**Technical Paper on Ukkonen’s Algorithm**

**Group 6**

**Abstract**

Our group plans to provide a study on “Construction of suffix tree using Ukkonen's algorithm”.  Ukkonen’s algorithm is a liner-time, online algorithm for constructing suffix trees.[1] Suffix trees are inherently asymmetric: prefix extensions only cause a few updates, while suffix extensions affect all suffixes causing a wave of updates. In his elegant linear-time on-line suffix tree algorithm Ukkonen relaxed the prevailing suffix tree representation and introduced two changes to avoid repeated structural updates and circumvent the inherent complexity of suffix extensions: (1) open ended edges that enjoy gratuitous leaf updates, and (2) the omission of implicit nodes.[2] Ukkonen’s algorithm constricts a sequence of implicitsuffix trees, the last of which is converted to a true suffix tree of the string S. [3] Output would be suffix tree edges like StartNode, EndNode, suffixIndex.

**Introduction**

Ukkonen's Algorithm was name after Esko Ukkonen who had developed ukkonen's algorithm for suffix tree construction in 1995. Ukkonen's algorithm is also referred as linear algorithm. Ukkonen has developed an algorithm with the motivation to develop an online algorithm, example one we can update with new sequences.

This on-line algorithm was developed to reduce the processing time and size of the atomic suffix tree of a suffix tree from *O*(*n*2) to *O*(*n*), which is the representation of linear form. As this algorithm is completely depending suffix tree, let's have a background about suffix tree. To understand suffix trees and their applications we need to have Math background.

Suffix tree contains a set of strings where all the suffixes of a given word are divided as their keys and positions in the word as their values. Using suffix tree, implementation of string operations is fast. To develop any algorithm on strings, suffix tree is very important. The regular linear algorithm takes too much time to locate substrings in a string where suffix tree helps to speed up the process but takes more space than just storing the string itself.

Consider a string A with length n, then the tree will have n leaves. Each node will have at least two children but not the root node. It is very important to maintain the relation between root and node or in other words relationship between parent node and child node. Each edge of the string A are non-empty substring of A. Ever edge must start with different character. By combining all sub-string labels from the root to the leaf, we should be able to get the string A. While branching the leaves, we use a terminating symbol $ of the sub-string. This suffix tree can be built in O(n) time. When we have a lengthy string, time to build a suffix tree is O(nlogn)times.

**Applications:**

Suffix trees are used to solve problems related to strings. Examples: finding the repetition of longest substring, finding the lengthiest common substring, finding the lengthy palindrome in a string. There are much more than the above-mentioned applications.

**Implementation:**

Ukkonen has set certain rules and extensions to build a suffix tree using online linear type algorithm.

Rule 1: Walk through the path add new character at the end.

Rule 2: If there is no path with certain label, create a leaf node.

Rule 3: if the path already exists, then we don't have to do anything.

By following above mentioned rules we can build a suffix tree, for every phase we are doing *O*(*n*2) time of work, where there are n Phases. total algorithm time is *O*(*n*3). To convert this into O(n), we should follow following method.

1. Skip count and edge label compression: Instead of storing actual characters in the edge, we will be storing index range of the characters in the input array. By using edge label compression, it reduces the space complexity and run-time complexity.
2. Rule 3 extension is a show stopper: By using rule 3 extension, we stop the current phase and starts a new one, this saves a lot of time.
3. Global end for leaves: Replace the last value in the index range with end. When a new character is added to the string, we don’t have to go through the complete process. Instead, we can increment end node by 1, rule 1 extension will be taken care of O(n) times. This is applicable only to the leaf nodes but not to the internal nodes.

Once the true suffix tree is built, we add index to every leaf by subtracting length of the leaf from length of the string.

**Example 1**

Let’s consider ‘xyzx’ is the tree. Before we start building a suffix tree, assign terminating character, $ at the end of the tree elements.

Let’s start constructing suffix tree using O(n3) time algorithm and then convert into O(n) time complexity using Ukkonen’s algorithm. So, let’s take i and j are phase and extension respectively. when i and j are “0” we have to check whether there is a path exists from root starting with “X”, so there is no path starting with “ X” . So as per rule 2 if there is no path with certain label, create a leaf node.

XYZX$

0 1 2 3 4

X

j

i

Then increment phase “I” by 1, and extension “j” is again “0”. Then we must check whether there is path between j to i that is “XY”. if we don’t have as per rule 1 walk through the path add new character at the end.

XYZX$

0 1 2 3 4

YX

j i

Then increment extension j by “1” , Phase i is same . So as per rule 2, we must check whether there is a path with label Y. So, we don’t have path so we create one.

XYZX$

0 1 2 3 4

Y

YX

i

j

Then we increase Phase “i” by 1, i.e., now our phase “i” becomes 2. So, and j again we start from “0”. We repeat the above steps again by rule 1 and rule 2. So we don’t have a path from “j” to “i” that is “XYZ” we add “z” at the end . and increment “j” by 1, we check for j to i that is “YZ” we add “Z” at the end. Again increase “j” by 1 now both “i” and “j” are at point 2 so as per rule 2 there is no path starting with “Z” so we create a path with “Z”. By following above steps will get the required suffix tree.

XYZX$ by Rule 1

0 1 2 3 4

Y

ZYX

j i

XYZX$ by Rule 1

0 1 2 3 4

YZ

ZYX

j i

XYZX$ by Rule 2

0 1 2 3 4

YZ

ZYX

i

Z

j

By repeating above steps by incrementing phase “i” by 1. Now phase “i” is 3 .

XYZX$ by Rule 2

0 1 2 3 4

XZYX

YZ

j i

Z

XYZX$ by Rule 2

0 1 2 3 4

YZX

XZYX

j i

Z

XYZX$ by Rule 2

0 1 2 3 4

YZX

XZYX

j i

ZX

XYZX$ by Rule 3 , So here we have path starting with “X” , so no need to do anything.

0 1 2 3 4

YZX

XZYX

i

ZX

j

By repeating above steps by incrementing phase “i” by 1 now phase “i” is 4.

XYZX$ by Rule 2

0 1 2 3 4

YZX

$XZYX

j i

ZX

XYZX$ by Rule 2

0 1 2 3 4

YZX$

$XZYX

j i

ZX

XYZX$ by Rule 2

0 1 2 3 4

YZX$

$XZYX

j i

ZX$

XYZX$, So we have to create internal node after “X” , so we get “X$”.

0 1 2 3 4

YZX$

$XZY

X

j i

$

ZX$

XYZX$, Rule 2

$

0 1 2 3 4

Figure 1.1

$XZY

X

YZX$

i

$

ZX$

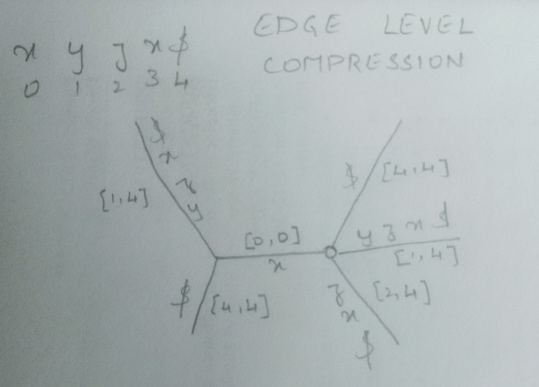
j

To convert this O(n3) time into linear O(n) form use the following 3 steps

1. By applying edge label compression

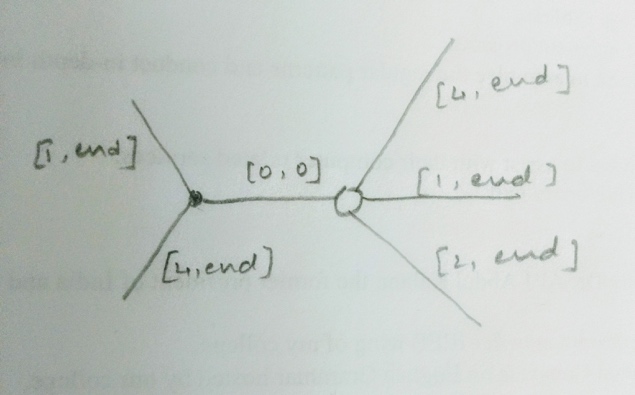
In edge label compression, we replace actual characters in the edge with index range of those characters in the input array (xyzx$). By using edge label compression space complexity will be reduced and the run time complexity would never get better than O(m2) without this trick.

Figure 1.2



1. Rule 3 extension is a show stopper, means it stops the current phase and move to the next one.
2. Global end for leaves – repeat the lead nodes range with ‘end’

Figure 1.3



**Example 2**

String: xyzx$axyb

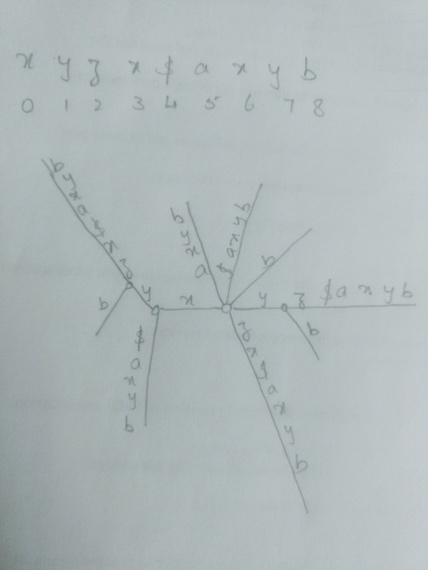


Figure 2

Add extension to the existing tree. Add axyb in the example 1. By applying ukkonen’s algorithms three rules let’s add the new elements into the tree.

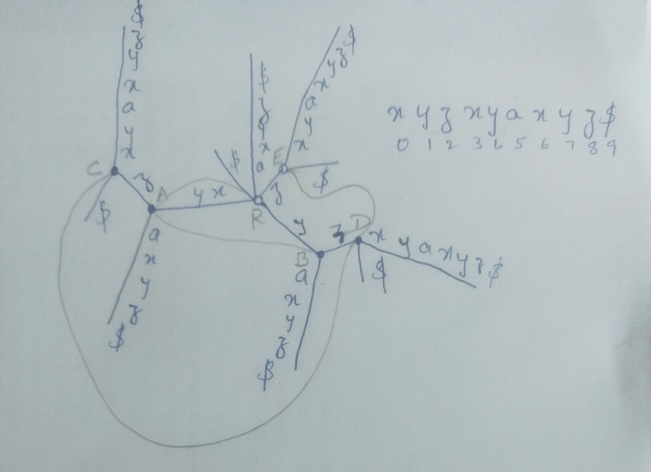
**Suffix links**

For any internal node ‘A’ with path ‘t𝜶’, there is another internal node ‘sv’ with 𝜶 which is susfix link of ‘v’. If 𝜶 is null, then the suffix link is root. Here, t holds a single element where 𝜶 varies from 0 to infinity. It’s proved that very internal node will have a suffix link as another internal link or node. Suffix link is important to traverse between the nodes in a tree which helps to improve the run-time.

Creating a suffix link – When an internal node is created during an extension of a phase and another internal node is created during the next extension of the same phase then the new internal node will be the suffix link of the older internal node.

**Example 3** String xyzxyaxyz$

Figure 3



As explained in the above examples let’s build a suffix tree using Ukkonen’s rules and draw suffix links between the nodes. Here R is the root node, A, B, C, D, E are the other nodes. B is the suffix of A, the path of A is xy – where x is t and y is 𝜶, so B is label ‘y’ is the suffix of A as per definition. D is suffix of C, path from C is xyz where x is t and 𝜶 is ‘yz’.

**Active point** is an extension from which next phase will start. Active point includes active node, active edge and active length. Active node is the current node in the tree. Active edge is the index of the actual character and active length is length of the current character. Initial values are

Active node: root

Active edge: -1

Active length 0

Now explaining active nodes for the example 3. After $ let’s add ‘x’, then the string is xyzxyaxyz$x

Active nodes using rule3 extension:

When the leaf is existing, we don’t have to do anything. In this case, increase the active length by 1 and active edge becomes 0. Implies from the root, path moves in the direction of x. Now add ‘y’ after x then the string becomes xyzxyaxyz$xy. As y path is already existing, we don’t have to do anything which increments the active length and it becomes 2. Now add ‘z’ to the string, xyzxyaxyz$xyz. Here position jumps a node, A to reach z so the active node becomes A.

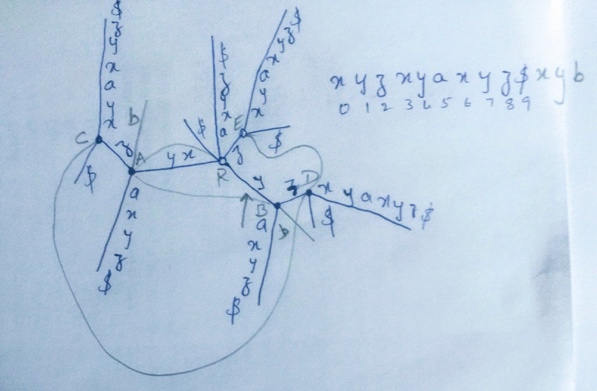
Active node: A

Active edge: 2 (index of z)

Active length: 1

Active point & rule 2 extension for the example 3:

Current string is xyzxyaxyz$. Add ‘x’ element, string is xyzxyaxyz$x. Here active node is root, active edge is 0 and active length is 1. Add second element ‘y’, string becomes xyzxyaxyz$xy. Active node: root, active edge: 0 anf active length: 2 (according to rule 3 extension). Add element ‘b’ after ‘y’, new string is xyzxyaxyz$xyb. Xyb path is not existing so we create one by using rule 2 extension. In this case, we decrement length by 1 and increment edge by 1 (index of y).



Active node: R

Active edge: 1

Active length:1

Now pointer moves from b to y as the active edge is 1. Xyb path has been created. Now we need to create yb path by adding new leaf ‘b’ at the ‘B’. Now create path ‘b’ as it not existing from the root.

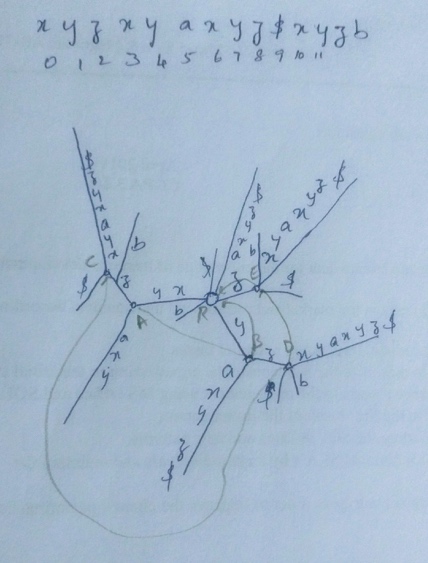
Active points & suffix links:

String: xyzxyaxyz$xyzb

Active node: B

Active edge: 2

Active length: 1



From the root, xyzb path is not created so we need to create a ‘b’ path after ‘z’. Now node becomes A as we jumped a node from root. Follow the suffix link, A to B and check at the node B for the existing paths and create a new path ‘b’ at ‘z’. Now active node becomes D, active edge becomes 2 and the position points ‘z’. Here create a new leaf ‘b’. active node becomes root, active length becomes 0. Create a new leaf at the node. This explains how add new elements in the suffix tree following suffix links and using active points.

**Usage:**

Ukkonen's algorithm is an 'online algorithm', this is true as far as this condition is satisfied - no suffix is a prefix of another, the intermediate tree are suffix trees. Also, we can build generalized suffix trees one string at a time. The main drawback of suffix tree is it takes too much memory to perform implementation of string into substrings and the Ukkonen’s algorithm reduces memory usage.

**Conclusion:**

Ukkonen’s algorithm helps to reduce space and run time complexity. Reduces O(n2), O(n3) into linear form, O(n).