1986

Parallel Log-time Construction of Suffix Trees

Alberto Apostolico

Costas Iliopoulos

Report Number:
86-632

PARALLEL LOG-TIME CONSTRUCTION OF SUFFIX TREES

Alberto Apostolico

CSD-TR-632
September 1986

Parallel Log-time Construction of Suffix Trees

Alberto Apostolico

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, U.S.A.

Costas Iliopoulos

Purdue University
and
University of London

ABSTRACT

Many string matching applications are based on suffix trees. Linear time sequential algorithms are available for constructing such trees. A CRCW Parallel RAM algorithm is presented here which takes $O(\log n)$ time with n processors, n being the length of the input string. The algorithm requires $\Theta(n^2)$ space, but only $O(n \log n)$ cells need to be initialized.

1. INTRODUCTION

Let x be a string of $n=|x|$ *symbols* from a finite alphabet I and # a special symbol not in I. Given a substring w of x, a *descriptor* of w is any pair $(i,|w|)$ such that i is the starting position in x of an occurrence of w. The *suffix tree* T_x associated with x is the trie (digital search tree) with n leaves and at most $n-1$ internal nodes such that: (1) each arc is labeled with a descriptor of some substring of $x\#$, (2) each leaf is labeled with a position of x and (3) the concatenation of the labels on the path from the root to leaf i describe the suffix of $x\#$ starting at position i (see Fig. 1 for an example). In practice, the label of the arc connecting node μ to its parent node is stored in μ.

For fixed alphabet size, the sequential algorithm in [MC] constructs T_x in linear time. The time bound becomes $O(n\log n)$ if the alphabet size is not a constant. References to alternative equivalent structures and constructions as well as to several applications can be found in [AA]. In the context of parallel computation, various open problems revolve around T_x [GA].

Here we address the problem of constructing T_x in parallel, within the following CRCW PRAM model of computation. We use n processors which can simultaneously read from and write to a common memory with $\Theta(n^2)$ locations. The overall *processors*×*time* cost of our algorithm is $O(n\log n)$, which is optimal for unbounded alphabets. Although the algorithm requires quadratic space, only $O(n\log n)$ locations need initialization. Our approach consists of two main parts. In the first part, an approximate version of the tree is built, called the *skeleton*. This part of the construction is reminiscent of an early approach to subquadratic pattern matching [KMR]. The second part consists of refining the skeleton so as to transform it into T_x.

2. CONSTRUCTING THE SKELETON TREE

From now on, we will assume w.l.o.g. that n is a power of 2. We also extend x by appending to it $n-1$ instances of symbol #. We still use $x\#$ to refer to this modified string. The skeleton D_x of x is a tree with n leaves and each internal node of which has at least two children. The links in D_x point from each node to its parent. Each leaf or internal node of D_x is labeled with the descriptor of some substring of $x\#$ having starting positions in $[1,n]$. If vertex μ is labeled with descriptor (i,l), then $l=2^q$ for some q, $0\leq q\leq \log n$. If μ is a leaf then $l=n$. If μ is an internal node other than the root, then q is the *stagenumber* of μ. If the label of μ corresponds to substring w of x, then we write $w=W(\mu)$, and we call μ the *locus* of w. Figure 2 shows the skeleton for the string of Fig. 1. A constructive definition for D_x is as follows.

(i) The root of D_x is the locus of the empty word. The root has $|I|$ sons, each one being the locus of a distinct symbol of I.

(ii) Assume that all nodes of stagenumber up to $l-1\geq 0$ have been inserted in D_x. To expand D_x to stagenumber $l\leq \log n$, consider the nodes of stagenumber $l-1$ one by one. For the generic such node μ, let $w=W(\mu)$. Now do the following.

1 If $w=z\#$ then make μ the (unique) leaf labeled (i,n), where i is the first component of the old label of μ.

2 If $w\neq z\#$ and there are $k\geq 2$ distinct substrings $s_1,s_2,\ldots,s_k$ of $x\#$ all of which have w as a prefix and such that $|s_t|=2|w|, t=1,2,\ldots,k$, then create k sons of μ, $v_1,v_2,\cdots,v_k$, and make v_t the locus of s_t, $t=1,2,\ldots,k$. Otherwise (i.e., if $w\neq z\#$ but $k=1$), then make μ the locus of s_1.

Observe that no two nodes of D_x can have the same label. The parallel construction of D_x is an easy task. We describe it in detail so as to acquaint the reader with the basic concurrent steps which are used throughout this paper.

We use n processors $p_1,p_2,...,p_n$, where i is the *serial number* of processor p_i. At the beginning, processor p_i is assigned to the i-th position of x, $i=1,2,...,n$. It is convenient to think of each processor as being assigned two segments of the common memory, each segment consisting of $\log n+1$ cells. The segments assigned to p_i are called ID_i, and $NODE_i$, respectively. By the end of the computation, $ID_i[q]$ ($i=1,2,...n$; $q=0,1,...,\log n$) contains (the first component of) a descriptor for the substring of $x\#$ of length 2^q which starts at position i in $x\#$, with the constraint that all the occurrences of the same substring of x get the same descriptor. If, for some value of $q<\log n$, $NODE_i[q]$ is not empty, then it represents a node μ of stagenumber q in D_x, as follows: the field $NODE_i[q].LABEL$ is a replica of $ID_i[q]$, and the field $NODE_i[q].PARENT$ points to the location of the parent of μ. Finally, $NODE_i[\log n]$ stores the leaf labeled (i,n) and thus is nonempty for $i=1,2,...,n$. For convenience, we extend the notion of ID to all positions $i>n$ through the convention: $ID_i[q]=n+1$ for $i>n$. The computation makes crucial use of a *bulletin board* (BB) of $n\times n$ locations in the common memory. All processors can simultaneously write to BB and simultaneously read from it. If more than one processor tries to write to the same location, only one succeeds. In the following, we call *winner* (i) the index of the processor which succeeds in writing to the location of the common memory attempted by p_i.

Procedure Skeleton-Tree
input: x ; the ROOT of D_x .
output: $NODE_i[q]$; $ID_i[q]$; (i=1,2,...,n, q=0,1,...,log$n$).
begin

Each processor initializes its *NODE* and *ID* array. Next, processors facing the same symbol of I attempt to write their serial number in the same location of BB. Say, if $x_i=s \in I$, processor p_i attempts to write i in $BB[1,s]$. Through a second reading from the same location, p_i reads $j=winner(i)$ and sets $ID_i[0] \leftarrow j$. (Thus $(j,1)$ becomes the descriptor for every occurrence of symbol s). For all i such that $winner(i)=i$, processor p_i sets $NODE_i[0].PARENT \leftarrow ROOT$ and copies $ID_i[0]=i$ into $NODE_i[0].LABEL$. Hence $NODE_i[0]$ becomes the locus of s.

for q=0 **to** log$n$−1 **pardo**

begin

Processor p_i creates a composite label TID_i, as follows.
$TID_i \leftarrow (ID_i[q], ID_{i+2^q}[q])$.
Processor p_i attempts to write i in $BB[TID_i] = BB[ID_i[q],ID_{i+2^q}[q]]$. Thus all processors with the same *TID* attempt to write in the same location of BB. The processors read: if $BB[TID_j]=i$, then p_j sets $ID_j[q+1]$ to i. Formally: $ID_i[q+1] \leftarrow winner(i)$,i=1,2,...,n.
The processors that are not winners become idle for the remainder of the stage. The successful processors now first create new locuses in their associated *NODE* locations. Whenever a node μ is created that has no siblings, then the pointer from parent(μ) is removed and copied into μ. This avoids the formation of chains of unary nodes. The condition that a node has no siblings can be checked easily, as explained below. Formally, each successful processor p_i performs the following.
$NODE_i[q+1].PARENT \leftarrow NODE_{ID_i[q]}[q]$
$NODE_i[q+1].LABEL \leftarrow ID_i[q+1]$
if $NODE_{ID_i[q]}[q]$ has only one child **then**
begin
$NODE_i[q+1].PARENT \leftarrow NODE_{ID_i[q]}[q].PARENT$;
Make $NODE_{ID_i[q]}[q]$ empty.
end

end

end.

The existence of the siblings can be checked as follows. Assume that for each row r of BB, there is a distinct memory location, say $AUX[r]$, known to all processors. At each stage, there are siblings iff two or more successful processors write to different locations of the same row of BB. To find out whether this is the case, all successful processors

writing in the same row r of BB attempt to write their index in $AUX[r]$. Next, all the processors in that row except the winner write a special marker in $AUX[r]$. Finally, all the processors in the same row check the status of $AUX[r]$. Clearly, processor p_i was the only successful processor in row r iff, at the time of checking, $AUX[r]=i$.

The correctness of the procedure follows by straightforward induction. Since no two n-symbol substrings of $x\#$ are identical, processor p_i $(i=1,2,...,n)$ must be occupying the "leaf" $NODE_i[\log n]$ at the end of the computation. The time complexity is obviously $O(\log n)$. Note that $NODE_i[q].LABEL$ not empty implies $NODE_i[q].LABEL = (i,2^q)$, that is, the label of a node, when defined, is nothing but the address of that node. Although the $LABEL$ fields are entirely redundant so far, assuming this node format from the start simplifies the rest of our presentation. Finally, we remark that BB needs not to be initialized.

3. REFINING D_x

In this section, we assume for simplicity that $|I|=2$. We will show later that our solution can be extended to alphabets of arbitrary size with no substantial penalty.

By the end of the construction of D_x, processor p_i will be occupying leaf i, $i=1,2,...,n$. Prior to starting the transformation of D_x in T_x, the labels of all vertices of D_x have to be modified as follows. Recall that the current $LABEL$ of a node μ is a starting position of $W(\mu)$ in $x\#$ and also the address of this node. The modified label (*mlabel*) to be constructed for μ is any pair (i,l) such that, letting $W(\mu)=W(parent(\mu))\cdot w$, it is $l=|w|$ and i is the starting position of an occurrence of w in $x\#$. Set aside the orientation of links, the main difference between T_x and the m-labeled skeleton D_x is that in T_x

there cannot be two sibling nodes such that their labels describe two substrings of x having a common prefix (i.e., D_x is not a trie). However, the m-labeled D_x shares with T_x the properties (1-3) listed in defining the latter.

A processor can trivially compute the mlabel of μ in constant time knowing the *LABEL* of μ, and the stagenumbers, say q and q', of μ and *parent*(μ), respectively. Formally, if j is the *LABEL* of μ, then $(j+2^{q'},2^q-2^{q'})$ is the mlabel of μ. The n processors can produce all mlabels in $\log n$ parallel steps. Using the parent pointers, the processors migrate towards ROOT with a synchronous pace based on stagenumbers: the mlabels of all children of nodes with the same stagenumber are computed at the same time. (Recall that the difference in stagenumber between a node and its parent is not necessarily 1.) At the beginning, all processors occupying leaves which are children of nodes of stagenumber $\log n-1$ change the labels of these nodes into mlabels. Next, the processors compete for the common parent node, say, by attempting to simultaneously write on it the labels (addresses) of the nodes which they currently occupy. The winners are marked "free": they ascend to the parent node where they will perform the necessary label adjustment at the appropriate stage. The losers simply take a record of the (old) label used by the winner. The $(q-1)$-th iteration involves all free processors on nodes with a stagenumber of q or higher. The operation is the same as above.

A byproduct of the mlabel construction process is a mapping that assigns some leaves and internal nodes to processors in such a way that the following property is met.

PROPERTY 1. If a node other than *ROOT* has k children, then precisely $k-1$ of the children have been assigned a processor. Moreover, each one of the $k-1$ processors knows the address of the unique sibling without a processor.

The proof of *property* 1 is straightforward. Let now (i,l) and (j,m) be the mlabels of two sibling nodes μ and ν of D_x, and let q be the stagenumber of *parent*(μ)=*parent*(ν).

FACT 1. The substrings of $x\#$ whose descriptors are the labels of μ and ν have a common prefix of length at most 2^q-1.

FACT 2. If k is the length of the longest common prefix of $x\#[i,i+l-1]$ and $x\#[j,j+m-1]$, then $ID_i[\lfloor \log k \rfloor] = ID_j[\lfloor \log k \rfloor]$.

Fact 1 follows from the definition of D_x, Fact 2 holds by the construction of the *ID*'s.

Assuming a binary alphabet, the transformation of the mlabeled version of D_x into T_x is done in two steps. First, a tree is produced that is identical to T_x save the fact that all arcs are directed upward, as in D_x. Next, the directions of all arcs are reversed.

The first and more important step is actuated by producing $\log n-1$ consecutive refinements of $D_x=D^{\log(n-1)}$. The q-th such refinement, denoted by $D^{\log(n-q-1)}$, is a labeled tree with n leaves and no unary nodes which has much the same structure of the mlabeled D_x. In particular, properties (1-3) of the definition of T_x hold for any refinement of D_x. In particular, $D^{(0)}$ is identical to T_x except for the arc directions. To specify the labels in the generic $D^{(k)}$, let a *nest* be any set formed by all children of some node in $D^{(k)}$, and let (i,l) and (j,k) be the labels of two nodes in some nest of $D^{(k)}$. An integer t is a *refiner* for (i,l) and (j,k) iff $x\#[i,i+t-1]=x\#[j,j+t-1]$. $D^{(k)}$ is labeled in such a way that no pair of labels of nodes in the same nest of $D^{(k)}$ admits of a refiner of size 2^k. This latter property is similar, though not identical, to the property in Fact 1.

The update transforming $D^{(k)}$ into $D^{(k-1)}$ affects all and only the *eligible* nests of $D^{(k)}$, i.e., those nests which might admit of a refiner of size $2^{(k-1)}$. Let $(i_1,l_1),(i_2,l_2),...,(i_m,l_m)$ be the set of all labels in some such nest of $D^{(k)}$. The transformation of the nest is performed in two steps.

STEP 1. Use the *LABEL* and *ID* tables to modify the nest rooted at v, as follows. With the child node labeled (i_j,l_j) associate the *split*-label $(ID_{i_j}[k-1],ID_{i_j+2^{k-1}})$, $j=1,2,...,m$. Now partition the children of v into equivalence classes, putting in the same class all nodes with the same first component of their split-labels. For each non-singleton class which results, perform the following three operations.

(1) Create a new parent node μ for the nodes in that class, and make μ a son of v.

(2) Set the *LABEL* of μ to $(i,2^{(k-1)})$, where i is the first component of the split-label of all nodes in the class.

(3) Consider each child of μ. For the child whose current *LABEL* is (i_j,l_j), change *LABEL* to $(i_j+2^{(k-1)},l_j-2^{(k-1)})$.

STEP 2. If more than one class resulted from the partition, then stop. Otherwise, let C be the unique class resulting from the partition. It follows from the definition of D_k that C cannot be a singleton class. Thus a new parent node μ as above was created for the nodes in C during STEP 1. Make μ a child of the parent of v and set the *LABEL* of μ to $(i,l+2^{(k-1)})$, where (i,l) is the label of the parent of v.

Theorem 1 The synchronous application of Steps 1 and 2 to all eligible nests of $D^{(k)}$ correctly produces $D^{(k-1)}$.

Proof. We prove that $D^{(k-1)}$ is a tree with no unary nodes. The correctness of the labels of $D^{(k-1)}$ relies on Fact 2: we leave the details as an exercise.

Clearly, the nest of the children of the root is not eligible for any $k>0$. Thus for any parent node v of an eligible nest of $D^{(k)}$, *parent*(*parent*(v)) is defined. By definition of $D^{(k)}$, v has more than one child, and so does *parent*(v). Let $\overline{D}^{(k)}$ be the structure resulting from application of Step 1 to $D^{(k)}$.

If, in $D^{(k)}$, the nest of *parent*(v) is not eligible, then v is a node of $D^{(k-1)}$, and v may be the only unary node in $\overline{D}^{(k)}$ between any child of v in $D^{(k)}$ and the parent of v in $D^{(k)}$. Node v is removed in STEP 2, unless v is a branching node in $\overline{D}^{(k)}$. Hence no unary nodes result in this part of $D^{(k-1)}$.

Assume now that, in $D^{(k)}$, both the nest of v and that of *parent*(v) are eligible. We claim that, in $\overline{D}^{(k)}$, either the parent of v has not changed and it is a branching node, or it has changed but still is a branching node. Indeed, by definition of $D^{(k)}$, neither the nest of v nor that of *parent*(v) can be refined in only one singleton equivalence class. Thus, by the end of STEP 1, the following alternatives are left.

1. The parent of v in $\overline{D}^{(k)}$ is identical to *parent*(v) in $D^{(k)}$. Since the nest of *parent*(v) could not have been refined into only one singleton class, then *parent*(v) must be a branching node in $D^{(k-1)}$. Thus this case reduces to that where the nest of *parent*(v) is not eligible.

2. The parent of v in $\overline{D}^{(k)}$ is not the parent of v in $D^{(k)}$. Then *parent*(v) in $\overline{D}^{(k)}$ is a branching node, and also a node of $D^{(k-1)}$. If v is a branching node in $\overline{D}^{(k)}$, then there is no unary node between v and *parent*(v) in $\overline{D}^{(k)}$, and the same holds true between any node in the nest of v and v. If v is an unary node in $\overline{D}^{(k)}$, then the unique child of v is a branching node. Since the current parent of v is also a branching node by hypothesis, then removing v in STEP 2 eliminates the only unary node existing on the path from any

node in the nest of v to the closest branching ancestor of that node. □

If the nest of $D^{(k)}$ rooted at v had a row R of BB all to itself, then the transformation undergone by this nest in Step 1 can be accomplished by m processors in constant time, m being the number of children. Each processor handles one child node. It generates the split-label for that node using its $LABEL$ and the ID tables. Next, the processors use the row of BB assigned to the nest and the split-labels to partition themselves into equivalence classes: each processor in the nest whose split-label has first component i competes to write the addresses of its node in the i-th location of R. A *representative* processor is elected for each class in this way. Singleton classes can be trivially spotted through a second concurrent write restricted to loosing processors (after this second write, a representative processor which still reads its node address in R knows to be in a singleton class). The representatives of each nonsingleton class create now the new parent nodes, label them with the first component of their split-label, and make each new node accessible by all other processors in the class. To conclude STEP 1, the processors in the same class update the labels of their nodes.

For STEP 2, the existence of more than one equivalence class needs to be tested. This is done through a competition of the representatives which uses the root of the nest as a common write location, and follows the same mechanism as in the construction of D_x. If only one equivalence class was produced in STEP 1, then its representative performs the adjustment of label prescribed by STEP 2.

The above discussion suggests that, once each vertex of, say, D_x=$D^{(\log n-1)}$ is assigned to a distinct processor, $D^{(\log n-2)}$ could be produced in constant time. The difficulty, however, is now with how to assign the vertices (notably, the newly inserted

ones) of $D^{(\log n-2)}$ in constant time. It turns out that profusing less processors into the game leads to a crisp (re-)assignment strategy.

By definition, $D^{(k)}$ does not have unary nodes. It is seen then that the manipulations of Steps 1-2 can be operated in constant time by assigning $m-1$ processors, rather than m to a nest of m nodes. The only additional assumption to be made is that, at the beginning, all $m-1$ processors have access to the unique node which lacks a processor of its own. Before starting STEP 1, the processors elect one of them to serve as a substitute for the missing processor. After each elementary step, this simulator "catches-up" with the others.

In view of Property 1, this shows that n processors can achieve the first refinement of D_x. As for which row of BB is assigned to which node of $D^{(k)}$, simply assign the i-th row to processor p_i. Then, whenever p_i is in charge of the simulation of the missing processor in a nest, its BB row is used by all processors in that nest.

For any given value of k, let a *legal* assignment of processors to the nodes of $D^{(k)}$ be an assignment that enjoys Property 1.

Theorem 2. Given a legal assignment of processors for $D^{(k)}$, a legal assignment of processors for $D^{(k-1)}$ can be produced from it in constant time.

Proof. We give first a constant-time policy that re-allocates the processors in each nest of $D^{(k)}$ on the nodes of $\overline{D}^{(k)}$. We show then that our policy leads to a legal assignment for $D^{(k)}$.

Let then v be the parent of a nest of $D^{(k)}$. A node which has a processor assigned to it will be called *pebbled*. By hypothesis, all children of v but one are pebbled. Also, all children of v are nodes of $\overline{D}^{(k)}$. In the general case, some of the children of v in $D^{(k)}$ are

still children of v in $\bar{D}^{(k)}$, whereas others became children of newly inserted nodes $\mu_1, \mu_2, \ldots, \mu_t$. Our policy is as follows. At the end of STEP 1, for each node μ_r of $\bar{D}^{(k)}$ such that all children of μ_r are pebbled, one pebble (say, the representative processor) is chosen among the children and passed on to the parent. In STEP 2, whenever a pebbled node v is removed, then its pebble is passed down to the (unique) son μ of v in $\bar{D}^{(k)}$.

Our policy can be clearly implemented in constant time. To prove its correctness, we need to show that it generates a legal assignment for $D^{(k-1)}$.

It is easy to see that if node v is removed in the transition from $\bar{D}^{(k)}$ to $D^{(k-1)}$, then the unique son μ of v in $\bar{D}^{(k)}$ is unpebbled in $\bar{D}^{(k)}$. Thus, in STEP 2, it can never happen that two pebbles are moved onto the same node of $D^{(k-1)}$.

By definition of $D^{(k)}$, the nest of node v cannot give rise to a singleton class. Thus at the end of STEP 1, either (Case 1) the nest has been refined in only one (nonsingleton) class, or (Case 2) it has been refined in more than one class, some of which are possibly singleton classes.

Before analyzing these two cases, define a mapping f from the children in the nest of the generic node v of $D^{(k)}$ into nodes of $D^{(k-1)}$, as follows. If node μ is in the nest of v and also in $D^{(k-1)}$ then set $\mu' = f(\mu) = \mu$; if instead μ is not in $D^{(k-1)}$, let $\mu' = f(\mu)$ be the (unique) son of μ in $\bar{D}^{(k)}$.

In Case 1, exactly one node μ is unpebbled in $\bar{D}^{(k)}$. All the nodes μ''s are in a single nest of $D^{(k-1)}$ and, by our policy, μ' is pebbled in $D^{(k-1)}$ iff μ is pebbled in $D^{(k)}$.

In Case 2, node v is in $D^{(k-1)}$. Any node μ in the nest of v is in $\bar{D}^{(k)}$. At the end of STEP 2, the pebble of node μ will go untouched unless μ is in a nonsingleton equivalence class. Each such class generates a new parent node, and a class passes a

pebble on to that node only if all the nodes in the class were pebbled. Thus, in $D^{(k-1)}$, all the children of v except one are pebbled by the end of STEP 1. Moreover, for each non-singleton equivalence class, all nodes in that class but one are pebbled. At the end of STEP 2, for each node μ which was in the nest of v in $D^{(k)}$, node μ' is pebbled iff μ was pebbled at the end of STEP 1, which concludes the proof. □

For a binary alphabet, $D^{(0)}$ is a ternary tree (due to the symbol #). Since the processors are legally assigned to the vertices of $D^{(0)}$ at the end of the computation, then the concurrent reversal of all arcs is straightforwardly achievable in constant-time.

For general alphabets, each node of $D^{(0)}$ must be changed into a binary tree before arc reversal can take place. Such a change can be obtained through a series of $\log|I|$ refinements of $D^{(0)}$ quite similar to those already discussed. The major difference is that now the ID tables are useless, since a more compact descriptor for a substring of x of $\log|I|$ bits or less is the substring itself. We leave the details of this part as an exercise.

References

[AA] A. Apostolico, The Myriad Virtues of Suffix Trees, *Combinatorial Algorithms on Words (A. Apostolico and Z. Galil, eds.)*, pp. 85-96, Springer Verlag (1985).

[GA] Z. Galil, Open Problems in Stringology, *Combinatorial Algorithms on Words (A. Apostolico and Z. Galil, eds.)*, pp. 1-10, Springer Verlag (1985).

[KMR] R. M. Karp, R. E. Miller and A.L. Rosemberg, Rapid Identification of Repeated Patterns in Strings, Trees and Arrays, *Proceedings of the 4-th ACM STOC*, pp. 125-136 (1972).

[MC] E. M. McCreight, A Space Economical Suffix Tree Construction Algorithm, *JACM 23*, 2, 262-272 (1986).

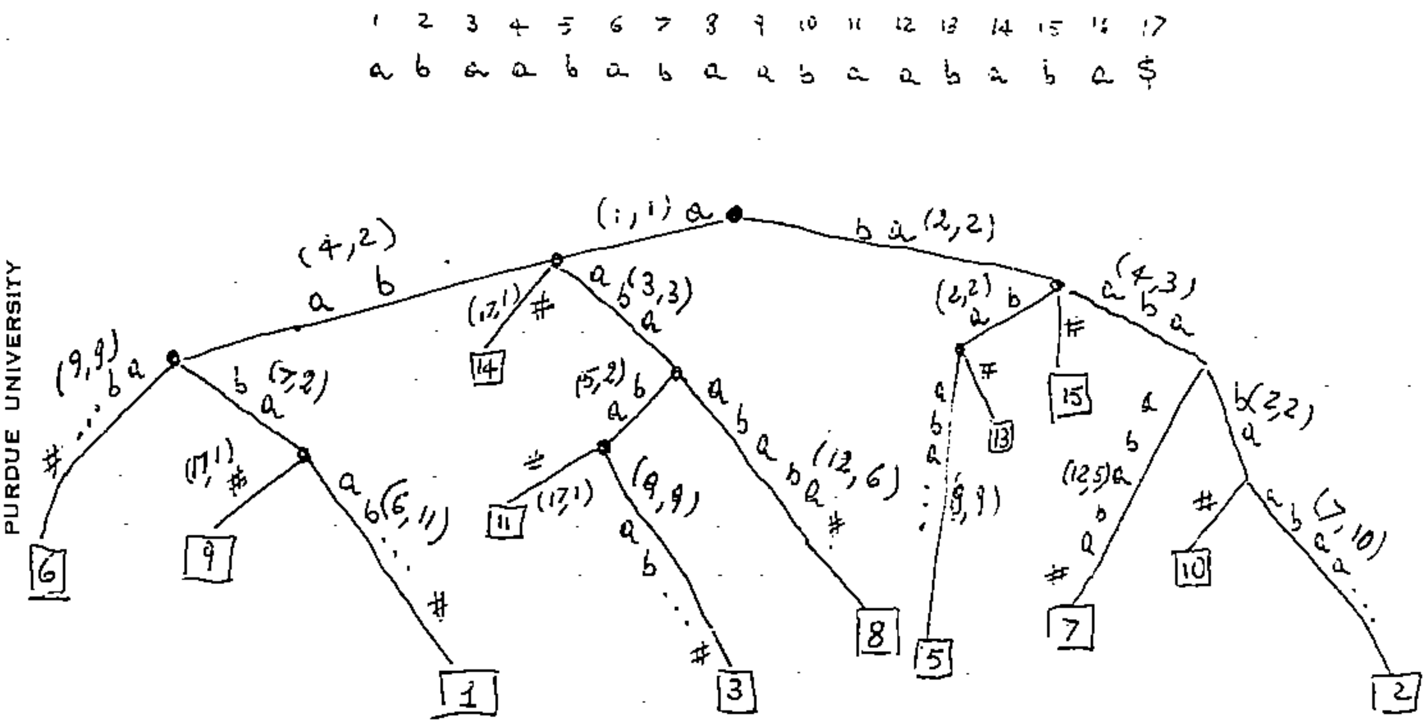

Fig. 1. The suffix tree for x = abaababaabaababa.
For convenience, arcs are labeled with both substrings of x and their descriptors.

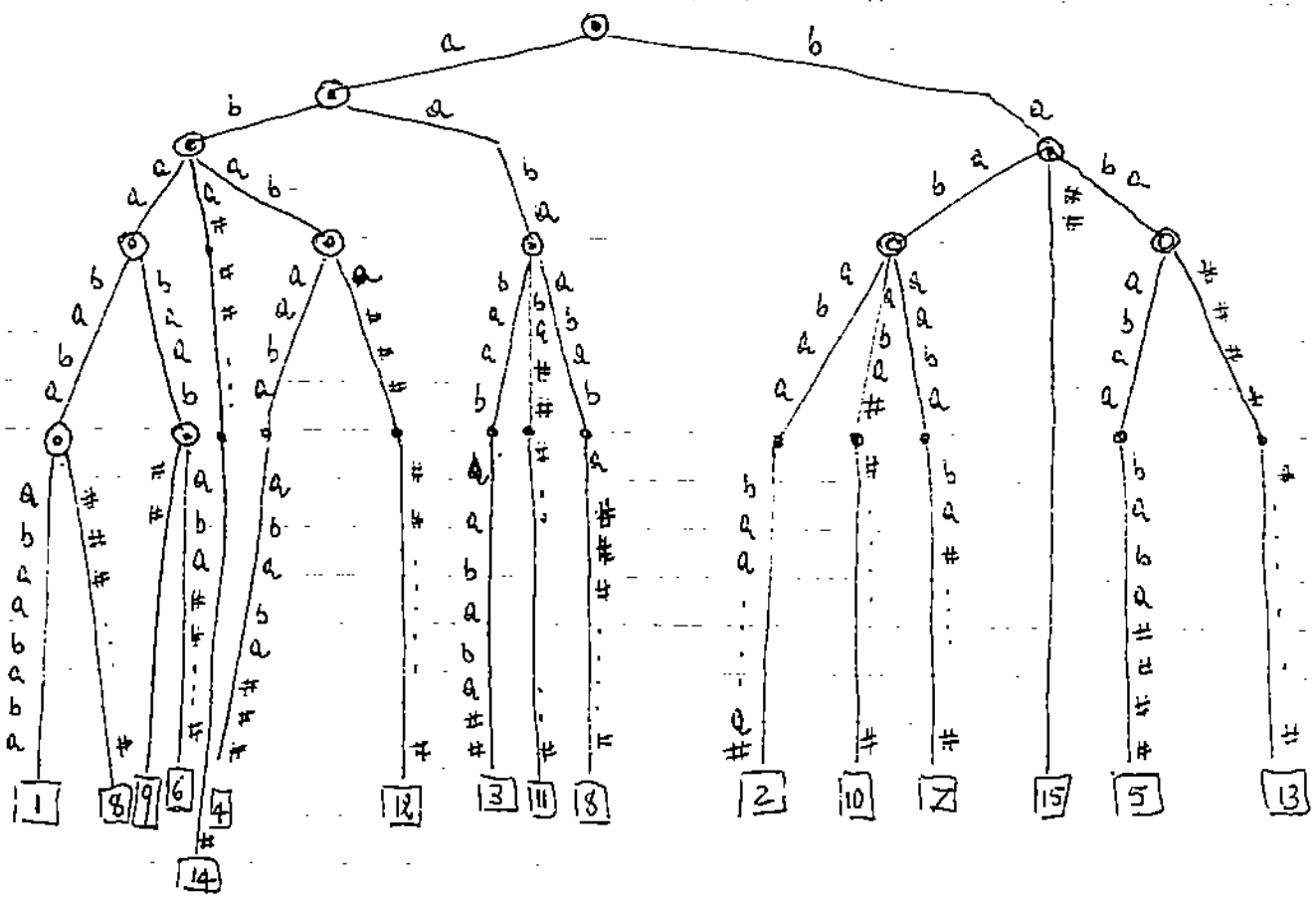

Fig. 2

The skeleton tree for the string of Fig. 1. Solid points mark deleted nodes. Node labels are not reported.