Suffix Tree Construction, a Tutorial

Robert Burton

1 Introduction

The suffix tree is a powerful, flexible, and efficient tool in a wide variety of string problems. This is in no small part due to the fact that they can be constructed in linear time using linear space. This linear construction can be difficult for many to approach, mostly due to a lack of reader friendly tutorials on the subject. I present here my attempt to fill this niche.

This document assumes that the reader is somewhat familiar already with the Suffix Tree. This tutorial is to enable the reader to efficiently implement the tree, not persuade the reader as to its utility.

Let the reader be advised that presented here is *McCreight*'s algorithm, rather than the sometimes claimed as easier to understand *Ukkonen*'s algorithm. In practice the two algorithms produce the same result with the same efficiency, and the choice between the two is primarily a matter of preference.

For a thorough examination of the applications of suffix trees, I recommend to the reader Dan Gusfield's *Algorithms on Strings, Trees, and Sequences*, which details more than twenty novel applications.

2 Notation and Definitions

2.1 String Notation

A **string** is an array of characters, counted starting from 0. A **substring of** S is written as $S[i : j]$ and consists of the ith character up to but not including the jth character. As such, the length of a substring is $j - i$.

2.2 Suffix

A **suffix of** S is any substring of the form $S[i : |S|] : i < |S|$. The full string, $S = S[0 : |S|]$, is considered a suffix for our purposes.

2.3 Suffix Tree

A **suffix tree** is a tree that represents all suffixes of some string S as root-to-leaf paths. Each node represents some substring of S such that the path from the root to each leaf when concatenated is a complete and distinct suffix. Nodes

branch such that they have at least two children, and that each child node's substring begins with a different letter.

3 Tricks and Concepts

3.1 Compact representation of substrings

If suffix nodes were to have explicit copies of their substrings, they would take $O(n^2)$ space; n for the first suffix, $n-1$ for the next, and so on. Instead, we use the more compact method of only recording the *start* and *end* indexes.

3.2 The *look* pointer and *path* variables

We will maintain a pointer to the current node, called *look* (as in, where we are currently looking). In addition, we will maintain a pair of variables: $path_{start}$ and $path_{end}$. At any given moment, the path from the root of the tree to the *look* node will represent the substring $S[path_{start} : path_{end}]$. The $path_{start}$ variable is governed by the for loop, and indicates the current suffix we are working to insert into the tree.

3.3 The Suffix Link, the *tail* and the old_{end} variable

The suffix link is an augmentation to the base suffix tree. It is a pointer possessed by all internal nodes. Given that an internal node represents some substring $S[i : j] : i < j$, the suffix pointer will be to the node that represents $S[i + 1 : j]$.

In addition, the existence of an internal node directly implies that the head of its suffix link already exists, either explicitly as an internal node or the root, or implicitly, as part of a node that needs to be split. To that end an old_{end} variable is maintained to keep track of both the target $path_{end}$ for the head of the suffix link, and to be aware of what parts of the string path being sought is already known to exist. Newly created internal nodes are also kept in *tail* so their suffix link can be completed.

3.4 Constant Time Child Retrieval

As previously mentioned, nodes in a suffix tree branch on unique letters at the start of their child node substrings. Through the use of a hash we can achieve constant time retrieval of children node addresses. The details of implementation will vary by usage.

4 Algorithm

Suffix Tree construction is best understood as a for-loop over each string index which applies five steps: Ascent, Link or Root, Descent, Create Node, and Create Leaf. Descent is then split into the Skip and Scan step. Each of these

phases have specific, easy to understand roles, especially with the context the prior section offers.

This construction method builds successive partial trees with each iteration of the outer loop. Given a string 'abacb', it would first make T_0 containing 'abacb$', then tree T_1 with 'abacb$' and 'bacb$', then T_2 with 'abacb$', 'bacb$', and 'acb$', and so on until all suffixes have been added to the tree.

The complete algorithm can be found in **Appendix A**.

```
function McCreight(S)
    Setup
    for each suffix do
        Ascent
        Link or Root
        loop Descent
            Skip
            Scan
        end loop
        Create Node
        Create Leaf
    end for
end function
```

4.1 Setup

Construction begins by saving a copy of the string with a unique terminal character appended. This guarantees that each suffix will be a unique substring. In addition, an empty node is created to initialize the tree. It currently holds no suffixes and might be thought of as the -1 intermediary tree, containing no suffixes.

1: **function** Construction(S)
2: $S \leftarrow S +$ "$"
3: $root \leftarrow node(\epsilon)$

4.2 For each suffix

Variables are initialized. For more information on the purpose and behavior of these variables, review the **Tricks and Concepts** section.

The for loop will add a suffix, of the form $S[path_{start} : |S|]$, to the tree each iteration. The *look* pointer will be focused on the parent node to the most recently added leaf.

4: $split \leftarrow 0$
5: $look \leftarrow root$
6: $tail \leftarrow 0$
7: $old_{end} \leftarrow 0$
8: $path_{end} \leftarrow 0$
9: **for** $path_{start} = 0$ **to** $|S| - 1$ **do**

4.3 Ascent

First, the algorithm ensures that *look* either has a suffix link, or is the root. All nodes are either the root or possess a suffix link, with the only possible exception being a freshly inserted node from a prior iteration. As *look* only ever points at nodes, if neither of these conditions are true, it need only move to the parent of the current node.

10: **if** *look* doesn't have a suffix link and isn't *root* **then**
11: $path_{end} \leftarrow path_{end} - len(look)$
12: $look \leftarrow look.parent$
13: **end if**

4.4 Link or Root

This step represents a lateral movement in the tree from the old suffix path to the current one. Lateral movement of a suffix link allows the retention of the $path_{end}$ variable, as a suffix link is defined as being between nodes representing $S[i:j]$ and $S[i+1:j]$. If at the root, $path_{end}$ must be adjusted to reflect a 0 length path.

14: **if** *look* is the root **then**
15: $path_{end} \leftarrow path_{start}$
16: **else**
17: $look \leftarrow look.suffix_link$
18: **end if**

4.5 Descent

Each iteration of this inner loop represents the examination of a node as the algorithm searches for the node which the new suffix will branch off from. At the same time, if there is an incomplete suffix link, nodes are checked to see if they are the head of such a link.

If a node is found to not have a child matching the indexed character to pursue, the loop is exited. All of the overlap between the tree and $S[path_{start} : |S|]$ has been found. Otherwise, a *child* is found to pursue.

19: **loop**
20: **if** $path_{end} = old_{end}$ **and** *tail* doesn't have suffix link **then**
21: $tail.suffix_link \leftarrow look$
22: **end if**
23: **if** $S[path_{end}] \notin look$ **then**
24: **break**
25: **else**
26: $child \leftarrow look[path_{end}]$
27: **end if**

4.5.1 Skip

First, the algorithm attempts to skip to the *child* node. The existence of a path representing $S[path_{start} - 1 : old_{end}]$ directly implies that $S[path_{start} : old_{end}]$ exists within the tree, though not necessarily ending upon a node. Regardless, it allows the algorithm to skip to the end of any child. Having advanced to a new node, it returns to the start of the descent loop.

If it is found that $S[path_{start} : old_{end}]$ ends partway through a node's substring, or if $path_{end}$ is already past old_{end}, the algorithm simply advances to that point and prepares for the Scan step.

28: $split \leftarrow 0$
29: **if** $path_{end} + len(child) \leq old_{end}$ **then**
30: $path_{end} \leftarrow path_{end} + len(child)$
31: $look \leftarrow child$
32: **continue**
33: **else if** $path_{end} < old_{end}$ **then**
34: $split \leftarrow child_{start} + old_{end} - path_{end}$
35: $path_{end} \leftarrow old_{end}$
36: **else**
37: $split \leftarrow child_{start} + 1$
38: $path_{end} \leftarrow path_{end} + 1$
39: **end if**

4.5.2 Scan

As of this step, $path_{end}$ will exceed old_{end}. The algorithm Scans *child*'s substring for a point of divergence, and once found will break from the loop. It is also possible to exhaust the substring, in which case *look* moves to the *child* and continues to a new iteration.

40: **while** $split < child_{end}$ **and** $S[split] = S[path_{end}]$ **do**
41: $split \leftarrow split + 1$
42: $path_{end} \leftarrow path_{end} + 1$
43: **end while**
44: **if** $split = child_{end}$ **then**
45: $look \leftarrow child$
46: $split \leftarrow 0$
47: **continue**
48: **else**
49: **break**
50: **end if**

4.5.3 Breaks

Upon breaking from the loop, *look* is either focused upon the node from which the new leaf will be added, or upon *look* and a *child* with an index to split *child*'s substring to create the node upon which the new leaf will be added.

51: **end loop**

4.6 Create Node

If necessary, the algorithm splits the *child* node upon the index *split*, and moves *look* to the resultant inner node. If *tail* has not yet found its head to complete the suffix link, this must necessarily be the head. This node is also noted as being the tail of a new suffix link.

The result is that *look* is now guaranteed to be focused upon the node from which the new leaf will be added.

52: **if** $split \neq 0$ **then**
53: Create *node* between *look* and *child*
54: $node.S \leftarrow S[child_{start} : path_{end}]$
55: $child.S \leftarrow S[path_{end} : child_{end}]$
56: $look \leftarrow node$
57: if old suffix link not complete, do so now with this node
58: $tail \leftarrow look$
59: **end if**

4.7 Create Leaf

From *look*, a leaf is created with the remaining substring.

60: Create leaf from *look* to represent $S[path_{end} : |S|]$

4.8 Conclusion

The end of the substring represented by *look* is recorded as old_{end}. The loop concludes.

At the end of each iteration of the loop, an intermediary suffix tree will have been created containing one leaf, and therefore suffix, per iteration thus completed. The *look* pointer will be upon the parent node to the new leaf, and if it is an internal node, it is the only one which may not have a completed suffix link.

61: $old_{end} \leftarrow path_{end}$
62: **end for**
63: **end function**

5 Space Complexity

With a few observations, it becomes simple to verify that this representation and construction of a suffix tree has linearly bounded space complexity.

The tree itself has at most $2|S|$ nodes. It must have $|S|$ leafs, one for each suffix. Similarly, there can be no more than $|S|$ internal nodes created; in fact, the 0th iteration can never create an internal node, but does begin with the root node.

The nodes themselves have constant bounds. By representing substrings as starting and ending indexes, the strings are compactly stored. The only complication is that nodes can have as many children as there are distinct characters in S. This is, however, still constant size.

The auxiliary memory used by the construction algorithm is limited to a set of well defined variables and pointers, and is therefore of constant size.

Therefore, a linear bound on some quantity of constant sized nodes and constant auxiliary space usage makes this method firmly of $O(n)$ space complexity.

6 Time Complexity

This construction method has a linear time complexity, which is easily demonstrated.

Most steps of the algorithm are trivially of constant time. The only exception is the Descent step, which may be executed repeatedly for a single suffix.

6.1 Maximum Number of Descents

It can be shown that the maximum number of descents (increase by one the depth of *look*) cannot exceed $3|S|$. This is based on the fact that construction is limited to $2|S|$ ascents.

First, it is observed that starting from the root at most $|S|$ descents can be performed, given that the greatest possible depth of a leaf is $|S|$. (That is, the case of $S[0 : |S|]$ and a series of nodes with substrings of length 1.) Any additional descents are dependent upon first ascending.

For each suffix, there exists the possibility of moving *look* to the parent node during the Ascent step. This accounts for an additional $|S|$ possible descents for a running total of $2|S|$.

For each suffix there also exists the possibility of moving *look* along a suffix link during the Link or Root step. This move will either maintain or reduce by one the depth of *look*. This is because all ancestor internal nodes to the original node will have suffix links to appropriate ancestors of the destination node which will be either an internal node or the root. That is, given the root-to-node path $S[i : j]$ and a suffix link to some root-to-node path $S[i + 1 : j]$, for any $S[i : k] : i < k < j$ there will also exist a $S[i + 1 : k]$. In the case that $i + 1 \neq k$, the destination nodes will be internal. In the case that $i + 1 = k$, which can only occur once, the destination node is the root. Therefore, the destination node will either have the same depth, or a depth one less.

It follows that each link that is traveled could possibly be an ascent. Over the course of the main loop this could create up to $|S|$ ascents, and therefore also making possible $|S|$ more descents for a final theoretical maximum of descents of $3|S|$.

6.2 Strictly Increasing Scan

It can be shown that there is a strict linear bound on the number of character comparisons that are performed. This is evident in the fact that in the global context, within the Scan loop, $path_{end}$ is strictly increasing and has an implicit maximum of $|S|$. This is easily included once it is observed that:

1. A post-condition of the Skip sub-step is that $path_{end} > old_{end}$.
2. $path_{end}$ cannot decrease in the Scan, Create Node, or Create Leaf steps.
3. old_{end} is assigned the value of $path_{end}$ at the end of the main loop.

It is assured that even in the worst case only $|S|$ character comparisons will be performed.

6.3 Combining

To tie it all together: most steps of the construction will be performed at most once for each suffix, and those steps which do not can be shown to have clear linear bounds as to how often they occur. Therefore, construction has a $O(n)$ time complexity.

7 Conclusion

Suffix trees, properly implemented, are efficiently made and stored. The final structure is surprisingly data rich, and there are any number of clever applications that can take advantage of the structure.

I consider this method (*McCreight*) to be more easily understood than either the *Ukkonen* or *Weiner* method because this method does not include implicit construction the other common linear construction methods make heavy use of.

For those interested in learning more, the *Ukkonen* method is most similar to the construction presented here, except that the old_{end} variable would be a pointer to the current end of the string represented by any intermediary tree. In this way, it would construct a complete tree for 'a', then 'ab', then 'aba' and so on until 'abacb$'. In all other ways it behaves exactly as the construction method here, including space and time efficiency.

The *Weiner* method, by contrast, builds trees from the end to the beginning: '$', then 'b$', then 'cb$' and so on until 'abacb$'. This method uses slightly more auxiliary space than the other two methods discussed.

References

Gusfield, D. (1997). *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge university press.

A Algorithm

```
1: function CONSTRUCTION(S)
2:     S ← S + "$"                                                    ▷ Setup
3:     root ← node(ϵ)
4:     split ← 0                                                      ▷ For each suffix...
5:     look ← root
6:     tail ← 0
7:     old_end ← 0
8:     path_end ← 0
9:     for path_start = 0 to |S| − 1 do
10:        if look doesn't have a suffix link and isn't root then     ▷ Ascent
11:            path_end ← path_end − len(look)
12:            look ← look.parent
13:        end if
14:        if look is the root then                                   ▷ Link or Root
15:            path_end ← path_start
16:        else
17:            look ← look.suffix_link
18:        end if
19:        loop                                                       ▷ Descent
20:            if path_end = old_end and tail doesn't have suffix link then
21:                tail.suffix_link ← look
22:            end if
23:            if S[path_end] ∉ look then
24:                break
25:            else
26:                child ← look[path_end]
27:            end if
28:            if path_end + len(child) ≤ old_end then                ▷ Skip
29:                path_end ← path_end + len(child)
30:                look ← child
31:                continue
32:            else if path_end < old_end then
33:                split ← child_start + old_end − path_end
34:                path_end ← old_end
35:            else
36:                split ← child_start + 1
37:                path_end ← path_end + 1
38:            end if
39:            while split < child_end and S[split] = S[path_end] do  ▷ Scan
40:                split ← split + 1
41:                path_end ← path_end + 1
42:            end while
43:            if split = child_end then
44:                look ← child
```

```
45:                 split ← 0
46:                 continue
47:             else
48:                 break
49:             end if
50:         end loop
51:         if split ≠ 0 then                                    ▷ Create Node
52:             Create node between look and child
53:             node.S ← S[child_start : path_end]
54:             child.S ← S[path_end : child_end]
55:             look ← node
56:             if old suffix link not complete, do so now with this node
57:             tail ← look
58:         end if
59:         Create leaf from look to represent S[path_end : |S|]      ▷ Create Leaf
60:         old_end ← path_end
61:     end for
62: end function
```