Symmetry Breaking for Suffix Tree Construction

(extended abstract)

Süleyman Cenk Ṣahinalp * *Uzi Vishkin* †

Abstract

There are several serial algorithms for suffix tree construction which run in linear time, but the number of operations in the only parallel algorithm available, due to Apostolico, Iliopoulos, Landau, Schieber and Vishkin, is proportional to $n \log n$. The algorithm is based on labeling substrings, similar to a classical serial algorithm, with the same operations bound, by Karp, Miller and Rosenberg. We show how to break symmetries that occur in the process of assigning labels using the Deterministic Coin Tossing (DCT) technique, and thereby reduce the number of labeled substrings to linear. We give several algorithms for suffix tree construction. One of them runs in $O(\log^2 n)$ parallel time and $O(n)$ work for input strings whose characters are drawn from a constant size alphabet.

1 Introduction

Suffix trees are apparently the single most important data-structure in the area of string matching.

We present a parallel method for constructing the suffix tree T of a string $S = s_1 \ldots s_n$ of n symbols, with s_n being a special symbol \$ that appears nowhere else in S. We use A to denote the *alphabet* of S. The suffix tree T associated with S is a rooted tree with n leaves such that:

1. Each path from the root to a leaf of T represents a different suffix of S.
2. Each edge of T represents a nonempty substring of S.
3. Each nonleaf node of T, except the root, must have at least two children.
4. The substrings represented by two sibling edges must begin with different characters.

An example of a suffix tree is given in Figure 1.

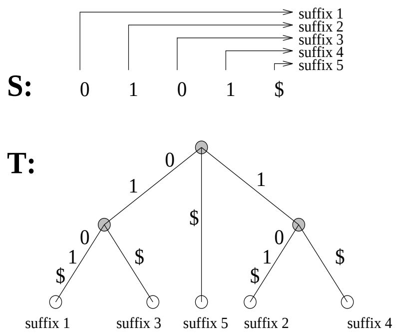

Figure 1: Suffix tree T of string $S = 0\ 1\ 0\ 1\ \$$

Serial algorithms for suffix tree construction were given in [KMR72], [We73], and [Mc76]. The two latter algorithms achieve a linear running time for an alphabet whose size is constant.

A parallel algorithm was given in [AILSV88].

A Symmetry Breaking Challenge: As in the algorithm of [KMR72] work complexity of the above mentioned parallel algorithm is $O(n \log n)$. The approach of [KMR72] and [AILSV88] does not lend itself to linear work for the following reason: As these algorithms progress, they label **all** $n-1$ substrings of size 2, then all $n-3$-substrings of size 4, and in general all $(n-2^i+1)$-substrings of size 2^i $(1 \leq i \leq \log n)$. This results in a number of labels which is proportional to $n \log n$ and this dictates the work complexity. The extra logarithmic factor in the label-count is due to the increasing redundance among these substrings (because of the overlaps), as they become longer. The problem is that there has been no consistent way for selecting only one among a subset of overlapping substrings, since they all "look-alike". The main new idea of this paper is in introducing a solution to this *symmetry breaking problem.*

Our **most interesting concrete result** is in being able to build a suffix tree (of a string of characters selected from a constant size alphabet), optimally in $O(\log^2 n)$ time.

The general area of string matching has been enriched by parallel methods that enabled new serial algorithms as in this paper. Previous examples include [Ga85], [Vi85], and [Vi91]. The new method is also relevant for sequence analysis in compressed data, since it allows for consistent compression

*Department of Computer Science, University of Maryland, College Park, MD 20742;

†Institute for Advanced Computer Studies, and Department of Electrical Engineering, University of Maryland, College Park, MD 20742; and Dept. of Computer Science, Tel Aviv University, Tel Aviv, Israel; Partially supported by NSF grants CCR-8906949 and CCR-9111348.

of data. This can be done in the context of parallel or serial algorithms.

2 The Basic Algorithm

We first describe a "basic" algorithm. The algorithm is randomized and runs in $O(n \log^* n)$ work, and $O(n^\epsilon)$ time (for any constant $0 < \epsilon \leq 1$) for an alphabets whose size is bounded by a polynomial in n. We describe how to improve it to an optimal (linear work) deterministic algorithm which has a time complexity of $O(\log^2 n)$ for a constant size alphabet in the next section. The ideas that will be presented in this section can be used to obtain an $O(\log^2 n)$ time algorithm, again for alphabets of size polynomial in n. However, the work complexity of this algorithm will be $O(n \log \log n)$.

2.1 High-level Description

The basic algorithm works in three stages. In the **first stage** we attach **labels** to various substrings of S, recognizing some identities. This is done in iterations.

In iteration 1, S is partitioned into at most $n/2$ blocks. Each block is labeled with a number between 1 and n, in a way which satisfies the following two **consistency properties**:

Partition-consistency (we state this property informally) Denote by X_i some "long enough" substring of S (which starts) at location i and denote by X_j a substring at location j, which is equal to X_i; then, with the exception of some (left and right) margins, X_i and X_j will be partitioned in the same way.

Label-consistency All blocks consisting of the same string of characters will get the same label.

An example of consistent partitioning and consistent labeling is given in Figure 2.

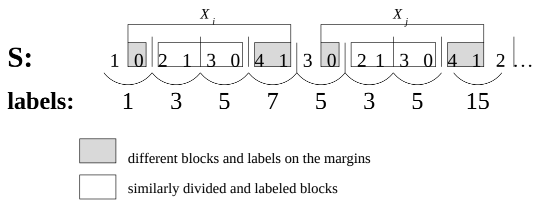

Figure 2: Consistent partitioning, consistent labeling and margins

So, iteration 1 partitions $S = S(0)$, "shrinking" it into a new string $S(1)$; the length of $S(1)$ is at most half the length of $S(0)$. Subsequent iterations apply the same procedure. Iteration i, $i = 2, 3, \ldots$ shrinks string $S(i-1)$ into string $S(i)$ satisfying similar partition-consistency and label-consistency properties. The size of string $S(i)$ will be at most $n/2^i$.

The labels of blocks obtained in iteration i are called ***i*-labels**, and the substrings they represent in S are called ***i*-substrings**. Here we note an important distinction; an ***i*-label** is a name which represents an i-substring. This label is considered as an ***i*-character** in $S(i)$, which will be the input for the iteration $i+1$ of the first stage. The i-substrings are said to be "built up" of $i-1$-substrings.

To motivate the second stage, we note that the strings $S(i)$, $i = 0, 1, 2, \ldots$, provide a hierachical system of subsets over the set of indices $\{1, 2, \ldots, n\}$, which are coarser and coarser partitions of $\{1, 2, \ldots, n\}$. Specifically, $S(0)$ provide singleton subsets, where each subset contains a single index. The subsets of $S(i)$ satisfy the following: (i) each subset of $S(i)$ is the union of subsets of $S(i-1)$; (ii) the union of the subsets of $S(i)$ is the set $\{1, 2, \ldots, n\}$; and (iii) the intersection of every pair of subsets is empty.

We wish to construct the suffix tree of S in iterations. However, the partition-consistency property above is too weak for us as we cannot guarantee for any two identical substrings of S that they are partitioned, and hence labeled, in the same way. The **Second Stage** develops an alternative hierarchical system of subsets. A subset in the alternative system is called a **core**. In the context of cores we will use the letter C for denoting substrings.

Similar to the First Stage, $C(0) = S(0) = S$, and the cores (subsets) of $C(i)$, $i = 1, 2, \ldots$ satisfy that: (i) each core of $C(i)$ is the union of cores of $C(i-1)$; and (ii) the union of the cores of $C(i)$ is the set $\{1, 2, \ldots, n\}$. However, (iii) the intersection of some pairs of core of $C(i)$ is not empty.

Cores representing the same substring will be given the same label. The label of a core is called a name. We give a small example to clarify the relation between substrings and labels of cores.

Example 1 Let $s_{11} = a$, $s_{12} = b$, $s_{13} = a$, $s_{14} = c$, $s_{15} = a$, $s_{16} = d$, $s_{17} = e$, $s_{18} = b$, $s_{19} = d$, $s_{20} = c$, $s_{21} = a$, $s_{22} = c$, $s_{23} = a$ be a substring of S and suppose that $\{11, 12, 13, 14, 15, 16, 17, 18\}$, $\{14, 15, 16, 17, 18, 19\}$, and $\{16, 17, 18, 19, 20, 21, 22, 23\}$ are cores of $C(1)$. Let the labels of these cores be 3, 2 and 6 respectively. Then, in $C(1)$ a substring 3 2 6 will correspond to the above substring of S.
Now, let $s_{31} = a$, $s_{32} = b$, $s_{33} = a$, $s_{34} = c$, $s_{35} = a$, $s_{36} = d$, $s_{37} = e$, $s_{38} = b$, $s_{39} = a$, $s_{40} = c$, $s_{41} = a$, $s_{42} = c$, $s_{43} = a$ be a substring of S, and suppose that $\{31, 32, 33, 34, 35, 36, 37, 38\}$, $\{33, 34, 35, 36, 37, 38, 39\}$, and $\{36, 37, 38, 39, 40, 41, 42, 43\}$ are cores of $C(1)$. Let the labels of these cores be 3, 1 and 5 respectively. Then, in $C(1)$ a substring 3 1 5 will correspond to the above substring of S.
Note that: (1) The single inequality of s_{19} and s_{39} implied more than one inequality in $C(1)$. (2) The redundancy among cores. For instance, in the substring 3 2 6 of $C(1)$, we do not really need the character 2 since the core of 3 intersects the core of 6. (3) Wherever needed we will also keep the number of characters at the beginning of each core before the next core begins (for the core of 3 above this number is 3 and for the core of 2 this number is 2.)

The reason for moving to the alternative system is that it enables to satisfy the following property: Consider two cores of some $C(i)$ which represent two equal substrings of S. Then, they will have the same core label. This property is stronger than the partition-consistency property above and will guarrantee that the Third Stage can be implemented quickly in parallel and at the same time will not miss any identities.

The **Third Stage** builds the suffix tree of S, as follows: The input for the last iteration is $T(1)$, which is the suffix tree of the labels of $C(1)$. The last iteration constructs the suffix tree of $C(0) = S$. The i'th-prior-to-the-last iteration constructs the suffix tree $T(i)$ (the suffix tree of labels derived from $C(i)$), by using $T(i+1)$.

2.2 First Stage

Consider iteration h of the **first stage**. The input for this iteration will be the string S_h of m characters (for some $n/2^h \geq m > 0$). S_h will be denoted by R for the rest of this section. Iteration h partitions R into m_1 blocks and labels each block. At this point we only say that $n/2^{h+1} \geq m_1 > 0$. The problem is how to do it so that: (1) the properties of partition-consistency (defined formally later) and label-consistency are satisfied; and (2) a too rapid shrinking does not occur (a too rapid shrinking may cause problems at a later stage of the algorithm). An iteration consists of two steps: partitioning R into block, and labeling the blocks. We first describe the partition step and then the labeling step.

2.2.1 Overview of the partition step

1. Only characters r_i whose substring length (in the input string S) is **short**, concretely whose length is $\leq 2^{h+2}$, "participate" in the iteration. For each character whose substring is longer than 2^{h+2} (such a character is called **long**), put block dividers to its left and right and this character "quits" the current iteration.

2. Each character r_i, now checks if it is in a substring of a single repeated character; that is, r_i is compared with r_{i+1} and r_{i-1}.

   - If not (i.e., $r_i \neq r_{i+1}$ and $r_i \neq r_{i-1}$) then we say that r_i belongs to a **changing substring**. For a changing substring $R_C = r_{j+1} \dots r_{j+k}$ we have that $r_j \neq r_{j+1} \neq r_{j+2} \dots r_{j+k} \neq r_{j+k+1}$ and neither r_j nor r_{j+k+1} belongs to a changing substring. For each changing substring we apply procedure CONTENT-BASED (which is described below). At this point we only need to know that this procedure partitions a changing substring $R_C = r_{j+1} \dots r_{j+k}$ of length larger than 1 (i.e., $k > 1$) into blocks of size 2 or 3.

   - If $r_i = r_{i+1}$ but $r_i \neq r_{i-1}$, then a *block divider* is put between r_i and r_{i-1}; similarly if $r_i = r_{i-1}$ but $r_i \neq r_{i+1}$, then a *block divider* is put between r_i and r_{i+1}. This gives blocks of single repeated characters. Such a block consists of a substring of the form $R_R = r_{j+1} \dots r_{j+k}$, where $r_j \neq r_{j+1} = r_{j+2} \dots = r_{j+k} \neq r_{j+k+1}$, for some $k \geq 2$.

3. The iteration should guarantee that the total number of blocks is $\leq n/2^{h+1}$. For this, we still need to consider characters r_i which form singleton blocks. A character r_i forms a singleton block if each of its immediate left and right neighbors (i.e., r_{i-1} and r_{i+1}) belongs to a block of a repeated character, or is a 'long character' itself (i.e., it is the label of a substring longer than 2^{h+2}). Below, we actually treat only a subset of such characters.

   - If the left block of such r_i consists of a single character which is repeated exactly twice (i.e., $r_{i-3} \neq r_{i-2} = r_{i-1} \neq r_i$), we merge r_i with this block to obtain a block of size three.

   - If such r_i is the first character of R (i.e., $r_i = r_1$), we merge it with its right block.

It is easy to see that if the length of R is $\leq n/2^h$, then the number of resulting blocks is $\leq n/2^{h+1}$.

2.2.2 Additional details on the partition step

Our detailed description starts with presenting procedure CONTENT-BASED. Consider a substring $R_C = r_\alpha \dots r_\beta$, where $r_i \neq r_{i+1}$, for $\alpha \leq i < \beta$. Namely, we do not allow a substring of the form aa. The main idea behind the *partitioning procedure* below is the use of the *deterministic coin tossing technique* of Cole and Vishkin [CV86a] for dividing R_C into blocks.

1. Put a divider to the left of r_α and to the right of r_β. Each instruction for putting a divider below should be augmented with the following *caveat*: we actually put the new divider only if it does not create a block consisting of a single character.

2. Put a divider to the right of $r_{\alpha+1}$. Put a divider to the left of $r_{\beta-1}$.

3. For each character r_i of R_C compute tag_i (in parallel), as follows. Suppose that r_i and r_{i+1} are given in binary representation. The tag_i is the index of the least significant bit in which r_i is different than r_{i+1}. If r_{i+1} does not exist (for now, this can happen only if $i = m$), set $tag_i := 0$. By saying "in parallel" we imply that this step has to be finished before proceeding to the next step.

4. For each character r_i of R_C, compare tag_i with tag_{i+1} and tag_{i-1}. If any of them does not exist (for now this can happen if $i = \alpha$, or $i = \beta$), take the non-existing value to be 0. For all "strict local maxima" (i.e., $tag_i > tag_{i+1}$ and $tag_i > tag_{i-1}$) put (in parallel) a block divider between r_i and r_{i+1}. For all "weakly local maxima" (i.e. $tag_i \geq tag_{i+1}$, and $tag_i \geq tag_{i-1}$), put (in parallel) a block divider between r_i and r_{i+1}, if bit tag_i of r_i is 1.

5. We consider separately each substring of R_C, which lies between two dividors, and do the following for each. If the substring has ≤ 3 elements those elements quit. Otherwise, for each character r_i, replace character r_i by tag_i, and recursively apply the CONTENT-BASED procedure to the substring.

Example 2 (content-based partitioning) Let $R = \dots 3\ 3\ 8\ 4\ 2\ 1\ 2\ 4\ 8\ 4\ 8\ 4\ 8\ 8 \dots$. Typically, we will apply the CONTENT-BASED procedure to a longest substring which satisfies the input conditions which in this

case will be:
R_C = 8 4 2 1 2 4 8 4 8 4 . After applying steps 1 and 2 we have:
|8 4|2 1 2 4 8 4|8 4| . In binary representation R_C becomes: 1000 0100|0010 0001 0010 0100 1000 0100|1000 0100 , and the corresponding *tag* values are:
3 2 1 1 2 3 3 3 3 0 . Hence in the first round of CONTENT-BASED, we partition R_C as follows:
8 4|2 1 2 4|8 4|8 4 . Then we apply CONTENT-BASED procedure again to get:
8 4|2 1|2 4|8 4|8 4 , as the final partitioning of R_C.

Comment. Our use of the deterministic coin tossing technique is novel. We use it for deriving "signatures" of strings, mapping similar substrings to the same signature. The only previous paper which made use of this technique for producing signatures is apparently by Mehlhorn, Sundar and Uhrig [MSU94]. They limited the use of these signatures to comparing full strings, and did not consider substrings. We had to develop significantly stronger tools for constructing suffix trees.

Lemma 3 (consistency lemma) Let R_d be a substring of R. Let R'_d be another substring of R which is equal to R_d. All but (at most) $\log^* n + 1$ characters in the right margin and (at most) $\log^* n + 1$ characters in the left margin of R_d are called the *interior* of R_d. If the above first stage puts a block divider in some location in the interior of R_d then it also puts a divider in the same location at R'_d.

2.2.3 The labeling step

Each block is labeled with a number between 1 and m to satisfy label-consistency. To achieve this, we determine the size of each block. This can be obtained by first applying a *nearest one* procedure to determine the starting and ending locations of each block, which takes $O(\log m)$ time with linear work [BV88]. (Given an array of n bits, a nearest one procedure finds for each position in the array the nearest bits whose value is one to it the left and right of the position.) Then we label each block in $O(1)$ time with $O(m)$ work in three substeps.

1. We first attach labels to blocks of a repeating single character (called, blocks of **repetition substrings**). This can be done easily in an arbitrary CRCW PRAM by using an array L of size m X m. Given a block of a character c which is repeated l times, its index in R is written, using the arbitrary-write convention into location $L(c, l)$. The index which is actually written into some $L(c, l)$ will then be used as the label by all blocks in which the character c is repeated l times.
2. In the second substep we attach labels to all blocks of size 2, as well as to the first two characters of all blocks of size 3. This is done using an $m \times m$ array L. For each such two-character substring ij, where $1 \le i, j \le m$, the location of i is entered into entry $L(i, j)$. This is similar to [AILSV88].
3. Now, we attach labels to all block of size 3. For each three-character substring ijk, we obtain its label by reapplying the second substep to fk where f denotes the label of ij as computed in the second substep.

The space complexity is $O(m^2)$. This can be reduced to $O(m^{1+\epsilon})$, for any fixed $\epsilon > 0$, as in [AILSV88]. Getting linear space using hashing (and thereby entering randomization), as suggested in [MV91], is also a possibility.

An example for block partitioning and labeling is given in Figure 3.

S(0): 0 2 | 1 3 0 | 4 1 | 0 0 0 0 | 1 | 0 0 | 2 1 | 3 0 | 4 1 | 0 0 0 0 | 2
S(1): 1 3 | 6 8 | 12 13 15 | 17 6 | 8 25
S(2): 1 3 5 8 10
⋮ ⋮

Figure 3: Block partitioning and labeling for successive iterations

2.3 Second Stage

We define a core of iteration k recursively. Any k-substring $S_{k,i}$ induces, or **spans** another substring of input which is called a k**-core**. Given a substring S_k, we show how to **extend** it to the left and to the right to obtain the k-core C_k that S_k **spans**. We use the following systematic notations and definitions. We denote $S_{k,i}$ by S_k. The $(k-1)$-substrings that build up $S_{k,i}$ will be denoted $S_{k-1,a}, \ldots, Sk-1, b$. The core spanned by a string denoted by S_α will be denoted by C_α. The label for S_α will be denoted by L_α. The core name of C_α will be denoted by N_α. The prefix of C_k which lies to the left of $S_{k-1,a}$ will be called the left extension of C_k and the suffix of C_k which lies to the right of $S_{k-1,b}$ will be called the right extension of C_k. The recursive definition follows.

1. If $k = 0$, then $C_k = S_k$.
2. If S_k is a long k-substring, built up of a single $(k-1)$-substring $S_{k-1,a}$, then c_k, the k-core spanned by S_k, will be equal to the $(k-1)$-core spanned by $S_{k-1,a}$, namely $C_{k-1,a}$.
3. (a) Consider the first $\log^* n + 3$ $(k-1)$-substrings to the left of $S_{k-1,a}$ (also called the *left vicinity* of S_k) and the $\log^* n + 3$ $(k-1)$-substrings to the right of $S_{k-1,b}$ (the *right vicinity* of S_k). Formally $S_{k-1,a-\log^* n+3}, \ldots, S_{k-1,a-1}$ is the left vicinity and $S_{k-1,b+1}, \ldots, S_{k-1,b+\log^* n+3}$ is the right vicinity. If none of them are long then C_k will be the concatenation (with overlaps) of the cores spanned by the $(k-1)$-substrings $S_{k-1,a-\log^* n+3}, \ldots, S_{k-1,b+\log^* n+3}$; namely, $C_{k-1,a-\log^* n+3}, \ldots, C_{k-1,a-1}$, $C_{k-1,a}, \ldots, C_{k-1,b}$, $C_{k-1,b+1}, \ldots, C_{k-1,b+\log^* n+3}$.
   (b) If some of $S_{k-1,a-\log^* n+3}, \ldots, S_{k-1,a-1}$ is long then denote the rightmost one among them, as $S_{k-1,left}$. The left extension of C_k will include all $(k-1)$-substrings to the right of $S_{k-1,left}$ until $S_{k-1,a}$ is reached. There will be an addition to that. Consider the repeated string of l-labels whose l-substrings build up $S_{k-1,left}$. C_k

will also include the rightmost 2^{k-l+1} of the l-cores spanned by these substrings. A similar definition is used for the right extension of C_k. Let the leftmost long substring (if exists) among $S_{k-1,b+1}, \ldots, S_{k-1,b+\log^* n+3}$ be $S_{k-1,right}$. The right extension of C_k will include all $(k-1)$-substrings to the left of $S_{k-1,right}$ until $S_{k-1,b}$ is reached. Given the repeated string of r-labels whose r-substrings build up $S_{k-1,right}$, C_k will also include the leftmost 2^{k-r+1} of the r-cores spanned by these substrings.

To visualize how C_k looks like, consider the third case, with no long $(k-1)$-substrings in the left and right vicinities of S_k. In the middle of C_k, there will be S_k. To its left there will be $S_{k-1,a-\log^* n-3}, \ldots, S_{k-1,a-1}$ which are $(k-1)$-substrings. To their left there will be more $\log^* n\ +3$ strings which are $(k-2)$-substrings, and so on till finally there will be $\log^* n\ +3$ 0-substrings (i.e. singletons). The concatenation of these substrings to the left of S_k is the **left extension** of S_k. There is also a symmetric **right extension**. So, a k-core is a "double staircase". Figure 4 illustrates such a double staircase where the left vicinity is the simple case where no long substring is encountered, while the right vicinity has a long $(k-1)$-substring.

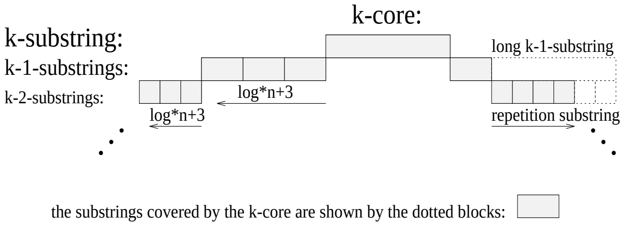

Figure 4: S_k is extended by $\log^* n + 3$ $(k-1)$-substrings followed by $\log^* n + 3$ $(k-2)$-substrings and so on towards left, while for the right extension case 3(b) applies

An example of a core is given in Figure 5.

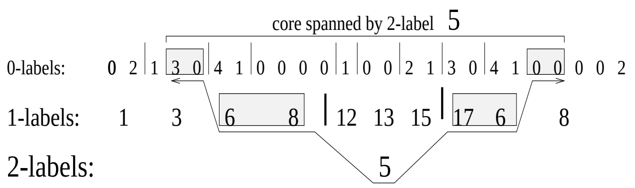

Figure 5: An example of a core. For illustration purposes, assume $\log^* n\ +3\ =\ 2$ (which is not possible)

We now give some definitions and then present some important properties of cores. The proofs of the lemmas are left to the full paper ([SV94])

The suffix (of the original pattern) which begins at the leftmost character of a k-core is called the **suffix of that core**, and is referred to as a k-**suffix**. Note that, if there are no long substrings in the left vicinity of a $(k+1)$-substring then its $(k+1)$-suffix is also a k-suffix. Moreover, we define a **long core** as a core spanned by a long substring and a **repetition k-core** as a long k-core which is spanned by a k-substring that is built up of repeated $(k-1)$-characters.

Lemma 4 (length lemma) The length of each of the left and right extensions of a core C_k, is $\leq 2(2^{k+2})(\log^* n+3)$

The cores obtained by the second stage satisfy the following **"Core Property"**.

Lemma 5 (core lemma) If a substring C_1 of S is a k-core for some iteration k, and if C_2 is another substring which is identical to C_1, then: (i) C_2 is also a k-core, and (ii) C_1 and C_2 are given the same core name.

The following lemma shows that if a k-substring is built up of many $(k-1)$-substrings, then all but a few of the cores that are spanned by these $(k-1)$-substrings should have identical names.

Lemma 6 (middle of core identity lemma) Suppose S_k is build up of more than $16(\log^* n+3)+2$ $(k-1)$-substrings (i.e. $b-a \geq 16(\log^* n+3)$); then the $(k-1)$-names of the $(k-1)$-cores $C_{k-1,a}, \ldots, C_{k-1,b}$ (denoted by $N_{k-1,a}, \ldots, N_{k-1,b}$), are in the following form: There exists two numbers $\alpha, \beta\ \leq\ 8(\log^* n+3)$ such that $N_{k-1,a+\alpha+1}\ =\ \ldots\ =\ N_{k-1,b-\beta-1}$.

A corollary to Lemma 6, will be one of the key ideas that would enable us to perform the **third stage** efficiently:

Corollary 7 The structure of the left extensions of C_k, in terms of j-cores ($j \leq k$) are as follows (a similar statement applies to right extensions):

- If all $(k-1)$-substrings in the left vicinity are short, then the left extension of C_k is built up of $\log^* n+3$ $(k-1)$-cores.
- If there is a long $(k-1)$-substring in the left vicinity, then C_k is built up of the following: To the left, there will be at most $8(\log^* n+3)+2$ l-cores (with possibly different core names), followed (to their right) by some number of l-cores with identical core names, followed (to their right again) by at most $8(\log^* n+3)+2$ l-cores (with possibly different core names), followed by at most $\log^* n+2$ $(k-1)$-cores (with possibly different core names).

2.3.1 Implementation of the second stage

For each $k\ =\ 1,2,\ldots$, where k is an iteration of the **first stage**, do the following. For each k-substring, compute the k-core that it spans and label each such core with a core name, which is called a k-**core name**.

The name computation is similar to the label computation of substrings in the **first stage**. There, a k-substring was built up of at most three $(k-1)$-substrings with different labels or two or more $(k-1)$-substrings with identical core names. This enabled us to use a table of size m^2, where m is the size of $S(k-1)$. Here, a k-core is built up of:

1. 2^{k-l+1} l-cores among which all but at most $16(\log^* n+3)+1$ are identical; followed by

2. some number of $(k-1)$-cores in the following pattern: a string of at most $(1+8)(\log^* n+3)$ $(k-1)$-cores is followed by a string of some number of identical $(k-1)$-cores, which is then followed by a string of at most $(1+8)(\log^* n+3)$ $(k-1)$-cores; followed by
3. 2^{k-r+1} r-cores.

Therefore, we have $O(\log^* n)$ different names. Casting everything in a string of length $O(\log^* n)$ of characters each consisting of $\log n$ bits is straightforward, since we only need to mark for each i-core its number of repetitions, and the index i. To name the k-cores consistently in $O(\log^* n)$ time with $O(m \log^* n)$ work, we keep a table of size n^2 by applying a similar procedure to the one in the**first stage**. This space requirement can be decreased to linear by the use of hashing techniques ([MV91]).

The time complexity of this stage is $O(\log^* n)$ per iteration. The total time complexity for this stage is $O(\log n \log^* n)$. The total work complexity is $O(n \log^* n)$.

2.4 Third Stage

The suffix tree is constructed iteratively. (extended abstract) In iteration k of the **Third Stage**, the suffix tree $T(k)$ **of k-cores using k-core names** (in other words, the suffix tree of the string $C(k)$) is built by using $T(k+1)$. $T(k+1)$ has limited resolution, as some possible identical prefixes between two $(k+1)$-cores cannot be precisely expressed with $(k+1)$-core names. **Iteration k builds $T(k)$ from $T(k+1)$ by improving the resolution with a more dense set of cores which are shorter**. Note that, as a matter of convenience, the iterations are numbered in reverse, where the final iteration (whose serial number is 1) gives the desired suffix tree.

Suppose a $(k+1)$-core forms a substring A in S. The substring of $C(k)$ which forms the longest substring in S which is included in A is said to be **the substring of k-cores covered** by A.

A concise representation of the string of k-cores that are covered by each $(k+1)$-core looks as follows.

Fact 8 Given a $k{+}1$-core, C_{k+1}, the string of k-cores covered by C_{k+1}, can be represented by a string of $O(\log^* n)$ tuples, where each tuple consists of two coordinates: the name of a k-core, and an integers which stands for the number of repetitions of the k-core.

During the Third stage it so happens that we first find identities between full tuples (i.e., both name coordinate and number coordinate match). Later on, we find identities between the name coordinate of different tuples.

2.4.1 Overview of the Third Stage

We need to do several things in order to build $T(k)$ from $T(k+1)$:

1. **Get Tree** $T(k)_0$. Procedure REFINE refines $T(k+1)$ into $T(k)_0$, in the following sense: We replace each $(k+1)$-core name on $T(k+1)$ with the names of the k-cores it covers. This involves the following substeps:
   - Take the first $(k+1)$-core name on each edge of $T(k+1)$ that is incident on the root of $T(k+1)$. Replace it by the strings of k-core names that it covers (see Example 9).
   - Now replace the rest of the $(k+1)$-core names by the strings of the names of the k-core that they cover. Due to overlaps, each $(k+1)$-core should only "take care" of being replaced by those k-core names that are not covered by its predecessor in $T(k+1)$.

   Notice that in the case where there are long k-substrings in the left vicinity of a $(k{+}1)$-substring, the suffix of this $(k{+}1)$-core is different from the suffix of the leftmost k-core it covers. Hence the suffixes we consider for $T(k)_0$ are not necessarily the suffixes of $T(k+1)$.

   Now, for each node of $T(k+1)$, **merge** its outgoing edges that have a common prefix of k-core names. The common prefix between the two suffixes which begin with two sibling edges can not exceed a string that any of the two edges represent. In other words a situation such as in Figure 6 cannot happen.

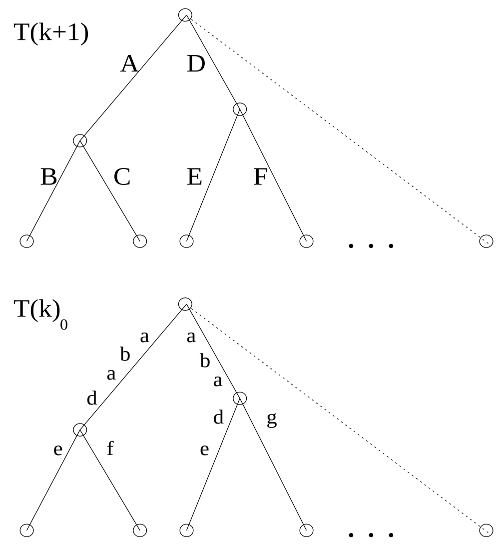

Figure 6: It is impossible to have three $(k+1)$-cores, A, D and E, with A becoming a,b,a,d, B becoming a,b,a, and D beginning with d

   This fact is implied by the core property and plays a significant role in the analysis of our algorithm.

2. **Get Tree** $T(k)_1$. Consider $C(k)$. A $(k+1)$-core is represented in $C(k)$ by the string of names of k-cores that it covers. The tail of C_k is defined as the substring in $C(k)$ which starts with N_k, includes the core names of all k-cores till the end of the first full $(k+1)$-core to C_k's right. An example of a tail is given in Figure 7. The tail of each k-core will be represented by a string of $O(\log^* n)$ tuples (each with $2\log n$ bits).

   Compute the tail of each k-core C_k in $C(k)$.

   Divide all k-cores into equivalence classes, $t_1, \ldots, t_\alpha$, according to their tail; namely, the cores having the same tail should be in the same equivalence class.

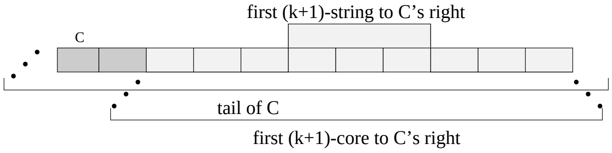

Figure 7: An illustration of the tail of a k-core, C

For each equivalence class t_x, have in $T(k)_1$ an edge, e_x, which comes out of its root (the edge represents the tail corresponding to this class) and a node n_x, at the end of this edge.

By way of motivation assume that each node n_x is the root of the full $T(k)_0$, and it should be clear that the resulting (huge) tree will enable to represent every suffix in $C(k)$. The problem, of course, is that many strings which are not suffixes in $C(k)$ are also included.

So, for each node n_x (and its equivalence class t_x) procedure CONTRACT contracts $T(k)_0$ to obtain a subtree beneath the node n_x; this subtree completes the representation of suffixes in $C(k)$ that go through n_x. The general idea is as follows. We first: (1) preprocess the tree $T(k)_0$, so that the lowest common ancestor (LCA) of each pair of nodes in it can be retrieved in a constant number of operations ([BV88], [SV88]); and (2) sort the leaves of $T(k)_0$ according to their order of appearance in preorder traversal of $T(k)_0$. For this the Euler Tour technique is applied. Focus now on the tree beneath some node n_x, and consider two leafs of this tree which represent actual suffixes in $C(k)$. The lowest common ancestor of the two leafs gives a node which should appear in the final $T(k)$. The next simple observation is that we do not really need to find the lowest common ancestor of every pair of such leafs; instead, we consider the leafs beneath each node n_x separately, and find the lowest common ancestor of each successive pair of leafs only. It turns out that this gives all the information needed for extracting from each copy of $T(k)_0$ the subtree needed for $T(k)$; we suppress implementation details of how to actually do this.

3. **Get Tree** $T(k)$. The only thing missing in $T(k)_1$ is that equal prefixes of tails have not been identified. For this, procedure MERGE merges the edges adjacent to the root of $T(k)_1$ (which represent distinct equivalence classes of tails). From Lemma 5, we know that the common prefix between the two suffixes which begin with two sibling edges emanating from the root, can not exceed a string that any of the two edges represent.

2.4.2 Further Details for the Implementation of the Third Stage

1. **Get tree** $T(k)_0$. Procedure REFINE, which will derive tree $T(k)_0$ from tree $T(k+1)$, works as follows. For obtaining $T(k)_0$, the basic idea is: For every internal node of $T(k+1)$, advance through the edges incident to it, by merging sibling edges that represent identical core names into a single edge.

The next example demonstrates how procedure REFINE works.

Example 9 (construction of $T(k)$) Consider Example 1 above. The tree $T(k)$, for an appropriate k will represent the suffix starting at s_{11} and at s_{31}. Suppressing the existence of other suffixes, the root will have an outgoing edge labelled 3 leading to a node denoted u. Node u will have two outgoing edges. One is labeled by the chain $2, 6$ followed by the subsequent cores in $C(k)$ and leads to a leaf l_1. The path from the root to leaf l_1 represents the suffix starting at s_{11}. Similarly, there is another outgoing edge from u which leads to a leaf l_2, which is labeled by $1, 5, \ldots$, and the path from the root to l_2 represents the suffix starting at s_{31}. See also Figure 8 (a).

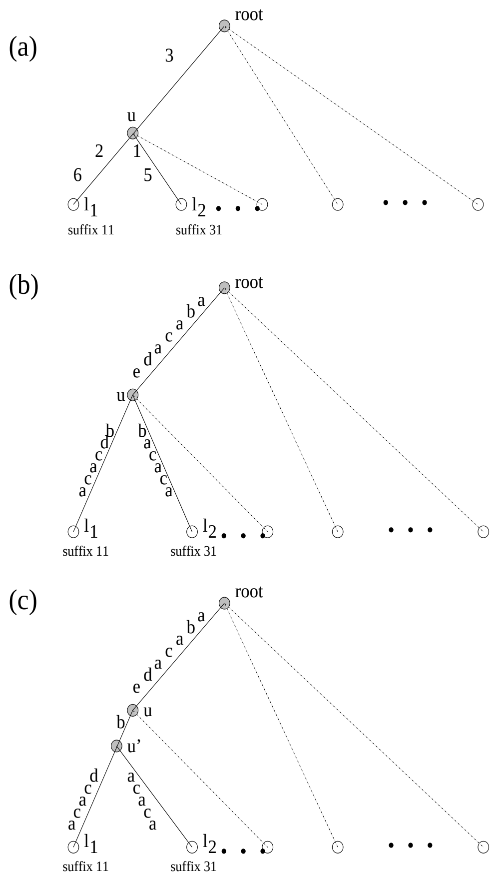

Figure 8: refining $T(k+1)$

To get $T(k)$ we first replace the $(k+1)$-core name, 3 with the string of k-core names (a, b, a, c, a, d, e). The $(k+1)$-core name 2 represents the string of k-core names (c, a, d, e, b, d); but because of the overlaps it is replaced by the string (b, d). Similarly, the $(k+1)$-core names, 6, 1, and 5, which represent the strings of k-core names (d, e, b, d, c, a, c, a), (a, c, a, d, e, b, a) and (d, e, b, a, c, a, c, a), are replaced by the strings (c, a, c, a), (b, a), and (c, a, c, a), respectively (see Figure 8 (b))..

After this replacement, the resolution with respect to $T(k+1)$ is improved by merging sibling edges by observing similar prefixes.

In our example, the two sibling edges, incident on the node u, represent the strings of k-core names, (b,d,c,a,c,a), and (b,a,c,a,c,a). We merge the two edges, as the first k-core they represent (which is b) is identical and obtain the node u' (see Figure 8 (c)).

Details of procedure REFINE
The substring of k-cores, of an edge, e, is represented by $O(\log^* n)$ tuples; the first coordinate being the k-core name and the second coordinate being how many times this k-core name is repeated successively. Let the first tuple of an edge, e, be referred as to $e(name, number)$.

For $O(\log^* n)$ iterations, we do the following to get an intermediate tree $T(k)'_0$:

- For each internal node, divide the edges incident to it into equivalence classes according to the first tuple they represent.
- Merge the edges that are in the same equivalence class. If all the edges incident to a node fall into one class, remove this node.

This simple strategy improves the resolution of $T(k+1)$. However we have not considered sibling edges, e_α, e_β with $e_\alpha(name) = e_\beta(name)$, and, $e_\alpha(number) \neq e_\beta(number)$. For merging these sibling edges, we lexicographically sort triples consisting of the parent of an edge (to identify sibling edges), and the two-coordinate tuples which label the edges.

2. **Get tree** $T(k)_1$. For each equivalence class we apply the CONTRACT procedure.
3. **Get tree** $T(k)$. The procedure MERGE works identically to the procedure REFINE, with the difference that it is applied to only the edges adjacent to the root of $T(k)_1$.

Complexity of Stage 3 The main problem of the third stage is the need for sorting to apply REFINE, CONTRACT and MERGE procedures. The range of elements to be sorted enables applying the integer sorting algorithm of [BDHPRS89], so that this algorithm runs in logarithmic time; however its work complexity is $O(m \log\log m)$ for a list of size m.

2.5 Complexity of the basic algorithm

The basic algorithm runs in $O(\log^2 n)$ time and $O(n \log\log n)$ work, for an alphabet hose size is polynomial in n.

An alternative implementation takes $O(n^\epsilon)$ time and $O(n \log^* n)$ work for any fixed $epsilon < 1$.

3 The Optimal Algorithm

We first recall the general case, where the input string is drawn from an alphabet whose size is bounded by a polynomial in n. The "basic algorithm" did not achieve linear work because of the following three problems:

1. Partitioning a string of size m into blocks, in the first stage, may need a number of operations which is proportional to $m \log^* n$. Naming the cores in the second stage needs also $O(m \log^* n)$ operations.
2. In the third stage, we need to perform integer sorting in each iteration for m integers, selected from a range of $1, \ldots, m^2$. At present, there is no known parallel algorithm which solves this problem optimally in polylogarithmic time.
3. The refinement of $T(i)$, at iteration i of the third stage, may need $\log^* m$ steps for each node, implying a number of operations which is proportional to $m \log^* m$.

Henceforth, we restrict ourselves to the case where the input string is drawn from an alphabet whose size is bounded by some constant c.

The optimal algorithm begins as follows. The first few, say x, iterations of the Basic algorithm, are replaced. ($x = O(\log\log^* n)$.) The goal is to reduce the length of $S(k)$ and $C(k)$ to below $n/(\log^* n)^2$. The output of iterations 1 through x in the First and Second stage of the Basic algorithm, are computed using an alternative algorithm. Similarly the last few (specifically, $\log\log_c \log n$) iterations of the Third stage of the Basic algorithm are replaced by an alternative algorithm. The other iterations of the three stages of the Basic algorithm will remain unchanged. (The design of the optimal algorithm is essentially an application of the accelerating cascades method for deriving an efficient parallel algorithm from two or more algorithms for the same problem, see [CV86b], or [Ja92].)

We first explain how we solve the first problem and then the second and third problems.

3.1 Solution of the first problem:

We start by discussing the emulation of the partitioning step in the first iteration (in the First and Second stage) of the Basic algorithm. The restriction to an alphabet of constant size enables to use a table look up method. Recall that a block divider is put in the location between two successive characters based on at most $\log^* n + 1$ characters to the left and at most $\log^* n + 1$ characters to the right. The number of possible substrings of size $2\log^* n + 2$ is $c^{2\log^* n + 2}$. So, we build a **look-up table** of size $c^{2\log^* n+2}$; given a location between two successive characters, the substring of length $\log^* n + 1$ to its left and the substring of of length $\log^* n + 1$ to its right, an entry of the table will tell whether a divider should be put at the location.

Building the table and retrieving information from it is standard and is , therefore, suppressed here.

A similar (actually simpler) table is used for assigning names to cores in the first x iterations of the second stage.

The next $x - 1$ iterations of the Basic algorithm are emulated in a similar way. The size of the table will be $O(n/\log^* n)$; so, the number of operations in an iteration is linear in the size of the input string for the iteration, and the time is sublogarithmic. As we are (at least) halving the size in each iteration, the total number of operations in the first x iterations will remain linear in n.

After the first x iterations, we can obviously keep the work linear throughout the first stage of the algorithm, as the size of the string for the next iteration is $O(n/(\log^* n)^2)$.

3.2 Solution of the second and third problems

The third stage of the optimal algorithm begins by using all but the last $\log\log_c \log n$ iterations of the previous section; since the deterministic integer sorting algorithm of [BDHPRS89] needs $O(m \log\log m)$ work for sorting m elements (from a range $1,\ldots,m^2$), we can use it in each of these iterations, and still satisfy a linear work upper bound.

Recall that if we can restrict the range of the values to be sorted to integers between 1 and $O(\log^2 n)$, then we can apply the deterministic stable sorting algorithm of [CV86a] to achieve linear work in $O(\log^2 n)$ time.

At this point we would like to present an overview of the improved third stage, starting from iteration $k = \log\log_c \log n$ (from the end). However, we leave the details of implementation to the full paper ([SV94]). There are four main steps. The first three aim at computing a suffix tree, to be denoted T', for some specific subset of suffixes relative to the input string $S = S(0)$. The definition of T' is given in Step 2.

1. We REFINE $T(k)$ "fully", by replacing the cores in $T(k)$ with the actual substrings in S; we replace the k-core names by substrings of S which span these cores and advance through characters of S. This gives the suffix tree for those suffixes which start at the locations where k-cores start (with respect to S).

2. We obtain a new string S', from $S(k)$ in the following way: Consider the (k)-labels in $S(k)$, which represent long substrings (at iteration k). Long substrings at iteration k should be longer than 2^{k+2}. Take the long substrings which consist of a single repeated label of some iteration $< k$. Replace each of these labels of long strings with this substring of a single repeated label. This new string will be S'. Let C' be the sequence of cores which are spanned by the substrings labelled by characters of S'. Consider the suffixes in S which are the same as the ones implied by the suffixes of C'. The suffix tree T' will be defined with respect to these suffixes in S.

3. We construct the suffix tree T', by using the suffix tree computed in Step 1.

4. Using T', we construct T, the final suffix tree of all suffixes of S.

   The following property (which is an obvious corollary to Lemma 5) guides us in doing this: Let $S(i)$ and $S(j)$ be two suffixes of S, and let P denote their (longest) common prefix. Any l-core (for all $0 < l \leq \log n$) which is included (relative to S) in P, and appears in one among $S(i)$ and $S(j)$ must appear in the other (with the same l-core name).

   The construction of T from T' is similar to the way steps 2 and 3 of the third stage construct $T(k)$ from $T(k)_0$.

   The main difference is that tables are used for representation of tails (including identity among tails). Since tails relative to C' are not too long, it is is possible to limit ourselves to tables whose size is at most $O(\log n)$.

3.3 Complexity of the optimal algorithm

The optimal algorithm runs in $O(\log^2 n)$ time and $O(n)$ work for an input drawn from an alphabet whose size is constant.

4 Conclusion

The method given in this paper enables to quickly identify long similarities among substrings. Actually, the first stage of the basic algorithm should be very useful. Furthermore, on-line implementation will enable to quickly identify similarities between recently received substrings and ones received earlier, in the spirit of Lempel-Ziv's data-compression algorithm ([LZ77]).

References

[AILSV88] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, Parallel Construction of a Suffix Tree with Applications, In *Algorithmica*, 3: 347–365, 1988.

[BV88] O. Berkman, and U. Vishkin, Recursive star-tree parallel data-structure, In *SIAM J. Computing*, 22,2: 221–242, 1993.

[BDHPRS89] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena, Improved Deterministic Parallel Integer Sorting, In *Information and Computation*, 94: 29–47, 1991.

[BLMPSZ91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J Smith, M. Zagha, A Comparison of Sorting Algorithms for the Connection Machine CM-2, In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[CV86a] R. Cole, and U. Vishkin, Deterministic Coin Tossing with Applications to Parallel List Ranking, In *Information and Control*, 70: 32–53, 1986.

[CV86b] R. Cole, and U. Vishkin, Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms, In *Proceedings of the 18th Annual ACM Symposium on the Theory of Computing*, pages 206–219, 1986.

[Ga85] Z. Galil, Optimal Parallel Algorithms for String Matching, In *Information and Control*, 67: 144–157, 1985.

[Ja92] J. Ja'Ja', An Introduction to Parallel Algorithms, Addison-Wesley, 1992.

[KMR72] R. M. Karp, R. E. Miller, and A. L. Rosenberg, Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays, In *Proceedings of the* 4^{th} *Annual ACM Symposium on the Theory of Computing*, pages 125–136, 1972.

[LZ77] J. Ziv, and A. Lempel, A Universal Algorithm for Sequential Data Compression In *IEEE Transactions on Information Theory*, 23: 337–343, 1977.

[MV91] Y. Matias, and U. Vishkin, On Parallel Hashing and Integer Sorting, In *Journal of Algorithms*, 12,4: 573–606, 1991.

[Mc76] E. M. McCreight, A Space - Economical Suffix Tree Construction Algorithm, In *Journal of the ACM*, 23: 262–272, 1976.

[MSU94] K. Mehlhorn, R. Sundar, and C. Uhrig, Maintaining Dynamic Sequences under Equality - Tests in Polylogarithmic Time, to appear In *Proceedings of the* 5^{th} *Annual ACM - SIAM Symposium on Discrete Algorithms*, 1994.

[ȘV94] S. C. Șahinalp, and U. Vishkin, Symmetry Breaking in Suffix Tree Construction, In preparation

[SV88] B. Schieber, and U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization, In *SIAM Journal of Computing*, 17: 1253-1262, 1988.

[Vi85] U. Vishkin, Optimal Parallel Pattern Matching in Strings, In *Information and Control*, 67: 91–113, 1985.

[Vi91] U. Vishkin, Deterministic Sampling - A New Technique for Fast Pattern Matching, In *SIAM Journal of Computing*, 20: 22–40, 1991.

[We73] P. Weiner, Linear Pattern Matching Algorithm, In *Proceedings of the* $latex stoc.final.4.tex 14^{th}$ *IEEE Symposium on Switching and Automata Theory*, pages latex stoc.final.4.tex1–11, 1973.