

McCreight's Suffix Tree Construction Algorithm

Milko Izamski B.Sc. Informatics

Instructor: *Barbara König*

1. Introduction

The main goal of McCreight's algorithm is to build a suffix tree in linear time. This is an auxiliary search tree which helps us to find a specific substring S_i within a given main string S .

2. The Algorithm

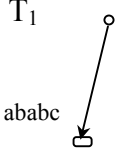
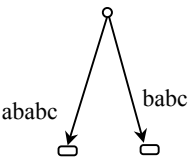
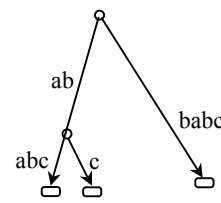
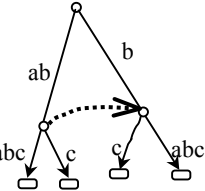
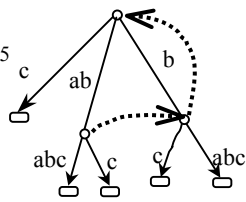
Before explaining the algorithm it is necessary to define some basic rules and constraints. At the beginning we check whether the final character (n) appears elsewhere in S , if it does, then S has to be extended to a string that satisfies the rule such that with no suffix S_i of S is a prefix of a different suffix S_j of S .

The suffix tree T represents all possible suffixes of S . An edge (also known as an arc) of the tree is labeled with some non-empty string from the alphabet that builds S (T1). Each internal node has at least two sons (T2). The sibling edges (sons) begin with a different character (T3).

2.1. Building the suffix tree

Basically T is created in n steps, where $\text{length}(S) = n$. Let suf_i to be suffix of S . Each time it is inserted into T_{i-1} . The leftmost character is at position 1 of the string. Every suffix is built by taking off the first character of its predecessor. Hence, $\text{suf}_1 = S$ and suf_n is the last character.

The next lines will explain how exactly the process work. During this explanation some basis definitions will be introduced. For this task we consider one example string $S = \text{ababc}$. All possible suffixes of S are defined in the table below. In Step 1 suf_1 is inserted in an empty tree T_0 , so T_1 is produced. In Step 2 we take suf_2 and compare it to suf_1 . We define head_i as the longest prefix of suf_i that is also a prefix of suf_j , for some $j < i$. In our situation, head_2 is empty (ϵ). We define tail_i as $\text{suf}_i - \text{head}_i \Rightarrow \text{suf}_i = \text{head}_i + \text{tail}_i$. It means that tail_2 is "abc". To insert suf_i into T_{i-1} we should find the extended locus of head_i , if necessary the tree is split with some non-terminal node (that is the locus of head_i), and then tail_i is insert. During processing in Step 3, $\text{suf}_3 = \text{abc}$, $\text{head}_3 = \text{ab}$ (compare suf_3 with suf_1), hence $\text{tail}_3 = \text{c}$. We split the edge "ababc" into two parts "ab" (that is head_3) and "abc", then insert tail_3 as we said before. The same procedure is repeated until we reach the last possible suffix.

| $S=ababc$ | | | | | | | |
|--|--------|---------|------------|----------|------------|------------|------------|
| | | | | | $head_i$ | | |
| T_{i-1} | Step i | suf_i | $head_i$ | $tail_i$ | α | β | γ |
| T_0 \circ root | Step 1 | ababc | ϵ | ababc | | | |
| T_1  | Step 2 | babc | ϵ | babc | ϵ | ϵ | ϵ |
| T_2  | Step 3 | abc | ab | c | a | b | ϵ |
| T_3  | Step 4 | bc | b | c | b | ϵ | ϵ |
| T_4  | Step 5 | c | ϵ | c | ϵ | ϵ | ϵ |
| T_5  | | | | | | | |

The main problem in the algorithm described above is to find the extended locus in T_{i-1} of $head_i$. All this should be achieved in a constant time. For this task we use a simple Lemma that says: “ If $head_{i-1}$ can be written as $x\delta$ for some character x and some (possibly empty) string δ , then δ is a prefix of $head_i$ “.

2.2. Suffix links

Using this property, auxiliary links (suffix link) can be added to the tree structure. Each locus of $x\delta$, where x is a character and δ is a string, can be linked to the locus of δ . These links enable us to speed up the searching of the locus of head_i , starting at the locus of head_{i-1} , which has been visited in the previous step. But before exploring exactly how that happens we denote every edge from the tree T with a pair of integers, the first element representing the start position of the edge's label in the main string and the second, the length of the edge's label.

Focusing on the following example we will explain how the suffix link is added. Let S be $'b^4ab^3a^2b^4c'$. Now we assume that T_{10} (T_{i-1}) already exists and suf_{11} (suf_i) is to be added - using the property of the suffix links. As we can easily see head_{10} (head_{i-1}) is $'abbb'$. Following the definition above we can represent it as $\chi\alpha\beta$, where $\chi = x = 'a'$ and $\alpha\beta = \delta = 'bbb'$. If the contracted locus of head_{10} (head_{i-1}) in T_9 (T_{i-2}) is the root then α is empty and we start "rescanning" from the root, otherwise it is the locus of $\chi\alpha$. The locus must have existed somewhere in T_9 . In our specific case $\alpha = 'b'$, hence, β must be $'bb'$. The algorithm then follows the already existing suffix link (between $\chi\alpha$ and α) and starts "rescanning".

Rescanning is a process which revisits a sequence of edges which spell out β . We can be sure that the prefix of head_{11} (head_i) will be $\alpha\beta = 'bbb'$ and some other (possibly empty) string γ . In our specific case $\gamma = 'b'$. We already know the locus of α . So following the logic in the last paragraph β has to be within the tree. Thus we start 'rescanning' the tree for β downward from the locus of α . In this process only the first character (see constraint T3) and the length of the child edge ρ are compared (denoting of the edges helps us). If ρ is shorter than β then the process starts recursively from the locus of ρ with $(\beta-\rho)$, otherwise β is a prefix of ρ and the rescan is complete. A new non-terminal node is constructed, which is the locus of $\alpha\beta$, if one does not already exist. The construction of such a node is only possible if γ is empty.

The only thing we don't know yet is, where the γ is located within the tree. We already have information about the length of β because of the head_{i-1} . In this situation we must travel downward into the tree and compare the characters one by one from left to right until we find γ and then create a new non-terminal locus of $\alpha\beta\gamma = \text{head}_i$, if it does not already exist. The process of finding γ in the tree is called "Scanning". At the end tail_i is attached to the locus of head_i and i -step is finished.

Remember that 'rescanning' of β was possible only because in the previous step 'scanning' on β has been performed.

3. Time complexity analysis

Let us define a suffix of S $res_i = \beta\gamma + \text{tail}_i$, where rescanning and scanning has been made to β respectively γ . During the rescanning of β there will always be a non empty string (ρ) that is in res_i but not in res_{i+1} . Hence, $\text{length}(res_{i+1})$ is at most $\text{length}(res_i)$ minus number of the nodes we had encountered by the rescanning(int_i). We already know that $\text{length}(res_n) = 0$ and $\text{length}(res_0) = n$, hence by using recursive substitution we see that $\sum_{i=0}^n int_i$ is at most n . The number of comparisons by the scan operation is $(\text{length}(head_i) - \text{length}(head_{i-1}) + 1)$ and in total algorithm $\sum_{i=0}^n (\text{length}(head_i) - \text{length}(head_{i-1}) + 1) = \text{length}(head_n) - \text{length}(head_0) + n = n$.

The algorithm needs n step for creating the suffix tree. For every step a constant time is needed. That means the algorithm runs in linear time depending on the length of the given string.

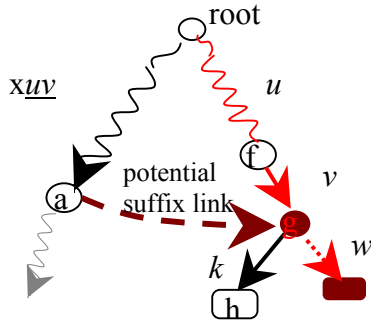
4. Updating the suffix tree

We assume that a substring of the main string S may need to be replaced by S' . Thus, suffix tree T has to be changed too. Let S be defined as $\alpha\beta\gamma$, for some strings α , β and γ , and it is to be changed to $\alpha\delta\gamma$. Adopting a string element position numbering scheme, we consider only those paths that are affected by the replacement of β with δ (or $\alpha\beta\gamma \rightarrow \alpha\delta\gamma$). Define α^* to be the longest suffix of α that occurs in at least two places in S . β -splitters are strings in the form $\varepsilon\gamma$, where ε is a non empty suffix of $\alpha^*\beta$. Hence, β -splitters properly contain the suffix γ . Equivalently, δ -splitters are in form $\omega\gamma$, where ω is a non empty suffix of $\alpha^*\delta$. Main goal of the algorithm is to find the β -splitters and replace them by the δ -splitters. It is achieved in three stages.

1. Discover $\alpha^*\beta\gamma$, the longest β -splitter.
2. Delete all paths $\varepsilon\gamma$ from the tree, $\varepsilon = \text{suf}(\alpha^*\beta)$.
3. Insert all paths $\omega\gamma$ into the tree, $\omega = \text{suf}(\alpha^*\delta)$.

Discovering the longest β -splitter is the first stage. Let us assume our string S is 'abcxbcd', which is to be changed to 'abcacd'. Hence, 'abc'= α , 'xb'= β , 'cd'= γ , 'a'= δ . The longest β -splitter is then 'bcxbcd' because of 'bc'= α^* . Note that 'cxbcd' is also a β -splitter but not the longest.

In the second stage we should care about deleting paths, which are suffixes of the longest β -splitter. Deletions are done in sequence from the longest string to the shortest. Let us assume that all suffixes longer than $\varepsilon\gamma$ have been deleted. We split up $\varepsilon\gamma$ into three (possibly empty) substrings u , v and w . Suppose we have the situation shown on the figure below.



If node g has more than two sibling edges then we just delete w . If g has exactly two offspring edges (it is not possible for g to have only one edge) then w -edge and g -node are removed, k and v are joined together. The only problem is that there could be a suffix link to g which will be deleted after merging of v and k into vk -edge.

The last stage cares about inserting all paths in form $\omega\gamma$, where $\omega = \text{suf}(\alpha^* \delta)$. Now we consider the remainder of the tree as pre-initialized suffix tree, which surely contains all suffixes of γ .

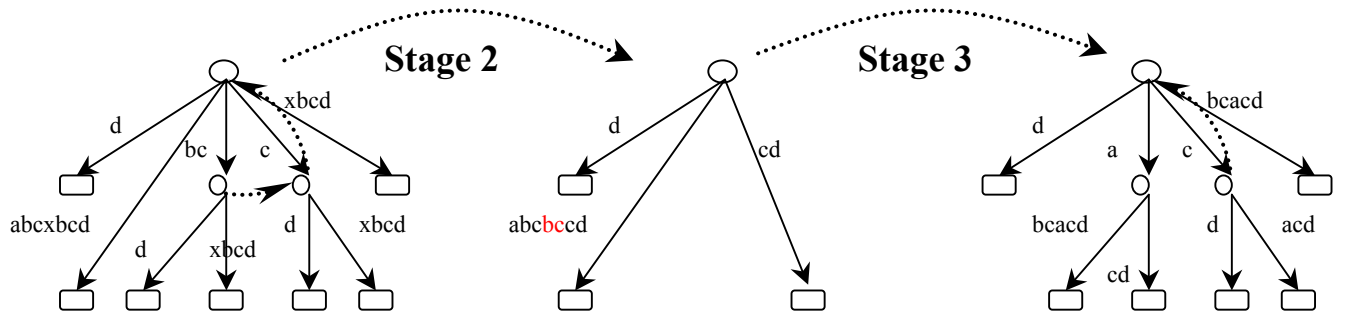


Figure. **Tree mutation in a different stages.**

5. Conclusion

In general McCreight's algorithm for building suffix trees does not differ from the Weiner's or Ukkonen's one. They all create a tree in at most linear time comparative to the length of the input string. A major advantage is McCreight's algorithm uses approximately 25 per cent less data space than similarly coded versions of the other algorithms. Of course in real computers the data movement (load, store, copy-) takes up linear time. Hence, saving data space could also mean saving time.