JASS 2008

Trees - the ubiquitous structure in computer science and mathematics

Suffix Trees

Caroline Löbhard

St. Petersburg, 9.3. - 19.3. 2008

Contents

1 Introduction to Suffix Trees

A suffix tree is a tree-like data-structure for strings, which affords fast algorithms to find all occurrences of substrings. A given String S is preprocessed in $O(|S|)$ time. Afterwards, for any other string P, one can decide in $O(|P|)$ time, whether P can be found in S and denounce all its exact positions in S. This linear worst case time bound depending only on the length of the (shorter) string $|P|$ is special and important for suffix trees since an amount of applications of string processing has to deal with large strings S.

1.1 Basics

In this paper, we will denote the fixed alphabet with Σ, single characters with lower-case letters $x, y, ...$, strings over Σ with upper-case or Greek letters $P, S, ..., \alpha, \sigma, \tau, ...$, Trees with script letters $\mathcal{T}, ...$ and inner nodes of trees (that is, all nodes despite of root and leaves) with lower-case letters $u, v, ...$

For a string $S \in \Sigma^*$ and $i, j \in \{1, ..., |S|\}, i \leq j$ the substring of S from position i to j is written as $S[i, j]$, the single character at position i as $S[i]$.

Definition 1 (Suffix tree) Given a string S of length $|S| = m$ over a fixed alphabet Σ, the *suffix tree* $\mathcal{T}_S$ *of* S is a rooted directed tree with

- edges labelled with nonempty strings and
- exactly m leaves labelled with the integers from 1 to m,

such that

- each internal node other than root has at least two children
- no two edges out of one node have edge labels beginning with the same character and
- for any leaf i, the concatenation of the path labels from root to leaf i is $S[i, m]$.

Example 1 As an example, we build the suffix tree of the string $S = bananas$ using a naive algorithm that would run in $O(|S|^3)$ time and therefore would never be used in practice.

Trying to build the suffix tree for nearly the same string, namely just omitting the last character s one immediately finds, that this is not possible. The paths of all suffixes that do also occur as substrings that are no suffixes of S (in our example a, an and ana)

do not end in leaves. To avoid this undesirable situation, we add a character \$ to the end of S that does not lie in the alphabet Σ. Instead of constructing the suffix tree for *banana* we will construct the suffix tree for *banana*\$.

A few more notions walking trough trees:

The *label of a path in* $\mathcal{T}$ is the concatenation of the labels of edges passed when following the path. A path does not need to end in a node and it does not need to start at root.

For any node u of $\mathcal{T}$ there is a unique path in the suffix tree from root to node u.
We denote the label of that path with $path(\mathcal{T}, u)$ and the number of nodes in that path with $|node - depth(u)|$.

Before we turn our attention to the main purpose of this paper, that is, to explain the most important algorithms for the construction of suffix trees, we leave a few words on the benefits of this preprocessing when dealing with strings and especially with the substring problem.

1.2 Getting a first feeling for the nice structure of suffix trees

Theorem 1 *Suppose the suffix tree $\mathcal{T}_S$ of a string S, and another string P are given.*

(1) P matches a path in $\mathcal{T}_S$ from root if and only if P occurs in S.

(2) If P occurs in S, then it can be found exactly at the positions numbered with the labels of all leaves of the subtree below the point of the last match in $\mathcal{T}_S$.

proof. (1) The paths in $\mathcal{T}_S$ from root to the leaves spell out exactly the suffixes of S. So P matches the beginning of such a path if and only if P is a prefix of all suffixes of S that end in the leaves below the end of that path.
(2) We labelled the leaves in a way such that for each i, the suffix $S[i, m]$ ends in the leaf with label i. So if M is the set of leaf-labels below the end of the path that matches P, the string P is prefix of the suffixes $\{S[i, m] \mid i \in M\}$ of S and this means, that P occurs in S exactly at the positions M. qed.

Theorem 2 *Let $S \in \Sigma^*$ be a string over the alphabet Σ and let $\mathcal{T}$ be the suffix tree for S\$. If v is a node in the suffix tree labelled with $path(\mathcal{T}, v) = x\alpha$ where $x \in \Sigma$ is a single character, then there is (exactly one) node u in the tree labelled with $path(\mathcal{T}, u) = \alpha$.*

proof. Each internal node of $\mathcal{T}$ has at least two children and so has v. No two edges out of one node have edge labels beginning with the same character. It follows that there are at least two suffixes of S, say $S[i, m]$ and $S[j, m]$ $(i \neq j)$ such that

$$S[i, m] = x\alpha\sigma \text{ and } S[j, m] = x\alpha\tau \text{ with } \sigma[1] \neq \tau[1].$$

Since $\mathcal{T}$ encodes all suffixes of S, it contains paths from root to a leaf for $S[i+1,m]$ and $S[j+1,m]$. But we have $S[i+1,m] = \alpha\sigma$ and $S[j+1,m] = \alpha\tau$, that is, we have a node u with $path(\mathcal{T},u) = \alpha$. qed.

Theorem 2 provides us with a map s on the internal nodes of a suffix tree that maps a node v with path label $x\alpha$ to the node $s(v) = u$ with path label α. All algorithms that we mention here despite of the one from Farach save these so called *suffix-links* for every node and use them move around in the tree quickly.

Theorem 3 *If the edge labels in a suffix trees $\mathcal{T}$ are written explicitly on the edges, then it is not possible to give a linear-time algorithm that constructs that tree for any String over a fixed or arbitrary alphabet containing at least two characters.*

proof. The amount of characters to be written can just grow to large. qed.

Theorem 3 sounds a little negative since we are looking for linear-time algorithms. But the problem can be easily solved: All the algorithms in this presentation use a pair of indices (i,j) to represent the edge-labels in the way that the pair (i,j) stands for the substring $S[i,j]$ of S. This representation is pretty natural and we will see in section 2 that for example for most of the work to be done in Ukkonen's algorithm, the explicit characters are not even used.

1.3 A historical overview of algorithms

The first linear-time algorithm for constructing suffix trees of strings over a fixed alphabet was given by Weiner in 1973. Weiner starts with the tail of the String $S[n]\$$ and then successively matches the suffixes $S[j,n]\$$ (j descending from $n-1$ down to 1) into the tree. His algorithm keeps two vectors at each non-leaf node where some kind of suffix link information is stored. Each vector is of length equal to the size of the alphabet and so one can imagine, that Weiner's algorithm is not very space-economic. In 1976, McCreight [8] improved this aspect suggesting a more space-economic version of Weiner's algorithm. He manages to avoid storing the possibly large vectors with suffix link information and builds up the suffix tree beginning with the entire String $S\$$ and matching the suffixes $S[j,n]\$$ (j increasing from 1 to n) into the tree.

Nearly 20 years passed by before Ukkonen [12] could give a completely new algorithm that was able to compute a suffix tree on-line, that is, to process the string symbol by symbol from left to right, providing the user in each step with a suffix tree for the already scanned part of the string. His algorithm runs in linear time and turned out to have a close connection to McCreight's algorithm. A more educational advantage of this algorithm is its understandability which gave a reason for us to leave McCreight's algorithm nearly undiscussed and rather concentrate on Ukkonen's algorithm (see section 2).

Nevertheless, something remained to be done: The constant appearing in the achieved linear time-bound in Ukkonen's algorithm still depends on the logarithm of the size $|\Sigma|$ of the Alphabet. So for an unknown alphabet, the algorithm does not match that linear bound anymore.

In the year 1995, the same year when Ukkonen published his algorithm, Delcher and Kosaraju [9] published an algorithm that seemed to solve this still open problem of constructing suffix trees of strings S over an arbitrary alphabet in $O(|S|)$ time. Unfortunately, their time-analysis was incorrect and the linear worst-case time bound could not be maintained ([10]).

Finally, in 1997, Farach [7] closed the gap in giving an algorithm that matched exactly Delcher's and Kosaraju's conditions.

2 Ukkonen's on-line space-economic linear-time algorithm

The motivation for us to have a close look at Ukkonen's algorithm is it's convenience which is nevertheless not Ukkonen's effort. Daniel Gusfield is the first to describe the algorithm, proof its correctness and investigate its time-analysis in a really simple way in his Textbook [11] in 1997. We will follow his suggested way of at first giving the algorithm as a simple but inefficient method and then improving the worst-case running-time stepwise by the use of simple implementation tricks.

2.1 High-level description

Definition 2 (Implicit suffix tree) An *implicit suffix tree of String S* is the tree obtained from the suffix tree of $S\$$ by removing the \$-symbols from the edge-labels, afterwards removing all edges without label and at last removing all nodes with less than two children.
Given a string S over a fixed alphabet Σ let $\mathcal{I}_i$ be the *implicit suffix tree for the prefix $S[1,i]$ of S*.

The construct the right suffix tree from the implicit one, let the algorithm continue with $S\$$ and for each leave, assign ($m-$ length of the path from root to that leave) as its label.

Ukkonen's algorithm runs in m *phases*, where m is the length of the input-string S. In phase 1, $\mathcal{I}_1$ is constructed and afterwards, in Phase $i+1$, tree $\mathcal{I}_{i+1}$ is constructed from $\mathcal{I}_i$. Each phase $i+1$ is further divided into $i+1$ *extensions*, one for each of the $i+1$ suffixes of $S[1..i+1]$. In extension j of phase $i+1$, the suffix $S[1..j]$ is fit into the tree. Procedurally, the algorithm is as follows:

Algorithm 1 (High-level Ukkonen algorithm)

```
Construct tree I_1
for i from 1 to m − 1 do
  for j from 1 to i + 1 do
    Find the end of path with label S[j, i]
    (exists because we have already done phase i)
    if ⓐ there is a path with label S[j, i + 1] in I_i then
      - nothing to do -
    else
      Append S[i + 1] to I_i due to the following rules:
      if ⓑ S[i, j] ends at a leaf then
        Append S[i + 1] to the leaf-edge
      else if ⓒ S[i, j] ends at an inner node then
        Add an edge labelled with S[i + 1] and a leaf labelled j to that node
      else if ⓓ S[i, j] ends in the middle of an edge-label then
        Add a node to the end-position
        Add an edge labelled with S[i + 1] and a leaf labelled j to that node
      end if
    end if
  end for
end for
```

For fixed alphabet, this runs in $O(|S|^3)$ time.

2.2 Using suffix links

The first extension of each phase is special and easy:
Since the path labelled $S[1, i]$ ends in leaf 1, we just have to append $S[i + 1]$ to the leaf-edge due to case ⓑ in the above algorithm. This can be done in constant time when a pointer has been left on leaf 1.)

Assume that extension $j - 1$ of phase $i + 1$ is just finished. This means that the implicit suffix tree $\mathcal{I}_i$ is constructed, the strings $S[k, i + 1]$ ($k \in \{1, ..., j - 1\}$) are inserted, most recently, we inserted $S[j - 1, i + 1]$, and now, we want to insert $S[j, i + 1]$.

Using suffix links and Theorem 2, one can do that as follows:

Algorithm 2 (Single Extension Algorithm)

1: Take the largest $k \in \{j-1, ..., i\}$ and $v \in V \cup \{root\}$ with $path(\mathcal{I}, v) = S[j-1, k]$ walking up at most one edge.
2: If v is an inner node and therefor has a suffix link, traverse this suffix link from v to $s(v)$.
If $v = root$, stay there.
3: Walk down the path $S[k, i]$ and
4: insert $S[i+1]$ due to the rules ⓐ, ⓑ, ⓒ
5: If a new internal node w was created in extension $j-1$, create the corresponding suffix link $(w, s(w))$ from w to the node where $S[j, i]$ ends.

Obviously, we take use of memory to save the time for matching the paths from root to suffix links again and again. The average runtime will improve, but the worst case time bound is still $O(n^3)$ $(n = |S|)$.

The problem lies in step 3. While steps 1,2,4 and 5 only take constant time, matching $S[k, i]$ into the tree $\mathcal{I}_i$ can take up to $3n^2$ operations per phase when it is done naively: Traversing one edge can cost up to $O(n)$ operations. In each extension, the node-depth of the current node is decreased by at most 2 (in step 1 it can be decreased by 1, following a suffix link can decrease it once more by 1). Since there is no node with node-depth more than n, the worst case will require $2n + n = 3n = O(n)$ edge-traversals. The total time required in step 3 is then n phases times $3n$ edge-traversals times n operations during the traversal, that is $O(n^3)$.

In the following section, we will improve step 2.

2.3 Edge-label compression and the skip/count trick

In this section, we will show, how the worst case time bound in Ukkonen's algorithm can be decreased to $O(|S|^2)$. To achieve this time bound, we will do further improvements on step 3 in the algorithm 2 for one extension j of one phase $i+1$.

Remembering Theorem 3 about edge label compression, the first thing to do is writing pairs of indices (g, h) on the edges instead of writing the full strings. In line 3 of Algorithm 2, the string $S[k, i]$ is matched into the tree $\mathcal{I}_i$ character by character following a path down from $s(v)$. But as $S[k, i]$ must appear in that tree and no two paths out of $s(v)$ start with the same character, it is totally enough to just find the edge (g, h) out of $s(v)$ such that $S[e] = S[k]$. If the number of characters $h - g + 1$ on this edge is less than the length $i - k + 1$ of $S[k, i]$, we jump to the next node without paying any attention to the exact characters of $S[k, i]$ and continue matching $S[k + h - g]$ into $\mathcal{I}_i$.
Algorithmically, line 3 of Algorithm 2 then looks as follows:

Algorithm 3 (Single Extension Line 3 with Skip/Count Trick)

1: Find edge (g, h) with $S[e] = S[k]$
2: **while** $h - g < i - k$ **do**
3: Jump to next node
4: $k := k + h - g$
5: Find edge (g, h) with $S[e] = S[k]$
6: **end while**

This approach ensures a constant time bound for walking down one edge and, reminding the time analysis in section 2.2, we can achieve a quadratic time bound for the hole algorithm.

2.4 Two more observations

At this point, we optimised the single extensions of a single phase as far as possible: It runs in constant time. To get the promised further improvements, the phase boundaries have to be broken.

We present two more implementation tricks. Looking at algorithm 1 and its case differentiation, one can observe:

(1) **once case ⓐ, rest of phase case ⓐ,**
since in case ⓐ, the path labelled with $S[j, i]$ continues with $S[i + 1]$ and so do the paths labelled $S[g, i]$ $(g \in \{j + 1, ..., i + 1\})$.

(2) **once a leaf, always a leaf,**
since in the algorithm, no leaf-edge is ever extended beyond its current leaf.

Gusfield [11] describes in detail how those two observations can be used to achieve a linear worst case time bound.

3 Generalised Suffix Trees

Suffix trees are of great use for many kinds of string matching problems. But the data-structure described in this paper is of course not a universal tool for any string computing task. There are nevertheless many problems that can be handled with slightly adapted suffix tree structures.

An example that is easy as well as very useful are the so called *generalised suffix trees*.

Definition 3 (Generalised suffix tree) A *generalised suffix tree* is a suffix tree containing all suffixes of a set $\{S_1, ..., S_n\}$ of strings.

A lot of applications working on a database of strings use this kind of suffix trees. Such a tree can be built in $O(\sum_{i=1}^{n} |S_i|)$ time. One possible way to do this is appending a different end of string marker to each string in the set, then concatenating all the strings together and building the suffix tree of the concatenated string. The resulting suffix tree will have one leaf for each suffix of the concatenated string, the leaf numbers can easily be converted to two numbers, one identifying a string S_i and the other a starting position in S_i. The edge labels (each represented by a pair of indices as suggested in section 1.2 after Theorem 3) have also to be adjusted and especially the second indices of labels of leaf edges have to be decreased.

There are many other ways of customising suffix trees.

4 Applications of Suffix Trees

Suffix trees are suitable for string matching problems. Imagine there is a large string S, containing for example genomic DNA data and imagine, we stored it as suffix tree. Theorem 1 now shows, that for any other string P, finding its occurrences in the large string S only takes $O(|P|)$ time as we just have to match the short string P into the suffix tree. That is, we are independent of the length of S.

This observation leads to an amount of applications in Bioinformatics. Of course there are a lot of applications that do not require exact substring matching. The possibility of generalising the suffix tree algorithms in a way such that we can build the suffix tree of a set of strings efficiently described in section 4 makes suffix trees even more applicable.

To name some problems that can be handled with the help of suffix trees, we will give a short list:

There is of course the **exact string matching problem**, both for single strings or for databases of strings. As the amount of applications that could take advantage of an efficient string matching algorithm is really large, we will not mention anyone of them here.

Given two Strings S_1 and S_2 one could wish to calculate the **longest common substring** of those Strings. To solve this problem, one can build the generalised suffix tree of $\{S_1, S_2\}$ and mark each leaf that belongs to a suffix of S_1 with 1 and each leaf that belongs to a suffix of S_2 with 2. The leaves marked with both numbers 1 and 2 are the common suffixes of S_1 and S_2 and the longest among them is the longest common substring. Everyone of the mentioned steps can be done in linear time and the algorithm therefor matches a linear worst-case time-bound.

A similar application is the **DNA contamination problem**. Suppose there is a sequenced DNA string of one species contaminated with the short and well known DNA of a host organism, for example bacteria or yeast, or it is contaminated by a fragment

used to incorporate the desired DNA into that host. Substrings of a length more than a given l that equal a part of the contamination string are candidates for unwanted pieces of DNA. So the Problem is: Given a new string S, a known string P and an integer l, find all substrings of P that occur in S and are longer than the specified l.

One way to solve this problem is very similar to the algorithm finding the longest common substring of two strings: Build the generalised suffix tree of $\{S, P\}$, find all common substrings with the marking method explained above and choose the ones that are longer than l.

References

[7] M. Farach-Colton. Optimal Suffix Tree Construction with Large Alphabets. ftp://ftp.cs.rutgers.edu/pub/farach/Suffix.ps.Z

[8] E.M. McCreight: A space-economical suffix tree construction algorithm. Journal of the ACM 23(2):262-272, 1976.

[9] S.R. Kosaraju and A.L. Delcher: Large-scale assembly of DNA strings and space-efficient construction of suffix trees. Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing STOC'95, pp. 169-177, 1995.

[10] S.R. Kosaraju and A.L. Delcher. Correction: Large-scale assembly of DNA strings and space-efficient construction of suffix trees. Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing STOC'96, p. 659, 1996.

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

[12] E. Ukkonen: On-line construction of suffix trees. Algorithmica, vol 14, pp. 249-260, 1995.