# Centroid Trees with Application to String Processing

Fei Shi and Dong-Guk Shin

Dept. Math and Computer Science
Suffolk University
Boston, MA 02114-428, USA
shi@cas.suffolk.edu
and
Dept. Computer Science and Engineering
The University of Connecticut
Storrs, CT 06269-3155, USA
shin@eng2.uconn.edu

e-mail: {goeman,clausen}@cs.uni-bonn.de

**Abstract.** A centroid of a tree $T$ is a node $v$ which minimizes over all nodes the largest connected component of $T$ induced by removing $v$ from $T$. A centroid tree $U$ of another tree $T$ is defined on the same set of nodes of $T$: the root $v$ of $U$ is a centroid of $T$ and the subtrees of $v$ (in $U$) are the centroid trees of the connected components of $T - v$. We describe some interesting properties of the centroid and of the centroid tree. Our linear algorithm to find a centroid of a tree improves on the previously known algorithms either in terms of space requirement or in terms of time requirement. From the algorithm for finding a centroid it is easy to obtain an $O(n \log n)$ time algorithm to construct a centroid tree of a given tree with $n$ nodes. However, we do not know whether this is the best that one can achieve. By exploiting the properties of the centroid tree, we devise an efficient algorithm for the longest common substring problem (LCS). Given two strings $S$ (the text) of length $n$ and $P$ (the pattern) of length $m$, the LCS problem is to find the longest substring that appears in both the text and the pattern. Our algorithm requires $O(n \log n)$ time and $O(n)$ space to preprocess the text. After preprocessing of the text, the algorithm takes $O(m \log n)$ time using $O(m)$ extra space to find the solution. The algorithm may be used in the DNA applications in which the text is very large and fixed and is to be searched with many different patterns ($n \gg m$).

**Key words:** balanced trees, centroid of trees, string pattern matching, the longest common substring problem

## 1 Introduction

Let $T$ be an arbitrary tree and let $V$ denote the set of nodes in $T$. Let $v \in V$ and let $T_1, T_2, \cdots, T_d$ be the connected components of $T$ induced by removing $v$ from $T$ (denoted by $T - v$). Let $|T|$ denote the number of nodes in $T$. Define

$$N(v) = \max_{1 \le i \le d} \{|T_i|\}.$$

A *centroid* of the tree $T$ [Har69] is a node $v_c$ which minimizes $N(v)$ over all nodes $v$, i.e.,$v_c$ satisfies

$$N(v_c) = \min_{v \in V}\{N(v)\}.$$

It can be shown that every tree has either one centroid or two. This fact has been extensively applied (see, for examples, [Gol71], [KH79], [FJ80], [MTZC81], [Sla82]). Goldman [Gol71] and Megiddo et al. [MTZC81] proposed linear algorithms for finding the centroid of a tree. All algorithms known to us that make use of a centroid finding algorithm call either Goldman's algorithm or Megiddo's algorithm as a subroutine.

Goldman's algorithm requires a copy of the original tree $T$ as an auxiliary tree on which it works. Therefore, $O(n)$ extra space is needed. While Megiddo's algorithm does not need any extra space, it has to visit each node *at least* once. In this paper, we present an algorithm, which might be viewed as a combination of Goldman's algorithm and Megiddo's algorithm. Our algorithm improves on the mentioned two algorithms either in terms of space or in terms of time. Specifically, our algorithm does not need an extra copy of the original tree; at the same time, it does not need more time than Goldman's algorithm. Our algorithm visits each node of the tree at *most* once; at the same time, it does not need more space than Megiddo's algorithm. Of course, one cannot improve the complexity order of the two mentioned algorithms since both are asymptotically optimal in terms of space and time.

The notion of the centroid of a tree inspired the notion of the centroid tree. A centroid tree $U$ of another tree $T$ has the same set of nodes as $T$. $U$'s root $v$ is a centroid of $T$ and $v$'s children (in $U$) are the centroid trees of the connected components of $T - v$. A nice property of the centroid tree is that its height is $\log n$. It is easy to obtain an $O(n \log n)$ time algorithm to construct a centroid tree from the algorithm for finding a centroid of a tree. However, it is unknown whether this is the best time complexity that one can achieve.

By exploiting the properties of the centroid tree, we are able to give an efficient algorithm for the longest common substring (LCS) problem. Given two strings $S$ (the text) of length $n$ and $P$ (the pattern) of length $m$, the LCS problem is to find the longest substring that appears in both the text and the pattern. An efficient solution to the problem can be useful for homology searching in nucleotide/protein sequence databases [Wat89], in the editing distance problem, in the multiple pattern searching problem, etc. Our algorithm requires $O(n \log n)$ time and $O(n)$ space to preprocess the text. After the preprocessing, a query can be answered in $O(m \log n)$ time. The algorithm is probabilistic and there is a small chance of error. That is, it may claim that a substring of the pattern is identical to a substring of the text while they are not really identical. This is called a "false match". However, the probability of a false match can be made arbitrarily (inverse-polynomially) small within the above time bounds. Our algorithm has obvious advantages over the previously known algorithms and is particularly useful for the DNA applications in which the text is very large and fixed ($n \gg m$) and in which one wishes to search the text with many different patterns (For example, the DNA sequence of a human being may have up to $3 \times 10^9$ nucleotides and a typical pattern sequence may have a few hundreds to thousands nucleotides).

The rest of the paper is organized as follows. In Section 2 we present our algorithm for finding a centroid of a tree. We address the problem of constructing a centroid tree in Section 3. In Section 4 we devise an algorithm for the LCS problem applying the results presented in Sections 2 and 3. We then conclude the paper by discussing some open problems in Section 5.

## 2 Finding centroid

**Lemma 2.1 ([Har69])** *Every tree has either one centroid or two. In the later case, the two centroids are connected by an edge.*

If $i$ and $j$ are two neighboring nodes of the tree $T$, then by removing the edge $(i, j)$ two connected components $C(i, j)$ and $C(j, i)$ are induced: $C(i, j)$ is the component which contains node $i$ and $C(j, i)$ is the component which contains node $j$ (Note that $C$ is defined on ordered pairs of neighboring nodes). Let $u$ be a node of $T$ and let $x_1, \cdots, x_d$ be all neighbors of $u$. Then $C(x_1, u), \cdots, C(x_d, u)$ are all the connected components of $T - u$. In the following we sometimes simply use $C(i, j)$ to refer to $|C(i, j)|$ (i.e., the number of nodes in $C(i, j)$) when no ambiguity would likely occur.

The following lemma is crucial for our algorithm to find a centroid of a tree correctly.

**Lemma 2.2** *A node $v$ is a centroid of the tree $T$ if and only if*

$$N(v) \leq n/2.$$

**Proof** We first prove the necessary condition of the lemma. Let $v$ be a centroid of the tree $T$. Suppose $N(v) > n/2$. Let $x_1, \cdots, x_d$ be all neighboring nodes of $v$. Then by the definition of a centroid, there must exist a neighboring node, say $x_{i_0}$, of $v$ such that $C(x_{i_0}, v) > n/2$. Let $y_1, \cdots, y_{k-1}, y_k = v$ be all neighboring nodes of $x_{i_0}$. Then,

$$N(x_{i_0}) = \max\{C(y_1, x_{i_0}), C(y_2, x_{i_0}), \cdots, C(y_{k-1}, x_{i_0}), C(v, x_{i_0})\} \tag{13}$$

We then have

$$\begin{array}{ll} N(x_{i_0}) & = C(v, x_{i_0}) \text{ if } C(v, x_{i_0}) \geq C(y_j, x_{i_0}) (j = 1, \ldots, k-1) \\ N(x_{i_0}) & < C(x_{i_0}, v) \text{ otherwise .} \end{array} \tag{14}$$

Since $C(x_{i_0}, v) > n/2$, $C(v, x_{i_0}) < n/2$. It then follows that

$$N(x_{i_0}) < C(x_{i_0}, v) = N(v).$$

So by the definition of a centroid of a tree, $v$ cannot be a centroid of the tree $T$. This contradicts the assumption that $v$ is a centroid of the tree $T$ and therefore establishes the necessary condition of the lemma.

We now turn to prove the sufficient condition of the lemma. Suppose $v$ is a node of the tree $T$ satisfying $N(v) \leq n/2$. Let $u$ be a centroid of $T$. If $u = v$, the sufficient condition is proved. We thus consider the case in which $u \neq v$. Let $x_1, \cdots, x_d$ be all neighboring nodes of $u$. $v$ must be in one of the connected components of $T - u$, say $C(x_{i_0}, u)$. Let $y_1, \cdots, y_k = v$ be all neighboring nodes of $v$. Let $y_{j_0} \neq v$ be the neighboring node of $v$ on the path from $x_{i_0}$ to $v$. From $N(v) \leq n/2$, we know that $C(y_{j_0}, v) \leq n/2$, and then $C(v, y_{j_0}) \geq n/2$. Because $C(v, y_{j_0})$ is a subtree of the component $C(x_{i_0}, u)$, we know that $C(x_{i_0}, u) \geq n/2$. Thus, $N(u) \geq n/2$. Thus,

$N(u) \geq N(v)$. Therefore, since $u$ is a centroid of $T$, $v$ must also be a centroid of $T$ and $N(u) = N(v)$. This completes the proof of the sufficient condition of the lemma. □

Lemma 2.2 says a node $v$ is a centroid of $T$ if no connected component induced by removing $v$ from $T$ contains more than $n/2$ nodes.

We now describe the algorithm. Without loss of generality, we let the tree $T$ be rooted at an arbitrary node $r$. We denote by $K(i)$ the number of nodes in the subtree rooted at $i$. Then it is easy to see the following:

1. $K(i) = 1$ if $i$ is a leaf, and
2. $K(i) = \sum_{\text{c: child of i}} K(c) + 1$ if $i$ is not a leaf.

The algorithm computes $K(i)$s by proceeding from the leaves of the tree towards the root. One may start from any leaf. But by rule, one is only allowed to use rules (1) and (2) to compute $K(i)$s (This is called the bottom-up manner).

## The algorithm

*Compute the $K(i)$s in the above defined bottom-up manner until a node $v$ is reached such that $K(v) \geq n/2$. Node $v$ is a centroid of $T$. If $K(v) = n/2$, $v$'s father is another centroid of $T$.*

## The cost

We assume that the representation of the tree allows us to access each leaf of the tree in constant time and any node can be reached from any of it's children in constant time. We note that it is easy to build a linked representation of the tree that will have these desired properties in linear time and space. Then in the worst case, the algorithm needs to visit each node of the tree just once. *The worst case occurs only when the sole centroid of the tree is also the root of the tree.*

We could use, for instance, the most common *left-child, right-brother* representation of a tree. In this representation, each node $x$ of the tree contains three pointers: 1. *parent*$[x]$ points to the parent of node $x$, 2. *left-child*$[x]$ points to the leftmost child of $x$, and 3. *right-brother*$[x]$ points to the brother of $x$ immediately to the right. Under this representation, the algorithm will enter each node $x$ at most twice: 1. either from its father or from its left-brother, and 2. (when $x$ is a nonterminal node) from one of its children. So if the *left-child, right-brother* representation of a tree is used, the algorithm needs at most $2n - f$ node visits where $f$ denotes the number of leaves of the tree. Note that this implementation of the algorithm does not make use of the assumption that at any point one can access the leaves of the tree in constant time. This is why this implementation may visit some nodes of the tree more than once (but at most twice). If we augment the *left-child, right-brother* representation of a tree with an array of pointers each pointing to a leaf node of the tree, the above algorithms only needs to visit each node of the tree *at most* once.

Megiddo's algorithm needs first to traverse the tree to compute some function whose definition is similar to that of $K(i)$ for each node $i$, then looks for the centroid along a "right" path of the tree. That is, it need at least $2n - f$ steps if the *left-child, right-brother* representation of the tree is used. While the idea of Goldman's algorithm is similar to that of ours, Goldman's algorithm requires an extra copy of the tree to work on. It deletes in some way the nodes of the extra tree until there is only one node left; this remaining node is then a centroid of the tree (see [KH79] for another version of Goldman's algorithm).

### The correctness of the algorithm

If $v$ is the first node we encountered in the course of computing the $k(i)$'s in the bottom-up manner such that $k(v) \geq n/2$, then $N(v) \leq n/2$. The correctness of the algorithm then follows from Lemma 2.2.

## 3 Centroid trees

**Definition 3.1 (centroid tree)** *A centroid tree $U$ of another tree $T$ is defined on the same set of nodes of $T$: the root $v$ of $U$ is a centroid of $T$, and the subtrees of $v$ (in $U$) are the centroid trees of the connected components of $T - v$ and $v$ (in $U$) is connected to the roots of these (sub-)centroid trees.*

We sometimes use $U(T)$ to denote a centroid tree of another tree $T$. Note that $U(T) \neq T$ in general but $U(U(T)) = U(T)$. So different trees may have the same centroid tree. Lemma 3.2 shows a nice property of the centroid tree, which motivated our work of searching for efficient methods for constructing centroid trees.

**Lemma 3.2** *For any tree $T$ with $n$ nodes, the height of its centroid tree $U$ is $O(\log n)$.*

**Proof** Each node except the leaves in $U$ has at least two children; by Lemma 2.2 the number of nodes in any branch at any node $v$ in $U$ is no more than half the number of nodes in the subtree rooted at $v$ in $T$. So the height of $U$ cannot exceed the height of a complete binary tree with the same number of nodes, which is $\lfloor \log_2 n \rfloor$. □

A straightforward approach to the construction of a centroid tree is to repeatedly call the centroid finding algorithm discussed in the previous section. This approach requires $O(n \log n)$ time. There are many ways to speed up this approach. However, it is not clear whether it is possible to asymptotically improve the time complexity of this naive approach. Let's call this simple approach Algorithm *Naive*.

The following simple observations may help us to gain more insight into the centroid tree construction problem.

**Lemma 3.3** *Let $u$ be any node of the tree $T$. If the sizes of all connected components of $T - u$ are less than or equal to $n/2$, then $u$ is a centroid of $T$. Otherwise, the centroid of $T$ must be in the maximal component of $T - u$.*

**Proof** The correctness follows from Lemma 2.2. □

**Lemma 3.4** *If a centroid $v$ of the tree $T$ is in a subtree $S$ of $T$, then $v$ must lie on the path $s, \cdots, u$ or lie on the path $s, \cdots, u, u'$ where $s$ denotes the root of $S$, $u$ is a centroid of $S$ and $u'$ is a child of $u$ (with respect to the root $s$). In the latter case, both $u$ and $u'$ are the centroids of $T$.*

**Proof** Let $v$ be a centroid of $T$. Suppose that $v$ is not on the path $s, \cdots, u$. Then there are two cases to consider.

In the first case, $v$'s father $f$ $(v \neq f)$ is on a path $f', \cdots, f, v$ such that $f'$ is on the path $s, \cdots, u$ (it is possible that $f = f'$). Since $v$ is a centroid of $T$, $|C(f, v)| \leq n/2$. Thus $C(v, f)$ (the subtree rooted at $v$) contains at least $n/2$ nodes. Then the connected component of $S$ that consists of $C(v, f)$ and the path $s, \cdots, f$ contains at least $n/2 + 1$ nodes. Therefore, by Lemma 2.2, $u$ cannot be a centroid of the subtree $S$, which leads to a contradiction.

In the second case, $v$ is a descendant of $u'$ and $v \neq u'$ where $u'$ is a child of $u$ (it is possible that u'=u). Since $v$ is a centroid of $T$, $C(v, u')$ (the subtree rooted at $v$) has at least $n/2$ nodes. We need to consider two subcases: **a.** $C(v, u')$ has exactly $n/2$ nodes. Then by Lemma 2.2, $u'$ is another centroid of $T$. It is easy to see that $u = u'$. Otherwise, the subtree rooted at $u'$ contains at least $n/2+1$ nodes and therefore, $u$ is not a centroid of the subtree $S$, which is a contradiction. **b.** $C(v, u')$ has more than $n/2$ nodes. This means a branch of $u$ that contains $u'$ has more than $n/2 + 1$ nodes. Thus, $u$ cannot be a centroid of the subtree $S$, which is also a contradiction.

This completes the proof of Lemma 3.4. □

**Lemma 3.5** *Let $s_1$ and $s_2$ be any two neighboring nodes of the tree $T$ with $|C(s_1, s_2)| = n_1$, $|C(s_2, s_1)| = n_2$ and $n_2 > n_1$. Let $u$ be a centroid of $C(s_2, s_1)$ and let $n_3$ denote the number of nodes of the subtree rooted at $u$ of $T$. If the $K(i)$s of all nodes $i$ of $T$ are known, we need at most* $\min(n/2 - n_1, n/2 - n_3)$ *steps each of which takes constant time to find a centroid of the entire tree $T$.*

**Proof** Let $v$ be a centroid of the tree $T$. By Lemma 3.4, $v$ must lie on the path $s_2, \cdots, u$. We can check the nodes on the path one by one until we finally reach a centroid of $T$. The connected component of $T - v$ that contains $s_2$ has at most $n/2$ nodes; so if we proceed from $s_2$ towards $u$ we need at most $n/2 - n_1$ steps before we reach a centroid of $T$. The connected component of $T - v$ that contains $u$ has at most $n/2$ nodes; so if we proceed from $u$ towards $s_2$ we need at most $n/2 - n_3$ steps. In either of these two directions, each step takes constant time because the $K(i)$s of all nodes $i$ of $T$ are known. □

We have modified Algorithm *Naive* by making use of Lemmas 3.3, 3.4 and 3.5. The resulting algorithm is called Algorithm *Heuristic.* We have applied Algorithm *Heuristic* to several random trees. The preliminary experimental results showed that Algorithm *Heuristic* constructed a centroid tree for a given random tree in time proportional to the number of nodes in the tree on the average. However, at we are unable to prove this behavior of Algorithm *Heuristic* rigorously.

## 4 Application to string processing

In this section we make use of the properties of the centroid tree to solve *the longest common substring (LCS)* problem. The problem is, given a string $S$ (the text) of $n$ characters and a string $P$ (the pattern) of $m$ characters over some finite alphabet $\Sigma$, to find the longest substring which occurs in both of the two strings. An efficient solution to the problem can be useful for homology searching in nucleotide/protein sequence databases [Wat89], in the editing distance problem, in the multiple pattern searching problem [Per93], etc. We are particularly interested in the case of the problem in which the text is given in advance and is fixed, and many queries with different patterns will be made later.

Three algorithms for the LCS problem are previously known (named algorithms *P1*, *P2*, and *P3* respectively) [Per93]. It is also possible to solve the problem by constructing a suffix tree for the concatenation of the two strings and then marking each node of the suffix tree that has leaves from both of the two strings in its subtree. Let's name this algorithm *Cat*. In the following we will propose a new algorithm for the problem. Table 1 shows the time and space bounds of the previously known algorithms compared with this new algorithm (named Algorithm *New*).

Table 1: Comparison of the LCS algorithms

| Algorithm | Preprocessing | | Searching time | |
|---|---|---|---|---|
| | space | time | worst case | average |
| P1 | $m\,\lvert\Sigma\rvert$ | $m\,\lvert\Sigma\rvert + m^2$ | $n$ | |
| P2 | $\lvert\Sigma\rvert$ | $\lvert\Sigma\rvert + m$ | $m\,n$ | $n \log n$ |
| P3 | $m + \lvert\Sigma\rvert$ | $2\,m + \lvert\Sigma\rvert$ | $m\,n$ | $(1 + \frac{m}{\lvert\Sigma\rvert})\,n$ |
| Cat | | | $m + n$ | |
| New | $n$ | $n \log n$ | $m \log n$ | |

A weakness of Algorithm *P1* is that it requires large amounts of space and preprocessing time for large alphabets and/or patterns. Algorithm *P2* requires that the size of the pattern be no more than the size of a word of the machine on which the algorithm is executed. When the size of the underlying alphabet is quite small, e.g., $|\Sigma| = 4$ in the case of DNA applications, the average-case performance of Algorithm *P3* deteriorates to its worst-case performance. While Algorithm *Cat* runs in $O(n+m)$ time, it is not proper for applications in which the text is very large and fixed and one wishes to search the text with many different shorter patterns ($n \gg m$). This is because although the text is fixed and static for many queries, for each new query (new pattern) Algorithm *Cat* has to rebuild a suffix tree for the text and the pattern which takes as much as $O(n + m)$ time. For example, a DNA sequence of a human being may have up to $3 \times 10^9$ nucleotides and a typical pattern sequence may have a few hundreds to thousands nucleotides. In such cases, $m + n \gg m \log n$, the time needed by our new algorithm to answer a query.

The new algorithm finds the longest prefix of each of the suffixes of the pattern $P$ in the text $S$. Note that $P$ has $m$ suffixes and therefore there are at most $m$ longest prefixes (of the suffixes) that appear in $T$. The algorithm then simply choose the longest one from these prefixes found as an answer to the LCS problem. It requires $O(n \log n)$ time and $O(n)$ space to preprocess the text. After the preprocessing, a query can be answered in $O(m \log n)$ time. An advantage of this approach is that in cases where the text is large (e.g., $n > m \log n$) and static for many queries, we only have to preprocess the text once; after the text has been preprocessed, a query can be answered quickly. It is a probabilistic algorithm and there is a small chance of error. That is, the algorithm may claim that a substring of the pattern is equal to a substring of the text while they are not equal at all (This is called a "false match"). However, as will be seen later, the probability of a false match can be made arbitrarily (inverse-polynomially) small.

The general structure of the algorithm is as follows:

- Preprocessing stage
  - construct a suffix tree $T$ for the text $S$
  - construct a centroid tree $U$ for the suffix tree $T$
- Searching stage
  - search the centroid tree $U$ for locations of the longest prefix of each of the suffixes of the pattern $P$ in the text $T$

Now, we describe the algorithm in detail. Since algorithms for building suffix trees in linear time and space are known in the literature [Wei73, McC76, Ukk95] and we have already presented an algorithm for building the centroid tree (in Section 3), we will concentrate on the searching stage of the algorithm.

Let the text be $S = S[1] \cdots S[n]$ and let the pattern be $P = P[1] \cdots P[m]$. We use a suffix tree to represent the text. Assuming that the suffix tree $T$ of the text $S$ and a centroid tree $U$ of $T$ are already available, our search algorithm searches the trees for the occurrences of the pattern in the text.

Let $w$ be the end node of the path that the pattern $P$ determines in $T$. If $P$ is not a substring of $S$, then we define the end node $w$ to be the node that corresponds to the longest prefix of $P$ that is a substring of $S$. Our goal is to find $w$.

We maintain the following variables:

- $v$, the current node in $U$; $v$ is a centroid of some connected component $C$ of $T$.
- $u$, the topmost node of $C$ (in $T$); the substring corresponding to $u$ is the longest substring of $S$ found so far that is a prefix of $P$.
- $i$, an index to $P$ such that $P[1], \cdots, P[i]$ determines the path from the root to $u$.
- $j$, the length of the substring determined by the path from $u$ to $v$.
- $k$, a pointer to $S$ that corresponds to the end position of the substring determined by the path from the root to $v$.

Furthermore, let $x$ be any node of $T$. We denote by $x.length$ the length of the substring determined by the path from the root to node $x$ and denote by $x.end$ an index to $S$ that corresponds to the end position of this substring in $S$. Note that by assumption, $x.length$ and $x.end$ are already stored in each node $x$ on construction of the suffix tree.

Given $u$ and $v$ computing $j$ and $k$ is easy:

$$\begin{aligned} j &:= v.length - u.length; \\ k &:= v.end. \end{aligned} \tag{15}$$

Initially, $u :=$ the root of $T$; $v :=$ the root of $U$; $i := 0$; and $j$ and $k$ are computed by (15).

In order to find $w$ efficiently we need to find a way to decide quickly which of the connected components induced by removing $v$ from $T$ contains $w$. There are two possibilities: $w$ is in the component that is "above" $v$ or $w$ is in one of the components that are "below" $v$. We notice that $w$ is a descendant of $v$ if and only if $S[k-j+1]\cdots S[k] = P[i+1]\cdots P[i+j]$. If $w$ is in the component "above" $v$, we assign the centroid of that component to $v$ and $u$ is unchanged; if we know which of the components "below" $v$ contains $w$, we assign the root of that component to $u$ and assign the centroid of that component to $v$. The above ideas are precisely presented in **procedure** *search* in Figure 1. **procedure** *search* finds and stores $w$ in its variable $v$ and stores the index to $P$ referring to the end position of the longest prefix of $P$ that is equal to a substring of $S$ in its variable $i$ when executed with $u$ being initialized to be the root of $T$, $v$ being initialized to be the root of $U$ and $i$ being initialized to be 0.

The question that is crucial to implement **procedure** *search* efficiently is: Given a substring $S[k]\cdots S[k+j]$ of $S$ and a substring $P[i]\cdots P[i+j]$ of $P$, how can we answer quickly whether they are equal or not? There is a probabilistic method [Nao91, KR87] which, after preprocessing the strings $S$ and $P$ in linear time and space, can test whether a substring of $S$ is equal to a substring of $P$ in constant time. There is a probability of error (a false match) in any test. But the probability of a false match can be made arbitrarily (inverse polynomially) small.

The method needs a prime $q$ which is chosen at random from a set of primes smaller than $M$ (to be stated soon). It is this prime $q$ that may lead to a false match. By Theorem 3 of [KR87] the probability of a false match is less than $\pi(n)/\pi(M)$ where $\pi(n)$ denotes the number of primes smaller than $n$. By Lemma 2 of [KR87] $\frac{u}{\ln u} \leq \pi(u) \leq 1.25506\frac{u}{\ln u}$. Thus, for example, if we choose $M$ to be $n^3 \log n$, the probability of a false match is (asymptotically) $1/n^2 \log n$.

We now look at the complexity of **procedure** *search*. Note that at each step $v$ is assigned to one of its children (in $U$). By Lemma 3.2 the height of $U$ is $O(\log n)$. So **procedure** *search* requires $O(\log n)$ steps. From the above discussion, each step takes constant time. So **procedure** *search* needs $O(\log n)$ time to find the longest prefix of $P$ that appears in $S$.

To solve the whole LCS problem, for every suffix $P_i = P[i]\cdots P[m]$ $(i = 1, \cdots, m)$ we find the longest prefix of $P_i$ that appears in $S$ with **procedure** *search*. From among all these (locally) longest prefixes found, we choose the (globally) longest one as an answer to the LCS problem. All this takes $O(m \log n)$ time.

To summarize, our algorithm for the LCS problem consists of:

- Preprocessing the text

```
procedure $search$(node: $u$, $v$; integer: $i$);
integer: $j$, $k$;
begin
  $j := v.length - u.length$;
  $k := v.end$;
  if $S[k-j+1] \dots S[k] = P[i+1] \dots P[i+j]$ then /* $j$ may be 0 */
    if $i+j=m$ then $i := m$; stop
    /* $P$ is equal to the substring of $S$ corresponding to node $v$ */
    else
      if there exists a child $c$ of $v$ in $T$ corresponding to the symbol
       $P[i+j+1]$ then
        if the substring $S[k+1] \dots S[k+l]$ of $S$ corresponding to the edge
        $(v, c)$ is equal to a substring of $P$ starting at $P[i+j+1]$ then
          $u := c$;
          $v := v$'s child in $U$ corresponding to the subtree rooted at $c$ in $T$;
          $i := i+j+l$;
          $search(u, v, i)$
        else
          /* Let $L$ denote the maximal $x$ in $[1, l]$ such that
          $S[k+1] \dots S[k+x] = P[i+j+1] \dots P[i+j+x]$ */
          find $L$ with binary search supported with the substring equality
          testing technique;
          $i := i+j+L$; stop
          /* $P[1] ... P[i]$ is the longest prefix of $P$ that is equal to a
          substring of $S$; this substring is the concatenation of the
          substring corresponding to node $v$ and $S[k+1] \dots S[k+L]$ */
        end
      else
        $i := i+j$; stop
        /* $P[1] \dots P[i]$ is the longest prefix of $P$ that is equal to a
        substring of $S$; this substring corresponds to node $v$ */
      end
    end
  else
    if there exists a child of $u$ in $T$ corresponding to $P[i+1]$ then
      $v := v$'s child in $U$ corresponding to the component "above" $v$;
      $search(u, v, i)$
    else
      $v := u$; stop
      /* $P[1] \dots P[i]$ is the longest prefix of $P$ that is equal to a
        substring of $S$; this substring corresponds to node $v$ */
  end
end
```

Figure 1: Search for end node of path determined by pattern.

- construct a suffix tree $T$ for the text $S$ in $O(n)$ time and space.
- construct a centroid tree $U$ for the suffix tree $T$ in $O(n \log n)$ time and using $O(n)$ space.
- process the text $S$ in order to be able to check quickly the substring equality. This takes $O(n)$ time and space.

• Searching for the pattern

- process the pattern $P$ in order to be able to check quickly the substring equality. This takes $O(m)$ time and space.
- search the centroid tree $U$ for locations of the longest prefixes of all the suffixes of the pattern $P$ in the text $S$ in $O(m \log n)$ time and $O(1)$ space.

That is, the preprocessing takes $O(n \log n)$ time and $O(n)$ space and the searching takes $O(m \log n)$ time and $O(m)$ extra space.

To make this algorithm error free, we can add a step that checks whether a claimed match is *true* or *false*. If the claimed longest match is false, we discard it and check the second longest match, and so on, until we reach a *true* match. Since the probability of a false match can be made arbitrarily (inverse-polynomially) small without asymptotically increasing the time and space requirements of the algorithm, the chance of using this checking step can be made arbitrarily inverse-polynomially small as well.

## 5 Open questions

It is of considerable interest to either establish that there exists a non-linear lower bound on the run time of all algorithms which construct a centroid tree for any given tree, or to exhibit an algorithm whose run time is $O(n)$.

It is also interesting, at least from a practical point of view, to find centroid tree construction algorithms that run in linear time on the average and require linear space even if their worst-case behavior could be much worse. Are there any deterministic algorithms to do the search (as discussed in Section 4) using the same order of time as the probabilistic one does?

## References

[FJ80] G.N. Frederichson and D.B. Johnson, Generating and searching sets induced by networks, *Proc. of the 7th International Colloquium on Automata, Languages and Programming*, LNCS 85, July 1980.

[Gol71] A.J. Goldman, Optimal center location in simple networks, *Trans. Sci.*, 3(1971).

[Har69] F. Harary, *Graph Theory*, Addison-Wesley, Mass., 1969.

[KH79] O. Kariv and S.L. Hakim, An algorithmic approach to network location problems. I: The $p$-centers, *SIAM J. appl. Math.*, Vol 37, No.3, Dec. 1979.

[Knu73] D.E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. Addison-Wesley, Reading, Mass, 1973, Ch. 6.3, pp. 490-493.

[KR87] R. Karp and M. Rabin, Efficient Randomized Pattern Matching Algorithms. *IBM J. Res. Develop.*, Vol. 31, No. 2, March 1987.

[McC76] E.M. McCreight, A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM* 23 (1976), 262-272.

[MTZC81] N. Megiddo, A. Tamir, E. Zemel and R. Chandrasekaran, An $O(n \log^2 n)$ algorithm for the $k$th longest path in a tree with applications to location problems, *SIAM J. Comput.*, Vol.10, No.2, May 1981.

[Nao91] M. Naor, String matching with preprocessing of text and pattern, *Proc. of the 18th International Colloquium on Automata, Languages and Programming*, Madrid, July 1991, pp.739-750.

[Per93] C.H. Perleberg, Three Longest Substring Algorithms, *Proc. First South American Workshop on Strong Processing*, Belo Horizonte, Brazil, 1993, eds. R. Baeza-Yates and N. Ziviani.

[Sla82] P.J. Slater, Locating Central Paths in a graph,*Trans. Sci.*, Vol.16, No.1, Feb. 1982.

[Ukk95] E. Ukkonen, On-line construction of suffix-trees. *Algorithmica* (1995) 14: 249-260, Springer-Verlag, New York.

[Wat89] M.S. Waterman (ed.), *Mathematical Methods for DNA Sequences*, CRC Press 1989, Boca Raton, Florida.

[Wei73] P. Weiner, Linear pattern matching algorithm, *Proc. 14th IEEE Symp. on Switching and Automata Theory*, 1973, pp. 1-11.