Tries and Suffix Trees

Inge Li Gørtz

1 Introduction

In the *String Indexing Problem* we are given a string S over an alphabet Σ. The problem is to preprocess the string S into a data structure such that given a pattern P we efficiently can answer queries of the form *"Return the starting position of all occurrences of P in S"*. The goal is to obtain a data structure that uses little space and supports the queries efficiently. In this note we will se how to solve the tree indexing problem in $O(n)$ and query time $O(m + \text{occ})$, where n is the length of the text S, m is the length of the pattern P, and occ is the number of times P occurs in S.

We will also see how to index a collection of strings $\mathcal{S} = \{S_1, S_2, \ldots, S_s\}$ such that we given a pattern P efficiently can answer queries of the form *"Is P a string in $\mathcal{S}$?"*

Preliminaries

Let S be a string of length n over an alphabet Σ. We use σ to denote the size of the alphabet. Let $S[i \ldots j]$ denote the substring of S starting at position i and ending at position j. A *prefix* of S is a substring of the form $S[0 \ldots j]$ and a *suffix* of S is a substring of the form $S[j \ldots n-1]$.

2 Tries

Let $\mathcal{S} = \{S_1, S_2, \ldots, S_s\}$ be a set of strings from an alphabet Σ. Assume that the set $\mathcal{S}$ is prefix-free, i.e., that no string in $\mathcal{S}$ is a prefix of another string in $\mathcal{S}$[1]. Let n be the total length of the strings in the set $\mathcal{S}$. A trie of the set of strings $\mathcal{S}$ is a rooted ordered tree $T_{\mathcal{S}}$ with the following properties:

- Each edge in $T_{\mathcal{S}}$ is labeled with a character from Σ.
- Edges from a node to its children are sorted from left-to-right alphabetically.
- Each root-to-leaf path represents a string in the set (obtained by concatenating the labels of the edges on the path).
- Common prefixes of two strings share the same path maximally.
- For a node v in $T_{\mathcal{S}}$ all the edges to its children have different labels.

[1] If $\mathcal{S}$ is not prefix-free we can turn it into a prefix-free set by adding \$ in the end of each string in the set, where \$ is a new symbol not already in Σ.

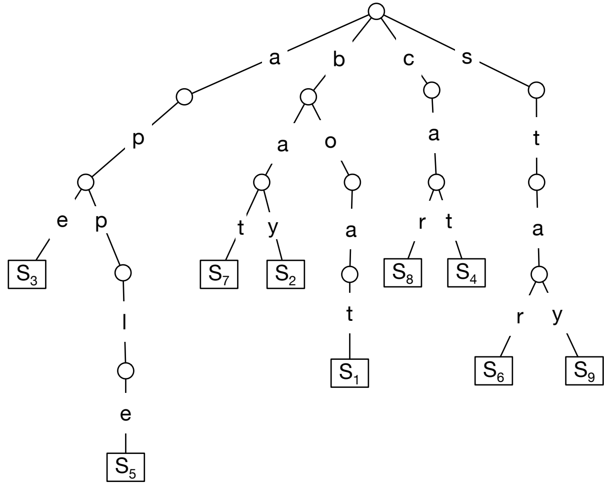


Figure 1: Trie over the set of strings $\mathcal{S} = \{\text{boat, bay, ape, cat, apple, star, bat, car, stay}\}$.

- Each leaf has a label indicating which string in the set it represents.

See Figure 1 for an example.

Each node v in $T_\mathcal{S}$ represents the string obtained by concatenating the labels of the edges on the path from the root to v. We will call this string $\text{str}(v)$. A string represented by a node in T is a prefix of one or more strings in the set $\mathcal{S}$. The number of leaves in a subtree rooted at a node v is equal to the number of strings from the set that has the string $\text{str}(v)$ as a prefix.

Properties The trie $T_\mathcal{S}$ has the following properties:

- The number of nodes in $T_\mathcal{S}$ is $O(n)$.
- The number of leaves in $T_\mathcal{S}$ is s.
- A node in $T_\mathcal{S}$ has at most σ children.
- The height of $T_\mathcal{S}$ is equal to the length of the longest string in $\mathcal{S}$.

2.1 Matching

We can use the trie to search for a pattern P: Start in root and keep following edges according to the next character in P. If we reach a leaf after having read all of P then P is a string in the set $\mathcal{S}$. If at some point during the search we reach a node that does not have an outgoing edge labeled with the next character of P, then P is not a string in $\mathcal{S}$.

The time it takes to search for a pattern P of length m is $O(\sigma m)$. In each node we have to determine which edge to follow. A simple comparison to the character of each edge out of the node gives us $O(\sigma)$ time in each node. In total this gives us a running time of $O(\sigma m)$, where $m = |P|$. For constant size alphabets this gives a search time of $O(m)$.

Lemma 1 *Given a trie over a collection of strings of total length n over an alphabet of size $O(1)$ we can search for a pattern P of length m in $O(m)$ time.*

2.1.1 Prefix Search

We can also use the trie to search for all strings that starts with P, i.e. that has P as a prefix. Simply search for P the same way as before. If the search for P ends in a node v after having read all characters of P then all strings that corresponds to a leaf in the subtree of v starts with the pattern P. We can now return all such strings by traversing the subtree and returning the values stored in the leaves.

Example If we search for $P = st$ in the trie in Figure 1 we can see that S_6 and S_9 both has P as a prefix.

The running time of a prefix search is $O(m + |T_{\mathcal{S}}(v)|)$, where $T_{\mathcal{S}}(v)$ is the subtree rooted at v and $|T_{\mathcal{S}}(v)|$ is the size of the subtree $T_{\mathcal{S}}(v)$. In the worst case the size of $T_{\mathcal{S}}(v)$ is $\Theta(n)$. In the next section we will see how to improve the running time of a prefix search to $O(m + \text{occ})$, where occ is the number of strings that has P as a prefix.

2.2 Construction of tries

Assume that the alphabet has constant size. To build the trie we can use an incremental algorithm inserting the strings one at a time. Let n_i be the length of the string S_i. To insert a string S_i into the trie $T_{\mathcal{S}}$, we first follow the path from the root matching S_i until we cannot continue any further. Since the set is prefix free the node v we end in will be an internal node in $T_{\mathcal{S}}$. Assume we matched $S_i[1 \dots j]$ up til now. Then we add a path starting in v with edge labels equal to the remaining part of S_i, ie., $S_i[j+1 \dots n_i]$. The length of the new inserted path is $n_i - j$. The time to insert the string S_i is $O(n_i)$: We use $O(j)$ time to follow the path and $O(n_i - j)$ to insert the new path.

The total time to insert all the strings is bounded by

$$\sum_{i=1}^{s} O(n_i) = O(n).$$

2.3 Large alphabets

If the size of the alphabet is non-constant then we can use a dictionary data structure in each node to quickly find the correct edge to follow.

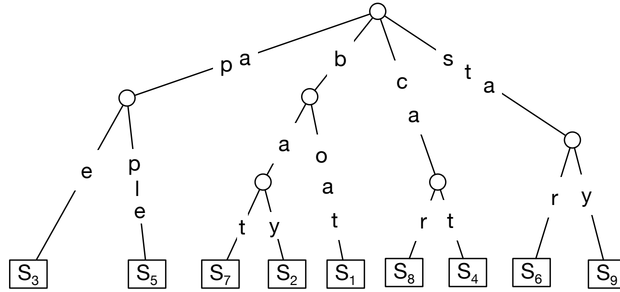
If we use a balanced binary search tree the the time spent in each node is $O(\log \sigma)$, giving us a search time of $O(m \log \sigma)$. The time to construct the trie is then

$$\sum_{i=1}^{s} O(n_i \log \sigma) = O(n \log \sigma).$$

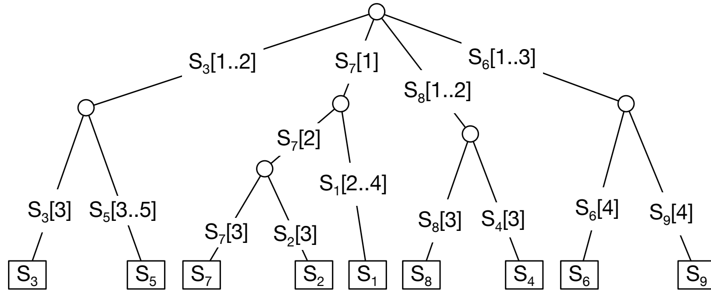
If we use a hashtable then the time spent in each node is *expected* $O(1)$, giving us an *expected* search time of $O(m)$ and $O(n)$ *expected* construction time.

3 Compact Tries

In this section we will show how to speed up the running time of the prefix search and at the same time save space.



(a)



(b)

Figure 2: (a) shows the tree T' obtained from the trie in Figure 1 by contracting paths. (b) shows the compact trie over the set of strings $\mathcal{S} = \{$boat, bay, ape, cat, apple, star, bat, car, stay$\}$. It can be obtained from (a) by replacing labels on edges with indexes into the strings.

The trie $T_{\mathcal{S}}$ over the set of strings $\mathcal{S}$ can have many nodes with only one child and there can be long paths of nodes with a single child. We construct a new tree $T'_{\mathcal{S}}$ by contracting each such path into a single edge as follows. Merge all nodes with only one child with their parent, and concatenate the label of the edge out of v with the label of its parent. This gives us a tree where all internal nodes have at least two children. The number of nodes in this tree is $O(s)$. See example in Figure 2(a).

The tree $T'_{\mathcal{S}}$ still uses $O(n)$ space since the total length of all labels can be strictly larger than the number of nodes in the tree. But we can replace the labels/strings on the edges with indexes into the set of strings. This requires that we save the set of strings, so in total we still use $O(n)$ space, but the space used by the trie is only $O(s)$. This trie is called the *compact trie* over the set $\mathcal{S}$. See Figure 2(b) for an example.

Properties The compact trie T_c has the following properties:

- The number of nodes in T_c is $O(s)$.
- The number of leaves in T_c is s.
- A node in T_c has at most σ children.

- The height of T_c is at most the length of the longest string in $\mathcal{S}$.

The compact trie can be constructed in $O(n)$ time for alphabets of constant size. For large alphabets we can build the trie in $O(n \log \sigma)$ time using balanced binary search trees as described in Section 2.3, or hash tables giving an *expected* construction time of $O(n)$.

Search We can search in the compact trie the same way as in the trie. Searching for a pattern P of length m takes time $O(m)$ if the size of the alphabet is a constant. If not we can use the techniques from Section 2.3 to search in $O(m \log \sigma)$.

Prefix Search We can perform a prefix search the same way as in the trie. We might end the search on the middle of an edge. To find all strings that have P as a prefix, we traverse the subtree below that edge and return the name of the strings saved in the leaves. The time is now improved compared to the prefix search in the trie. To prove this we need the following property of trees:

Let T be a tree where all internal nodes have at least 2 children. Then the number of internal nodes in T is at most the number of leaves in T -1.

We can now analyse the running time of a prefix query. The time for a prefix query is the time to search for P plus the time to traverse the subtree $T_c(e)$ under the edge e that we end on. Each leaf in $T_c(e)$ corresponds to a string in $\mathcal{S}$ that has P as a prefix, thus the number of leaves in the subtree $T_c(e)$ is equal to occ. Since all internal nodes in a compact trie has at least two children then the number of internal nodes in $T_c(e)$ is at most occ. We can do the tree traversal in linear time in the size of the subtree. Thus the running time of the query is $O(m) + O(|T_c(e)|) = O(m) + O(\text{occ}) = O(m + \text{occ})$ for alphabets of constant size. For large alphabets the running time is $O(m \log \sigma + \text{occ})$ if we use balanced binary search trees and expected $O(m + \text{occ})$ if we use hashtables.

Lemma 2 *Given a compact trie over a collection of strings of total length n over an alphabet of size $O(1)$ we can search for a pattern P of length m in $O(m)$ time and do prefix searches in time $O(m + occ)$.*

4 Suffix Trees

In this section we will show how to solve the string indexing problem in $O(m)$ query time and $O(n)$ space.

A *suffix tree* of a string S is the compact trie over all suffixes of S. We add a new character \$ not already in Σ in the end of S to ensure that the set of the suffixes are prefix free. Each leaf in the suffix tree has a number that indicates which suffix of S it corresponds to. That is, the leaf corresponding to the whole string has label 0, the leaf corresponding to the suffix starting at position two in string has label 1, and so on. See Figure 3 for an example. When we draw suffix trees in this note, we will draw them with labels on the edges as in Figure 3(b) for clarity. When implemented they should always be with indexes into the string—otherwise the space will blow up by a linear factor.

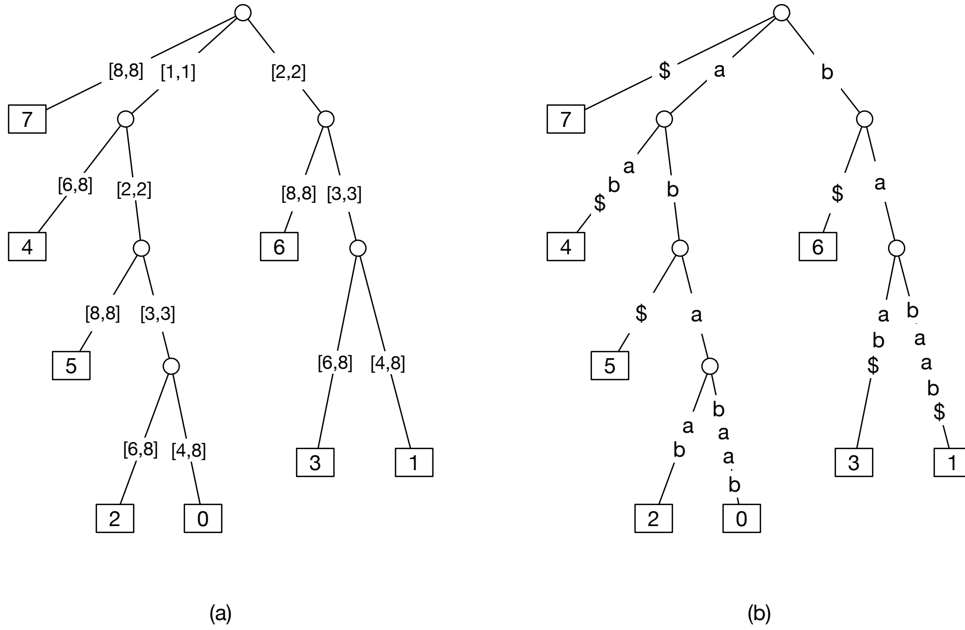


Figure 3: (a) Suffix tree for the string $S = ababaab$. (a) Suffix tree for S with labels on the edges instead of indexes into the string.

4.1 Searching

We can use the suffix tree to find all occurrences of a pattern P in the string S. Just match P from the root following edges according to the next character in P. If the search stops before all of P is read, then P does not occur in S. Otherwise, the search stops on some edge e (or in a node). Now traverse the subtree below e and return the labels of the leaves in this subtree.

4.2 Analysis

Let n be the length of S. The suffix tree uses $O(n)$ space, since there are $n+1$ suffixes of the string $S\$$. In addition to this we need to save the string $S\$$. In total our index uses $O(n)$ space.

Using the algorithm to build a compact trie we can construct the suffix tree in $O(n^2 \log \sigma)$ time or $O(n^2)$ time if the alphabet has constant size.

There exists algorithms there can build a suffix tree in $O(\text{sort}(n, \sigma))$ time, where $O(\text{sort}(n, \sigma))$ is the time it takes to sort n elements from an alphabet of size σ. Thus if σ is constant the suffix tree can be constructed in $O(n)$ time.

Lemma 3 *Let S be a string of length n over an alphabet of length σ. A suffix tree for S can be constructed in $O(sort(n, \sigma))$ time and it supports search for a pattern P of length m in $O(m \log \sigma + occ)$ time.*

If the size of the alphabet is constant then the suffix tree for S can be constructed in $O(n)$ time and it supports search for a pattern P of length m in $O(m + occ)$ time.

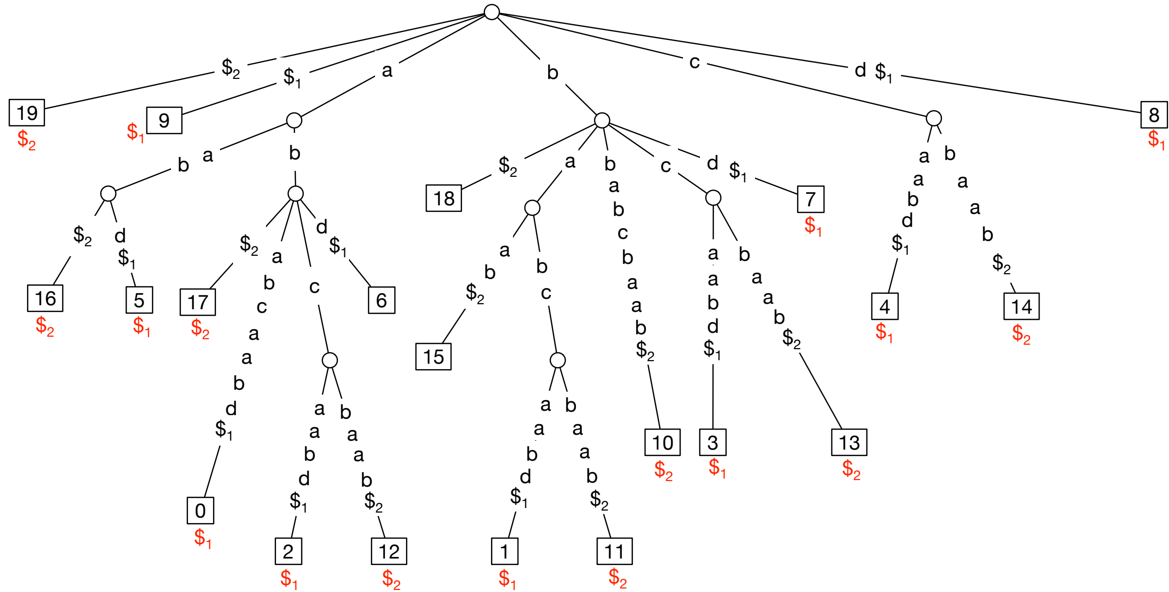


Figure 4: Generalized suffix tree for $S_1 = ababcaabd$ and $S_2 = bbabcbaab$.

4.3 Applications: Longest Common Substring

A string is a common substring of two strings S_1 and S_2 if it is a substring of both S_1 and S_2. A longest common substring of two strings S_1 and S_2 is the longest string that is a substring of both S_1 and S_2.

Example $S_1 = ababcaabd$ and $S_2 = bbabcbaab$. The common substrings of S_1 and S_2 are a, b, c, aa, ab, ba, bc, aab, abc, bab, $babc$. The substring $babc$ is the longest common substring of S_1 and S_2.

Generalized suffix tree We can use a suffix tree to find the longest common substring of two strings S_1 and S_2 in linear time (assuming the size of the alphabet is $O(1)$). First construct the suffix tree of the string $S_1\$_1S_2\$_2$, where $\$_1$ and $\$_2$ are new symbols not already in Σ. Label each leaf with either a $\$_1$ or a $\$_2$ in the following way: If the suffix corresponding to the leaf starts in $S_1\$_1$ then label the leaf with $\$_1$ otherwise label it with $\$_2$. This can be done in linear time once the suffix tree is constructed: Do a depth first traversal of the tree to visit all leaves. In each leaf v check if the label indicating which suffix the leaf corresponds to is less than or equal to the length of S_1, ie., if $\text{label}(v) \leq |S_1|$. If yes, then label the leaf v with $\$_1$ otherwise with $\$_2$. This takes linear time in the size of the suffix tree. The size of the generalized suffix tree is $O(|S_1\$_1S_2\$_2|) = O(|S_1| + |S_2|)$.

This suffix tree T is also called the *generalized suffix tree of S_1 and S_2* (See Figure 4 for an example).

Generalized suffix tree and longest common substring We can use the generalized suffix tree to find the longest common substring of S_1 and S_2. Any node in T that has both a $\$_1$ and a $\$_2$ in its subtree corresponds to a substring that occurs in both S_1

and S_2 (try to convince yourself that this is true). That is, all nodes with both a $\$_1$ and a $\$_2$ in its subtree corresponds to a common substring of S_1 and S_2. The *string-depth* of a node v is the length of the string $\mathrm{str}(v)$ represented by v. The longest common substring of S_1 and S_2 is the node with deepest string-depth among the nodes that has both a $\$_1$ and a $\$_2$ in its subtree. Next we show how to find this in linear time in the size of the generalized suffix tree.

Algorithm To find the longest common substring of two strings S_1 and S_2 do the following.

1. Build the generalized suffix tree T of $S_1\$_1$ and $S_2\$_2$.
2. Mark the nodes of the generalized suffix tree bottom up using a recursive algorithm. A node is marked with a $\$_i$ if one of its children is marked with $\$_i$. Then do a depth first traversal of the tree, adding the string-depth of the node to each node (the string-depth of node v is the string depth of its parent + the number of characters on the edge between them). During the traversal keep track of the node with largest string-depth that is marked with both $\$_1$ and $\$_2$.
3. After the depth first traversal we know the length ℓ of the longest common substring of the two strings ($\ell=$ string-depth(v)) and which node v that corresponds to this substring. It is now straightforward to return the string $\mathrm{str}(v)$ represented by v. Go to any leaf in the subtree of v. This leaf is labeled with a position p in $S_1\$_1S_2\$_2$ that is the starting poisition of an occurrence of $\mathrm{str}(v)$. If this position is in S_1 return the string $S_1[p \ldots p+\ell]$, if it is in S_2 the substring corresponding to $\mathrm{str}(v)$ starts at position $p' = p - (|S_1| - 1)$ in S_2 and we return the string $S_2[p' \ldots p' + \ell]$.

Analysis Assume the alphabet is of constant size. The time to build the generalized suffix tree T is $O(|S_1| + |S_2|)$ Both step 2 and 3 takes time linear in the size of the suffix tree. In step 3 we use at most $O(\mathrm{depth}(T))$ time to go to a leaf and then time proportional to the length of the longest common substring. Both of these are bounded by $O(|S_1| + |S_2|)$. Therefore, the total time to find the longest common substring of S_1 and S_2 is $O(|S_1| + |S_2|)$, when the alphabet has constant size.

If the alphabet is large it takes $O(\mathrm{sort}(|S_1|+|S_2|, \sigma))$ to build the suffix tree. The rest of the steps takes linear time as before. The time to find the longest common substring of two strings for non-constant alphabets is therefore $O(\mathrm{sort}(|S_1| + |S_2|, \sigma))$.