On Parallel Transformations of Suffix Arrays into Suffix Trees

Costas Iliopoulos
Department of Computer Science
Kings College, London, UK
csi@dcs.kcl.ac.uk

Wojciech Rytter
Department of Computer Science
New Jersey Institute of Technology, USA
rytter@oak.njit.edu

Abstract

Two simple algorithms for parallel construction of suffix trees via suffix arrays are discussed in the paper. This is motivated by the fact that several simple interesting sequential/parallel constructions of suffix arrays have been designed recently, see [9, 12, 11, 3, 4]. It is known that the suffix array can be easily transformed into the suffix tree in linear sequential *time*, see [10, 2]. In this paper we show that very simple parallel transformations of suffix arrays to suffix trees are also possible. The *parallel complexity* of these transformations is similar to the *parallel complexity* of constructing suffix arrays. Our both algorithms work for integer alphabets, they use only $O(n)$ space and have total work $O(n \log n)$. Our first algorithm results as a very simple application of the parallel *divide-and-conquer*, it works in $O(\log^2 n)$ *time*. The core of the paper is the algorithm BUILD2. The main result is the design of an algorithm which is faster and works in $O(\log n)$ parallel *time*. This is an improvement on the previously known algorithms which use super-linear space or work in $O(\log^4 n)$ time. The basic model of parallel computing used in this paper is the CREW PRAM, see [5]. The advantage of our algorithms is their simplicity.

1 Introduction

The suffix tree is a fundamental data structure in text algorithms, see [1, 6, 2]. There are several complex classical algorithms for a sequential suffix tree construction, [17, 16, 15, 3, 4]. The parallel algorithms are even more complex, [5, 13, 8]. The algorithm of [8] has $O(n \log n)$ total work but uses super-linear space $\Theta(n^{1+\epsilon})$. On the contrast, our algorithms use only $O(n)$ space. The most efficient $O(n)$ space algorithm of [7] is rather complicated, it works in $O(\log^4 n)$ time and has $O(n \log n)$ work for general alphabets. Our second algorithm

is faster than one from [7], it works in $O(\log n)$ time and $O(n)$ space with $O(n \log n)$ total work for general alphabets if the suffix array and the array LCP are already computed. They can be computed in a simple way in $O(log^2 n)$ time with $O(n)$ processors.
We assume that the alphabet is $[1 \ldots n]$.

It was known that suffix trees are closely related to suffix arrays, much simpler but very useful data structure introduced by [14]. Until recently there was no simple linear time algorithm to produce suffix arrays. In 2003, three groups of people discovered three different algorithms, the simplest of them is the algorithm of Karkkainen and Sanders, which is a simplified version of Farach's algorithm [3, 4]. Karkkainen and Sanders described a simple linear time sequential algorithm for suffix arrays as well as its parallel versions. In this paper we extend it to the parallel computation of suffix trees. The corresponding algorithms are extremely simple. The first is a kind of a naive divide-and-conquer algorithm. The second one is more efficient and more interesting. It gives also a new linear time transformation of suffix arrays to suffix trees. The basic subroutine in the algorithm is the parallel computation of the table of *Nearest Smaller Neighbors*. It is easy to compute this table in logarithmic time and $O(n \log n)$ work. It is also easy to do it sequentially in linear time.

Assume we are given a string x of length n, assume also that we append to x a special end-marker symbol $x_{n+1} = \#$, which is smaller than any other symbol of x. The suffix array SUF $= [i_1, i_2, \ldots, i_n]$ is the permutation which gives the sorted list of all suffixes of the string x. This means that

$$suf(i_1) < suf(i_2) < suf(i_3) < \ldots < suf(i_n),$$

where $<$ means here lexicographic order, and $suf(i_k) = x[i_k \ldots n]\,\#$. In other words $suf(i_k)$ is the k-th suffix of x in sense of lexicographic order. There is another useful table of the length of adjacent common prefixes: LCP$[k]$ is the length of the longest common prefix of $suf(i_k)$ and $suf(i_{k-1})$, for $1 < i < n$. Define LCP$[1] = 0$.

Example. Let us consider an example string:

$$x \;=\; a\; b\; a\; a\; b\; a\; a\; b\; a\; b\; a\; a\;\; \#.$$

For this string we have:

$$\text{LCP} \;=\; [\;\; 0 \;\; 1 \;\; 2 \;\; 4 \;\; 1 \;\; 4 \;\; 6 \;\; 3 \;\; 0 \;\; 3 \;\; 5 \;\; 2 \;\;]$$

$$\text{SUF} \;=\; [\;\; 12 \;\; 11 \;\; 3 \;\; 6 \;\; 9 \;\; 1 \;\; 4 \;\; 7 \;\; 10 \;\; 2 \;\; 5 \;\; 8 \;\;]$$

The following fact has been shown using a simple parallel algorithm.

Theorem 1 [9] *The arrays SUF, LCP can be constructed in* $O(\log^2 n)$ *time and* $O(n \log n)$ *work on an EREW PRAM.*

2 A very simple $O(\log^2 n)$-time construction

Our first algorithm is a very simple application of the parallel *divide-and-conquer* approach. Its correctness is based on the following fact:

Observation. Assume $i < j < k$, then the lowest common ancestor in the suffix tree of leaves $SUF[i]$ and $SUF[j]$ is on the branch from the root to the leaf $SUF[k]$. Hence for a fixed k all lowest common ancestors of $SUF[i]$ and $SUF[j]$ for all $i < k < j$ are on a same branch.

In our example, see Figure 1.b, the sequence corresponding to the suffix array is $\alpha = [i_1, i_2, \dots i_n] = [12, 11, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8]$. Let us consider $SUF[6] = 1$, and the branch from the root to 1 is a separator of the suffix tree, it separates the tree into two subtrees with a common branch: the first subtree is for the first half $\alpha 1 = [12, 11, 3, 6, 9, 1]$ and the second subtree for $\alpha 2 = [1, 4, 7, 10, 2, 5, 8]$. The sizes of "halves" differ at most by one.

The input to the function is a subsegment α of the sequence $[i_1, i_2, \dots i_n]$ of indices of suffixes of x according to the suffix array. The suffix trees are built for the subsegments of of $suf(i_1), suf(i_2), \dots suf(i_n)$. α is split into two parts $\alpha 1$, $\alpha 2$ of approximately same size. The suffix trees $T1$ and $T2$ are constructed in parallel for the left and the right parts. The last element of $\alpha 1$ is the same as of $\alpha 2$. Hence the resulting suffix tree results by merging together the rightmost branch of $T1$ with the leftmost branch of $T2$. The algorithm is written as a recursive function $BUILD1$.

```
Algorithm BUILD1($\alpha$); { Given SUF = $[i_1, i_2, \dots i_n]$}
    { $\alpha = (s_1, s_2, \dots s_k)$ is a subsegment of $[i_1, i_2, \dots i_n]$}
    if $|\alpha| \le 2$ then compute the suffix tree $T$ sequentially in $O(1)$ time;
    Comment: the table LCP is needed for $O(1)$ time computation in this step
     else
    in parallel do
        {$T1$ := BUILD1($s_1, s_2, \dots s_{k/2}$); $T2$ := BUILD1($s_{k/2}, s_{k/2+1}, \dots s_k$)};
    Merge the rightmost branch of T1 with the leftmost branch of $T2$;
  return the resulting suffix tree $T$;
```

The history of the algorithm for our example string is shown in Figure 1.

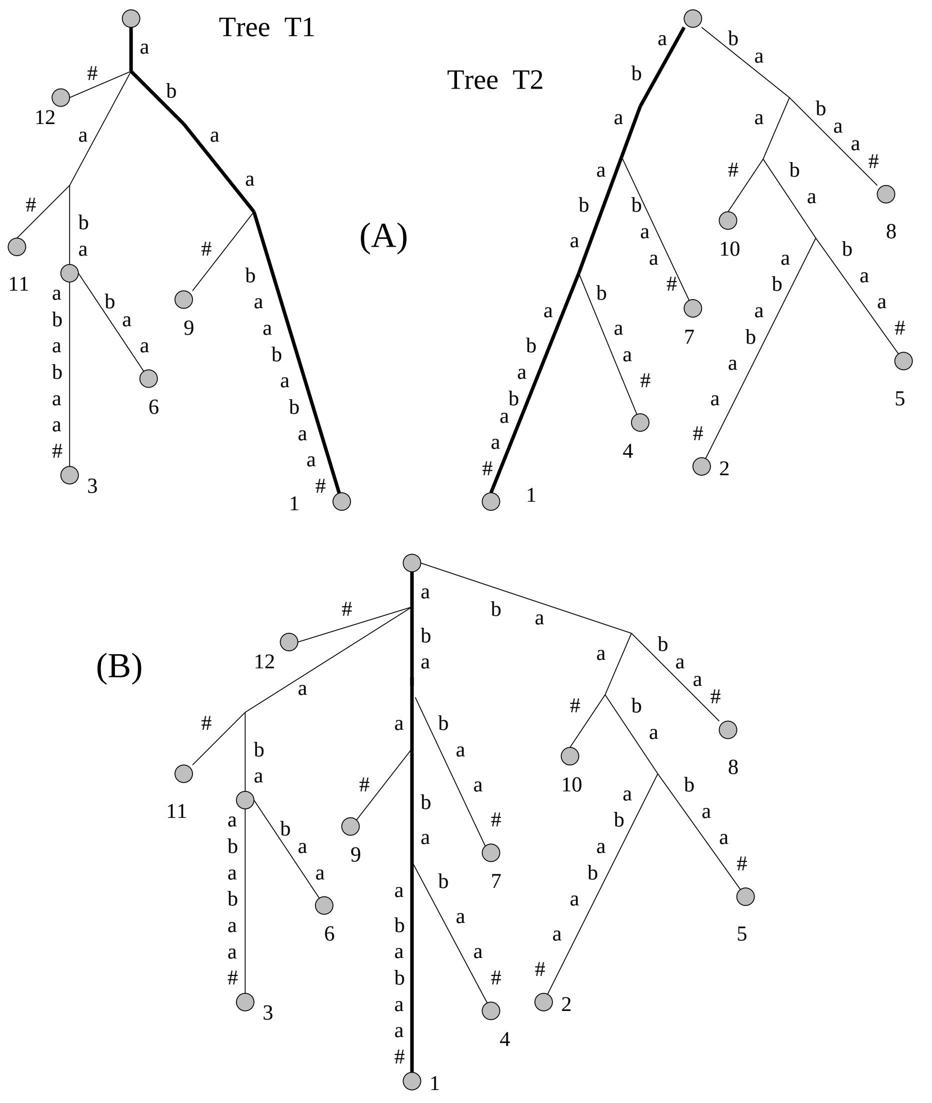

Figure 1: **(A)** The suffix trees for two "halves" of the sorted sequence of suffixes. The "middle" suffix (in bold) belongs to both parts. **(B)** The suffix tree resulting after merging the branches corresponding to the common suffix $suf(1)$, belonging to both parts from Figure 1. The final suffix tree T results from the subtrees $T1$, $T2$ built for sequences of suffixes corresponding to sequences of indices (starting positions) (12, 11, 3, 6, 9, 1) and (1, 4, 7, 10, 2, 5, 8).

Theorem 2 *Assume that the arrays SUF, LCP are given for an input string of size n. Then the algorithm BUILD1 computes the suffix tree for x in $O(\log^2 n)$ time and $O(n)$ space with $O(n \log n)$ total work.*

Proof: If $|\alpha| = 2$a and we have only two suffixes $suf(i_{j-1})$, $suf(i_j)$, for some j, then we can create for them a partial suffix tree in $O(1)$ time as follows. The tree has a root, two leaves corresponding to whole suffixes $suf(i_{j-1})$, $suf(i_j)$ and one internal node corresponding to the longest common prefix of $suf(i_{j-1})$, $suf(i_j)$. The length of this prefix is given by LCP[j].

The main operation is that of *merging* two branches corresponding to a same suffix. The *total depth* of a node is the length of a string corresponding to this node in the suffix tree. The nodes on the branches are sorted in sense of their total depth. Hence to merge two branches we can do a parallel merge of two sorted arrays of size n.

This can be done in logarithmic time with linear work, see [5]. The total time is $O(\log^2 n)$ since the depth of the recursion of the function $BUILD1(\alpha)$ is logarithmic.

The total work satisfies similar recurrence as the complexity of the merge-sort:

$$Work(n) \ = \ Work(n/2) + Work(n/2+1) + O(n).$$

The solution is a function of order $O(n \log n)$. This complete the proof. □

3 A simple $O(\log n)$-time construction

The second algorithm is non-recursive. The leaf nodes are well defined, they correspond to starting positions of suffixes. The main point is a proper interpretation of internal nodes of the suffix tree. denote by LCA the lowest common ancestor function. We use the following obvious fact.

Observation Each internal node is a lowest common ancestor of two consecutive leaves. It is possible that it is a lowest common ancestor for two different pairs of consecutive leaves. Let i_{k-1}, i_k be two consecutive leaves. Then $LCA(i_{k-1}, i_k)$ is the node on the branch from the root to i_k, the string corresponding to the path from root to v is of length $LCP[k]$.

Assume that the sequence of leaves $(i_1, i_2, \ldots, i_n)$ read from left to right corresponds to the lexicographically ordered sequence of suffixes. For an internal vertex v define

$$repr(v) \ = \ \text{the leftmost leaf } i_k \text{ such that } LCA(i_k, i_{k-1}) = v.$$

We identify each internal node v with a pair $(i_k, LCP[k])$, where $i_k = repr(v)$, see Figure 2.

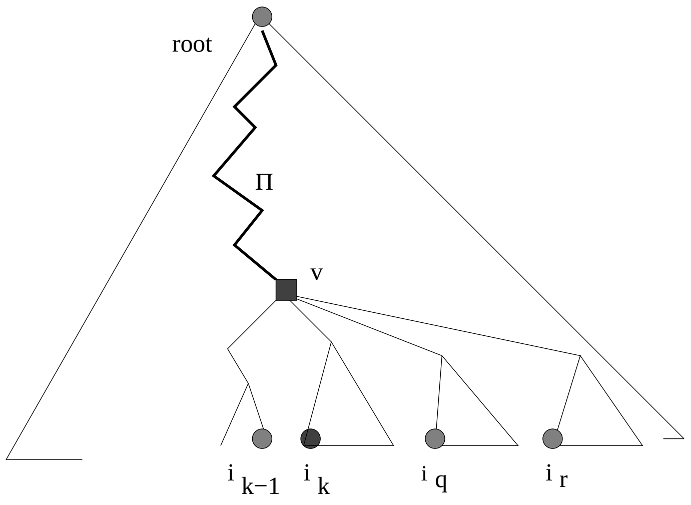

Figure 2: The node v is identified with $(i_k, LCP[k])$, $repr(v) = i_k$ and the length of the string Π corresponding to path from *root* to v equals $LCP[k]$. Observe that $Leftmost[r] = Leftmost[q] = Leftmost[k] = k$.

We introduce two tables: the *Nearest Smaller Neighbor* table NS and the table $Leftmost[1 \ldots n]$. For all $i > 1$ such that $LCP[i] > 0$ define:

$$SN[i] \quad = \quad \begin{cases} L[i], \text{ if } LCP[L[i]] \geq LCP[R[i]] \\ R[i] \text{ otherwise} \end{cases}$$

where

$$L[i] = \max\{j < i : LCP[j] < LCP[i]\} \quad R[i] = \min\{j > i : LCP[j] < LCP[i]\}$$

The table $Leftmost$ is defined for each $1 \leq i \leq n$ as follows:

$$Leftmost[i] \quad = \quad \min\{ \ 1 \leq j \leq i \ : \ LCP[j] = LCP[i] \text{ and } LCP[j] = 0 \text{ or } SN[i] < j \ \}$$

The tables are illustrated below for our example string

Lemma 1 *Assume $(i_k, LCP[k])$ is an internal non-root node and i_k is an external node of the suffix tree, then*
(a) *The father of $(i_k, LCP[k])$ is $(i_j, LCP[j])$, where $j = Lefmost(SN(k))$.*
(b) *If $LCP[k] \leq LCP[k+1]$ then the father of i_k is $(i_j, LCP[j])$ where $j = Leftmost[k]$, otherwise the father of i_k is $(i_{k+1}, LCP[k+1])$.*

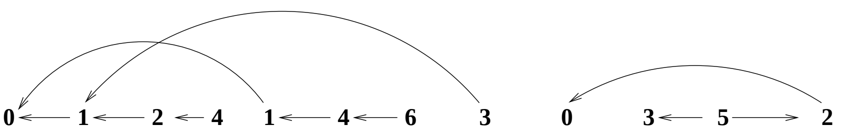

Figure 3: The table SN.

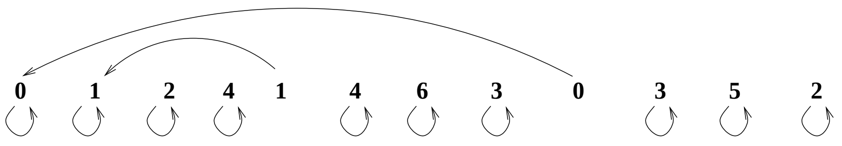

Figure 4: The table $Leftmost$.

Lemma 2 *The tables SN and $Leftmost$ can be computed on a CREW PRAM in $O(\log n)$ time with n processors.*

Proof: We show how to compute the table L within claimed complexities. Assume w.l.o.g.. that n is a power of 2 and construct complete regular binary tree R of height $\log n$ over positions 1 .. n. Using a classical algorithm for parallel prefix computation we compute for each node v of R the minimum of LCP over the interval corresponding to leaf-positions of the subtree rooted at v. Then for each position i we compute $L[i]$ by first traversing up in R from the i-th leaf until we hit a node v whose value is smaller then $LCP[i]$, and then we go down to the position $L[i]$. $\square$

Theorem 3 *Assume that the arrays SUF, LCP are given for an input string of size n. The algorithm BUILD2 computes the suffix tree for x on a CREW PRAM in $O(\log n)$ time with $O(n \log n)$ work and $O(n)$ space.*

Proof: The algorithm $BUILD2$ is presented informally below. Once the table SN is computed all other computations are computed independently *locally* in $O(\log n)$ time with n processors. The main idea in the construction is to identify the internal nodes of the suffix tree with the pairs $(i_k, LCP[k])$. The internal node (j,p) corresponds to the subword $x[j \dots j+p-1]$ and the external node k corresponds to the suffix $x[k \dots n]$.

```
Algorithm BUILD2($x$);
    compute in parallel the tables $SN$ and $Leftmost$;
    create external nodes $1, 2, \ldots, n$;

    for each $k \in [1 \ldots n]$
        create internal node $(i_k, LCP[k])$ if $Leftmost[k] = k$;

    compute father links using formula from Lemma 1

    for each internal node $(i_k, r)$, where $r > 0$ do in parallel
        $(i_s, p) \; = \; father(i_k, r)$;
        the edge $(i_s, p) \Rightarrow (i_k, r)$ gets label $x[i_k + p \ldots i_k + r]$;

    for each external node $i_k$ do in parallel
        $(i_s, p) \; = \; father(i_k 0$;
        the edge $(i_s, p) \Rightarrow (i_k)$ gets label $x[i_k + p \ldots n]$;
```

□

4 Conclusions

We have given two constructions for the parallel transformation of suffix arrays into suffix trees.

Th second construction gives also a simple linear time (independent on the size of the alphabet for integer alphabets) sequential algorithm for the suffix tree construction due to the following well known fact:

Lemma 3 *The table SN can be computed in $O(n)$ sequential time.*

References

[1] Apostolico, A., The myriad virtues of suffix trees, in: Apostolico, A., and Galil, Z., editors, Combinatorial Algorithms on Words, NATO Advanced Science Institutes, Series F, vol 12, Springer-Verlag, Berlin, 1985.

[2] Crochemore, M., and Rytter, W., Jewels of stringology: text algorithms, World Sceintific (2003)

[3] Farach, M., Optimal suffix tree construction with large alphabets, FOCS 1997.

[4] Farach, M. , Ferragina, P., and Muthukrishnan, S. On the sorting complexity of suffix tree construction, Journal of the ACM, vol. 47(6), 987–1011, 2000.

[5] A.Gibbons, W.Rytter, Efficient parallel algorithms, Cambridge University Press 1988

[6] R. Grossi, G. Italiano, Suffix trees and their applications in string algorithms, Techn. Report CS-96-14, Universita di Venezia, 1996

[7] R. Hariharan: Optimal parallel suffix tree construction. STOC 1994: 290-299

[8] Iliopoulos, C., Landau, G.M., Schieber, B., and Vishkin, U., Parallel construction of a suffix tree with applications, Algorithmica 3 (1988): 347–365.

[9] J. Karkkainen, P. Sanders, Simple linear work suffix array construction, ICALP 2003

[10] Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K., Linear-time longest-common-prefix computation in suffix arrays and its applications, in: (Proc. 12th Combinatorial Pattern Matching) 181–192.

[11] D.K.Kim,J.Sim, H.Park, K.Park, Linear time construction of suffix arrays, Combinatorial Pattern Matching 2003

[12] P.Ko, S.Aluru, Space efficient linear time construction of suffix arrays, Combinatorial Pattern Matching 2003

[13] Landau, G.M., Schieber, B., and Vishkin, U., Parallel construction of a suffix tree, in: (Automata, Languages and Programming, Lecture Notes in Computer Science 267, Springer-Verlag, Berlin, 1987): 314–325.

[14] Manber, U., and Myers, E., Suffix arrays: A new method for on-line string searches, in: (Proc. of Ist ACM-SIAM Symposium on Discrete Algorithms, American Mathematical Society, Providence, R.I., 1990): 319–327.

[15] McCreight, E.M., A space-economical suffix tree construction algorithm, J. ACM 23, 2 (1976): 262–272.

[16] Ukkonen, E., Constructing suffix trees on-line in linear time, in: (IFIP'92): 484–492.

[17] Weiner, P., Linear pattern matching algorithms, in: (Proc. 14th IEEE Annual Symposium on Switching and Automata Theory, Washington, DC, 1973): 1–11.

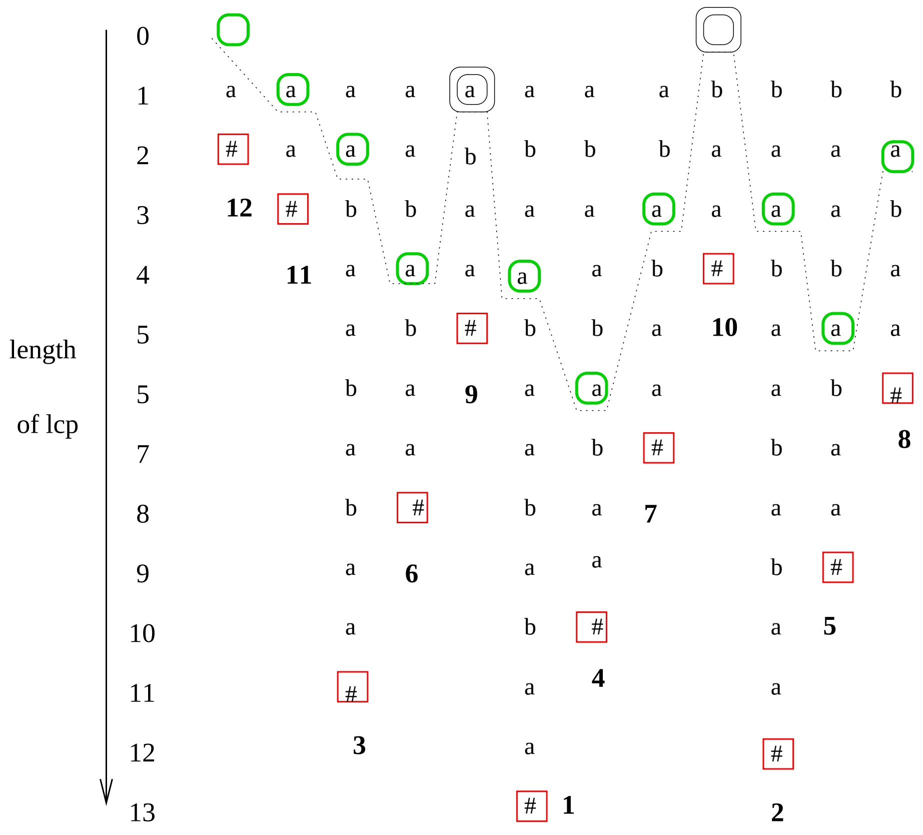

Figure 5: The lexicographically sorted list of suffixes, each suffix listed vertically top-down. The circled nodes show the value of LCP and correspond to internal nodes of the suffix tree. The square nodes correspond to the whole suffixes, the starting position of the suffix is shown at each square node, they correspond also to external nodes of the suffix tree. The "shape" of LCP values (circled nodes), gives the shape of the suffix tree. Two circles nodes are dubly circled, these are redundant nodes, they do not appear in the suffix tree since their leftmost representatives are different.

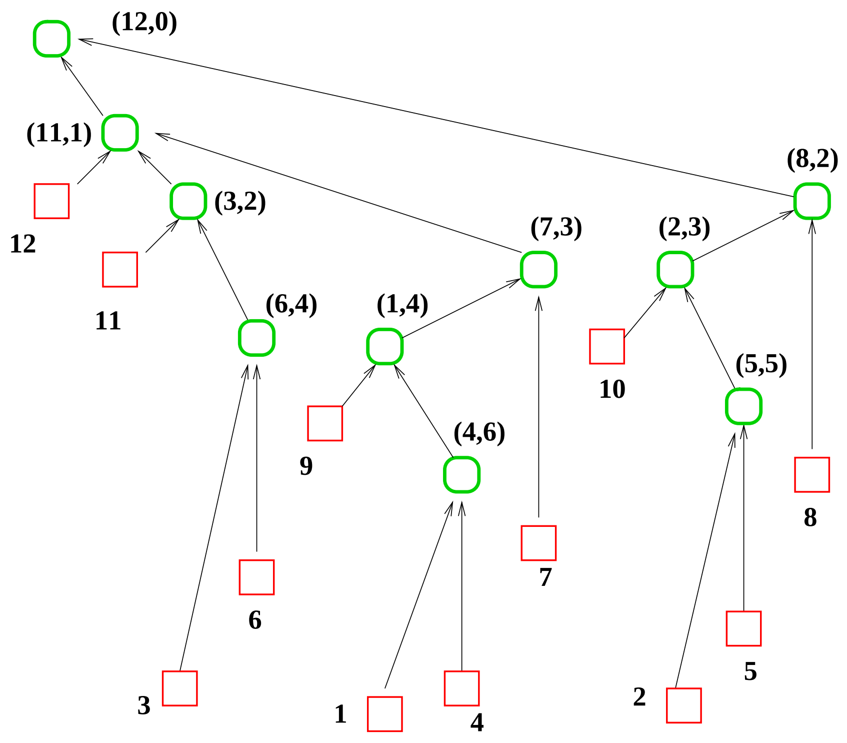

Figure 6: The structure of the suffix tree. Only the links to the fathers are shown. The internal nodes are $(1, LCP[1])$, $(2, LCP[2])$, $\ldots, (i_n, LCP[i_n])$, where $SUF = [i_1, i_2, \ldots, i_n]$. The pair $(i, LCP[i]$ is omitted if $Leftmost[i] < i$. Two such pairs are omitted here, compare with the previous figure.