.7\times$ over the input, respectively. The raw index requires a huge amount of peak memory during its construction compared to the compressed indices, requiring factors above $12\times$ over the input in all scenarios.

Considering the space of the final structure, the fastest variant of the proposed implementation, showed in the Figure 11 consumes $\approx 19$ bits per input symbol, which is slightly more than the requirement from the SuDS index, which is $\approx 14$

bits per input symbol. Considering the slowest variants, shown in Figures 12 and 13, the proposed implementation uses only ≈ 14 and ≈ 13 bits per input symbol respectively, beating the space required by SuDS index. The raw index requires ≈ 80 bits per input symbol in the scenarios, which represents a huge demand of space in comparison with the compressed indices.

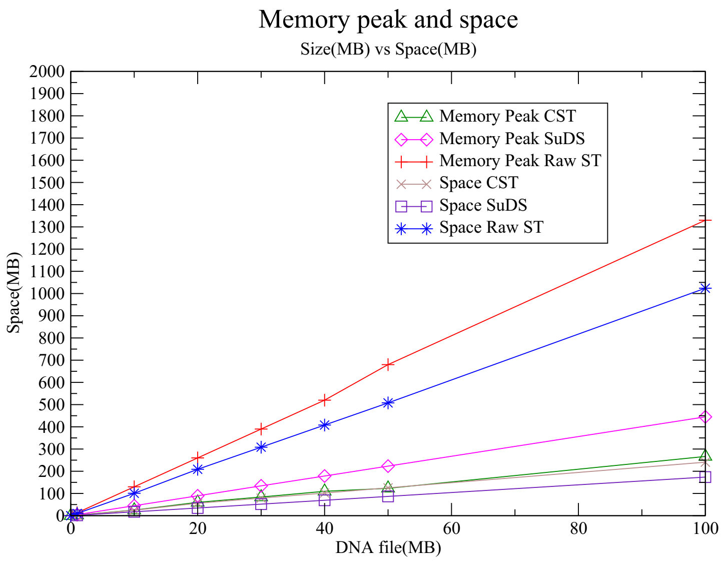

Figure 11. Space for $\mathcal{K}=10$ and $\mathcal{L}=8$.

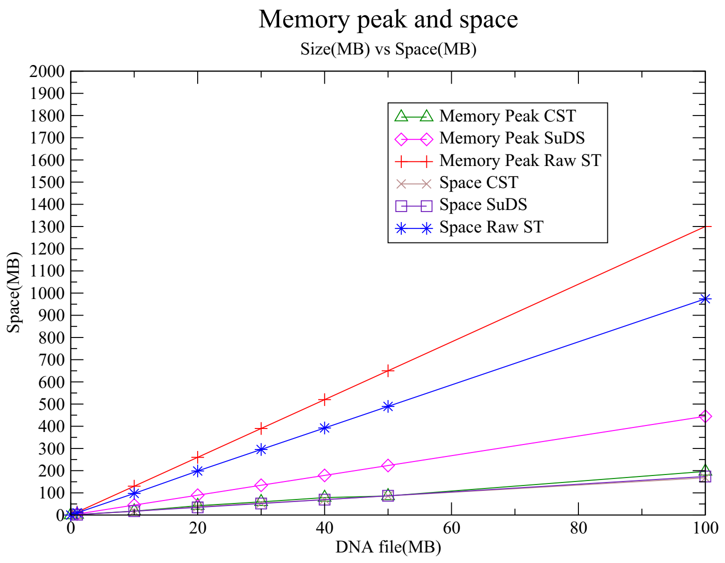

Figure 12. Space for $\mathcal{K}=20$ and $\mathcal{L}=16$.

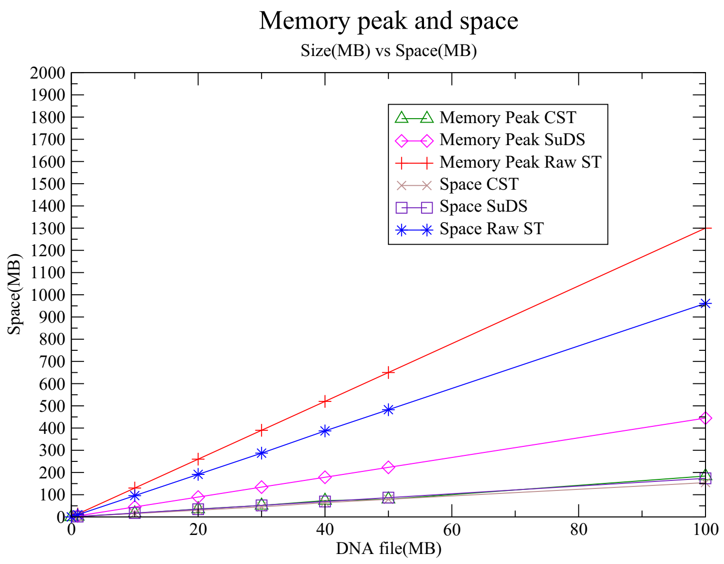

Figure 13. Space for $\mathcal{K} = 20$, $\mathcal{L} = 32$.

Figures 14, 15 and 16 relate the running time of several critical operations for the compressed data structure and the raw one. The comparison was done between these two indices only, because they are based on the same types of queries ($PSV$, $NSV$ and $RMQ$ queries), while the SuDS index is based on a succinct balanced parenthesis representation to execute the operations [31].

The $LCA$ queries were executed between random leaves of the trees. The other queries were executed at random positions. When the $\mathcal{CST}$ occupies more space, the running time of the operations is smaller than when it occupies less space. While in the fastest variant all operations run under microseconds using $\approx 19$ bits per input symbol, as shown in Figure 14, in the slowest variant more complex operations, such as $RMQ$ and $LCA$, run in few milliseconds using only $\approx 13$ bits per input symbol, as illustrated in the Figure 16.

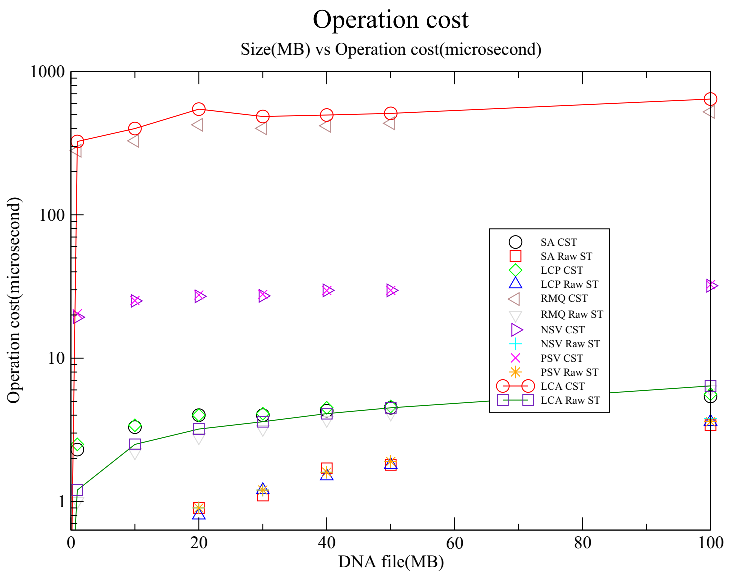

Figure 14. $\mathcal{ST}$ operations for $\mathcal{K}=10$, $\mathcal{L}=8$.

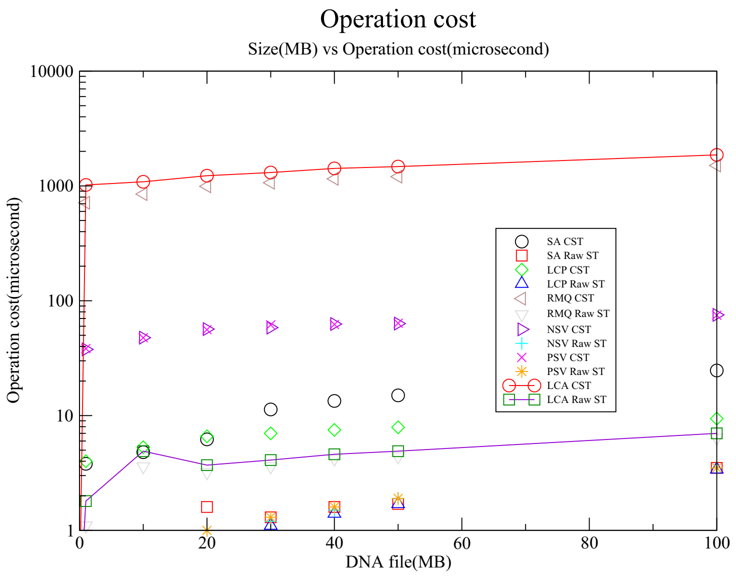

Figure 15. $\mathcal{ST}$ operations for $\mathcal{K}=20$, $\mathcal{L}=16$.

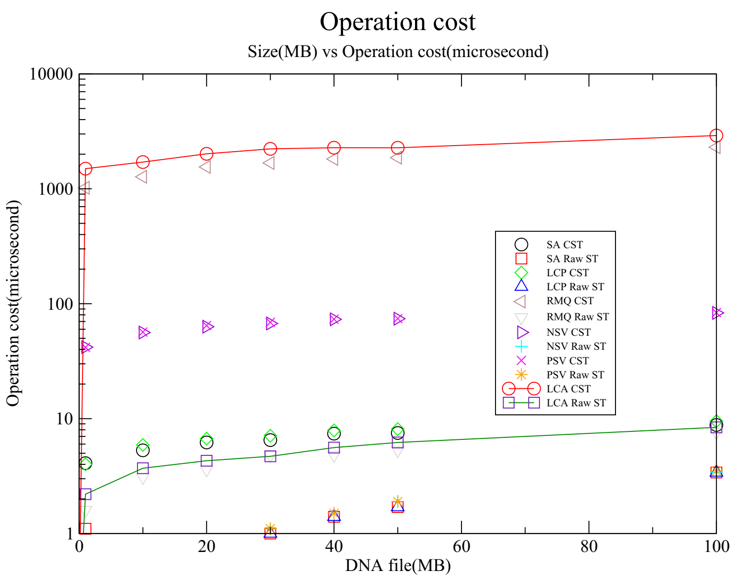

Figure 16. $\mathcal{ST}$ operations for $\mathcal{K}=20$, $\mathcal{L}=32$

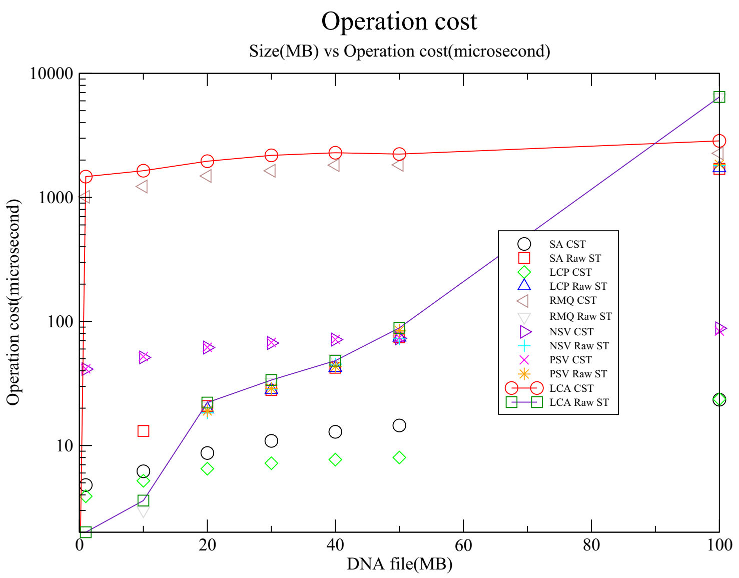

Figure 17. $\mathcal{ST}$ operations with $\mathcal{K} = 20$ and $\mathcal{L} = 32$ when few memory is available.

Additional experiments have confirmed that when enough memory is not available the operation's cost for the raw structure is completely degraded, giving rise to running times greater than the necessary for the same operations with the compressed index. This happens because the raw structure does not fit entirely into main memory, as the compressed does, and disk access is required in order to recover the necessary pages. This scenario is illustrated in the Figure 17. Since the implementation does not handle huge texts yet, the environment was configured to use only 580MB of RAM memory.

## 7 Conclusion and Future Work

A compressed suffix tree index, $\mathcal{CST}$, with very low memory peak usage was proposed and implemented. The structure offers a good time and space trade-off and was shown useful for dealing with non trivial operations such as $LCA$ computations.

Running time and memory usage were analyzed regarding a raw $\mathcal{ST}$ implementation based on uncompressed suffix arrays and $LCP$ information and the SuDS $\mathcal{CST}$ [33].

The proposed $\mathcal{CST}$ needs only $\approx 13$ bits per symbol in one of the variants, that is much less than the $\approx 80$ bits per symbol that the raw structure requires. This represents a huge economy in space regarding other approaches, which allows manipulation of large texts when memory availability is critical. The current implementation also shows to be competitive and efficient in practice regarding the use of space if compared to state-of-the-art $\mathcal{CST}$ implementations such as the SuDS $\mathcal{CST}$. Actually, the peak memory of the proposed implementation is much lower than the one of the SuDS index and the final space to represent the structure is very competitive in space-efficient variants of the proposed implementation with respect to the one of the SuDS $\mathcal{CST}$.

Additional comparisons with other $\mathcal{CST}$ implementations, such as the ones given in [5] and [27] are proposed as future work. Performing these comparisons will allow to identify precisely the practical capabilities of the proposed index. Experiments with different alphabets than the one of DNA sequences, such as the ones of proteins and natural languages, shall be done to measure the capability of compression of the structure.

Further improvements of the structure will be possible through a faster and more space-efficient mechanism for the representation and use of the $\Psi$ function and the compressed suffix array information, as for instance those solutions given in [16]. One of these solutions is to adopt the FM-Index, introduced in [8], as substitute for the compressed suffix array, which is of great interest since it can outperform the second structure with respect to time and space in some scenarios. The investigation of the RLFM index is also interesting[22].

Other possible solution is to encode the $\Psi$ function using Elias gamma code instead of the Rice code, since the former can be faster and more space-efficient than the latter for small alphabets such as DNA or proteins. Such mechanisms would allow a better space and time usage, which can improve the proposed implementation of the incremental algorithm introduced by [15], since the implementation is worse in practice than the other implementations of the same algorithm such as the ones in the Bioinformatics tools BWT [20] and BWA [21].

Improvements in the execution of *Rank* and *Select* queries using the methodology from [26], which has been shown faster than the method used in our approach in [11], would speed up the operations running time and building time of the proposed implementation, as all them are dependent on the time required by these queries.

The adoption of other mechanisms to build the $LCP$ information, such as those proposed in [17],[9] and [4] can improve the building time and query time for this information, which would indirectly speed up the $NSV$, $PSV$ and $RMQ$ queries and tree traversals operations, since they are based on $LCP$ access.

Also, to make the structure of practical usefulness for applications with huge texts, it should be ported to a 64-bit architecture.

## References

[1] Abouelhoda, M., S. Kurtz and E. Ohlebusch, *The Enhanced Suffix Array and Its Applications to Genome Analysis*, in: *Workshop on Algorithms in Bioinformatics*, Lecture Notes in Computer Science **2452** (2002), pp. 449–463.

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

[3] Apostolico, A., *The myriad virtues of subword trees*, in: A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, NATO ASI Series F, Computer and System Sciences **12**, Springer Verlag, 1985 pp. 85–96.

[4] Beller, T., S. Gog, E. Ohlebusch and T. Schnattinger, *Computing the longest common prefix array based on the Burrows-Wheeler transform*, in: *String Processing and Information Retrieval*, Lecture Notes in Computer Science **7024** (2011), pp. 197–208.

[5] Cánovas, R. and G. Navarro, *Practical compressed suffix trees*, in: *Proceedings of the 9th international conference on Experimental Algorithms*, Lecture Notes in Computer Science **6049** (2010), pp. 94–105.

[6] Farach, M., *Optimal suffix tree construction with large alphabets*, in: *Proceedings 38th Annual Symposium on Foundations of Computer Science FOCS*, IEEE (1997), pp. 137–143.

[7] Ferragina, P. and G. Manzini, *Opportunistic data structures with applications*, in: *Proceedings 41st Annual Symposium on Foundations of Computer Science FOCS*, IEEE (2000), pp. 390–398.

[8] Ferragina, P. and G. Manzini, *Indexing compressed text*, Journal of the Association for Computing Machinery **52** (2005), pp. 552–581.

[9] Fischer, J., *Inducing the LCP-array*, in: *12th International Symposium on Algorithms and Data Structures WADS*, Lecture Notes in Computer Science **6844** (2011), pp. 374–385.

[10] Fischer, J., V. Mäkinen and G. Navarro, *An (other) entropy-bounded compressed suffix tree*, in: *19th Annual Symposium Combinatorial Pattern Matching CPM*, Lecture Notes in Computer Science **5029** (2008), pp. 152–165.

[11] González, R., S. Grabowski, V. Mäkinen and G. Navarro, *Practical implementation of rank and select queries*, Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms WEA (2005), 27–38 pp.

[12] Grossi, R. and J. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, SIAM Journal on Computing **35** (2005), pp. 378–407.

[13] Grossi, R. and J. S. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract)*, in: F. F. Yao and E. M. Luks, editors, *Proceedings of the thirty-second annual ACM symposium on Theory of computing* (2000), pp. 397–406.

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

[15] Hon, W., T. Lam, K. Sadakane, W. Sung and S. Yiu, *A space and time efficient algorithm for constructing compressed suffix arrays*, Algorithmica **48** (2007), pp. 23–36.

[16] Hon, W., T. Lam, W. Sung, W. Tse, C. Wong and S. Yiu, *Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences*, in: *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments* (2004), pp. 31–38.

[17] Kärkkäinen, J., G. Manzini and S. Puglisi, *Permuted longest-common-prefix array*, in: *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching CPM*, Lecture Notes in Computer Science **5577**, Springer (2009), pp. 181–192.

[18] Kasai, T., G. Lee, H. Arimura, S. Arikawa and K. Park, *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*, in: A. Amir and G. M. Landau, editors, *12th Annual Symposium on Combinatorial Pattern Matching CPM*, Lecture Notes in Computer Science **2089** (2001), pp. 181–192.

[19] Kurtz, S., *Reducing the space requirement of suffix trees*, Software Practice and Experience **29** (1999), pp. 1149–1171.

[20] Lam, T., W. Sung, S. Tam, C. Wong and S. Yiu, *Compressed indexing and local alignment of DNA*, Bioinformatics **24** (2008), pp. 791–797.

[21] Li, H. and R. Durbin, *Fast and accurate short read alignment with Burrows–Wheeler transform*, Bioinformatics **25** (2009), pp. 1754–1760.

[22] Mäkinen, V. and G. Navarro, *Succinct suffix arrays based on run-length encoding*, Nordic Journal of Computing **12** (2005), pp. 40–66.

[23] Manber, U. and G. Myers, *Suffix arrays: a new method for on-line string searches*, in: *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics (1990), pp. 319–327.

[24] McCreight, E., *A space-economical suffix tree construction algorithm*, Journal of the Association for Computing Machinery **23** (1976), pp. 262–272.

[25] Mori, Y., *Libdivsufsort: a lightweight suffix sorting library*, Available in: `https://code.google.com/p/libdivsufsort/` (2007).

[26] Navarro, G. and E. Providel, *Fast, small, simple rank/select on bitmaps*, in: *Symposium on Experimental Algorithms*, Lecture Notes in Computer Science **7276** (2012), pp. 295–306.

[27] Ohlebusch, E., J. Fischer and S. Gog, *CST++*, in: *Proceedings of the 17th international conference on String processing and information retrieval*, Lecture Notes in Computer Science **6393** (2010), pp. 309–321.

[28] Puglisi, S. J., W. F. Smyth and A. H. Turpin, *A taxonomy of suffix array construction algorithms*, ACM Comput. Surv. **39** (2007).

[29] Sadakane, K., *Compressed text databases with efficient query algorithms based on the compressed suffix array*, Algorithms and Computation **1969** (2000), pp. 295–321.

[30] Sadakane, K., *Succinct representations of LCP information and improvements in the compressed suffix arrays*, in: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics (2002), pp. 225–232.

[31] Sadakane, K., *Compressed suffix trees with full functionality*, Theory of Computing Systems **41** (2007), pp. 589–607.

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

[33] V. Mäkinen, N. Välimäki, J. Siren and S. Kazi, *SuDS group website*, Available in: `http://www.cs.helsinki.fi/group/suds/cst/` (2009).

[34] Välimäki, N., V. Mäkinen, W. Gerlach and K. Dixit, *Engineering a compressed suffix tree implementation*, ACM Journal of Experimental Algorithmics **14** (2009).

[35] Weiner, P., *Linear pattern matching algorithms*, in: *14th Annual Symposium on Switching and Automata Theory SWAT (FOCS)*, IEEE (1973), pp. 1–11.