0.1 Generalized suffix trees

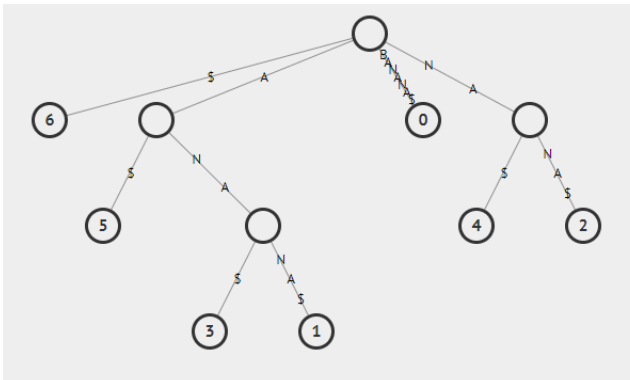
- *Algorithm:* Generalized suffix trees(algo. 0.1)
- *Input:* a set of strings
- *Complexity:* $\mathcal{O}(N)$[1], *where N is the total length of strings*
- *Data structure compatibility:* suffix trie, suffix array
- *Common applications:* substring check, searching all patterns, longest repeated substring, build linear time suffix array, longest common substring, longest palindromic substring

Problem. Generalized suffix trees

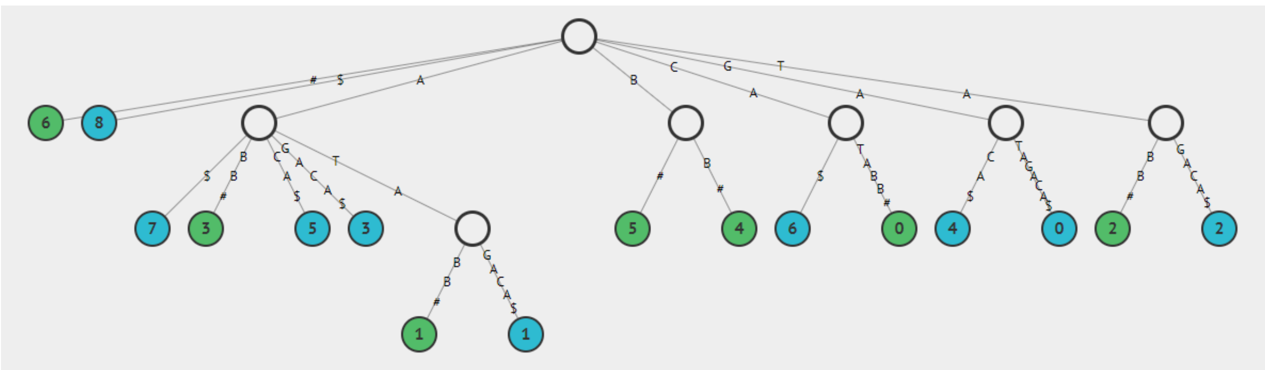
Generalized suffix tree is a suffix tree made of a set of strings. A suffix tree is a compressed tree containing all the suffixes of the given string as edges and positions in the string as values of leaves.

Description

Suffix Tree (Fig. 1) provides a particularly fast implementation for many important string operations, such as substring check, longest repeated substring. Suffix Tree (Fig. 1a) consists of suffixes of the given string. The **i**-th suffix is the substring that goes from the **i**-th character of the string up to the **last** character of the string, e.g., The first suffix of "BANANA\$[2]" is "BANANA\$" and the third suffix is "NANA\$". Generalized suffix tree (Fig. 1b) is a suffix tree made of a set of strings. Generally, we merge the set of strings with "#", e.g., "CATABB#GATAGACA\$".



(a) Suffix tree of "BANANA\$"



(b) Suffix tree of "CATABB#GATAGACA\$"

Figure 1: Suffix tree figures generated by *Visualgo*[3]

[1] Space complexity

[2] We use \$ to mark the end of the string

Complexity

The space complexity of generalized suffix tree is $\mathcal{O}(N)$, where N is the sum of the length of the set of strings. Considering the maximum number of nodes in a generalized suffix tree, there are at most N leaf nodes. And there are at most $N-1$ non-leaf nodes including the root, on account for suffix tree is commpressed, all non-leaf nodes must be branching. Hence, the maximum number of nodes in generalized suffix tree is $2N-1=\mathcal{O}(N)$, which is much better than $\mathcal{O}(N^2)$ of suffix trie. Here is a simple example (Fig. 2)

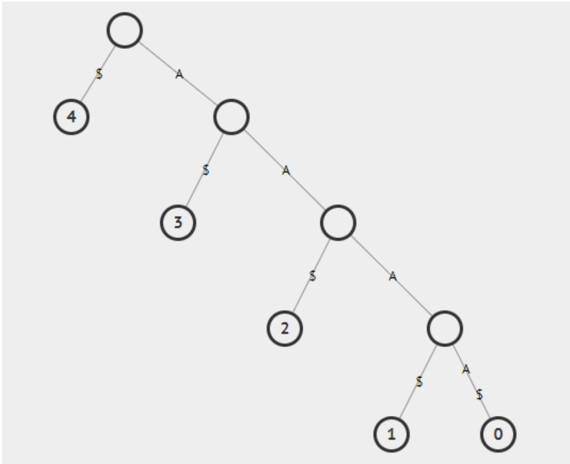
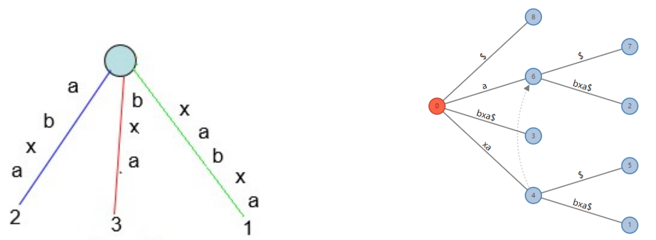


Figure 2: Suffix tree figure of "AAAA$" generated by *Visualgo*[3]

Suffix Tree Construction[4]

In 1995, Esko Ukkonen published *On–line construction of suffix trees*[5], which introduced the method to construct suffix tree in time linear in the length of the string. Different from trie, as we mentioned before, suffix tree is compressed, i.e., each edge is a substring **[i, j]**[3] rather than a single character. It seems that we could build a suffix trie and compress it to obtain a suffix tree. However, the time complexity of suffix trie construction is $\mathcal{O}(bN)$, where b is the length of the alphabet. Therefore, we introduce **Ukkonen's suffix tree construction**(algo. 1).

Before we start, we get two new terms, **Explicit suffix tree** and **Implicit suffix tree** (Fig. 3). Implicit suffix tree does not branch when a suffix is a substring of a suffix existing in the tree. In the example of implicit suffix tree of "xabxa" (Fig. 3a), "xa" and "a" are substring of existing suffixes, but they are not shown "explicitly". To generate an implicit tree, we could simply insert each character of the string in turns. If the inserting character does not existing in the tree, get it a new edge. If not, skip it, then add the new character to each edge. Apparently, we construct an implicit suffix tree in time linear. Hence, the key to Ukkonen's suffix tree construction is to expand the tree into explicit suffix tree (Fig. 3b) while generating an implicit suffix tree.



(a) Implicit suffix tree of "xabxa"

(b) Explicit suffix tree of "xabxa$"[1]

Figure 3: Implicit and explicit suffix trees

[3]For convenience, we use [**i**, **j**] to represent the substring which goes from **i**-th character to the **j**-th character of the string.

Now, let us go through Ukkonen's suffix tree construction of "abcabxabcd$". As for the first character, "a", which is not in the tree (empty), thus create a new edge for it (Fig. 4). The value of the node is the position of "a" in the string, 1[4].

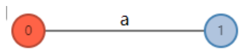


Figure 4: Inserting "a" to the suffix tree[1]

Then we deal with the second character "b". Create a new edge for "b", the value of the node is the position, 2. Add "b" to each edge.

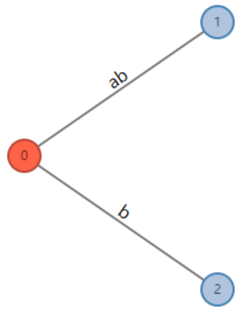


Figure 5: Inserting "b" to the suffix tree[1]

Keep this operation, we insert the 4-th character "a" and get an implicit suffix tree (Fig. 6), which is already existing in the tree. We have to consider how to branch the tree.

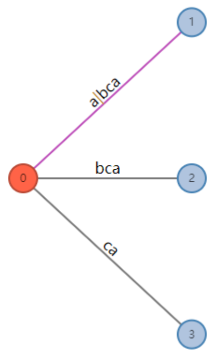


Figure 6: Inserting "a" to the suffix tree[1]

Here we introduce two definitions, **active point** and **remainder**.

- Active point: (active_node, active_edge, active_length)
- Remainder: an integer representing how many new suffix we are going to insert

We will explain the definitions in details during the construction. Now, we are dealing with the 4-th character "a", which is already in the tree, so we did not create a new edge for it. Thus, remainder is 1 for we still have 1 suffix "a" to be insert. Then active point is (root, a, 1), where *active_node* is initialized to *root* which is not change, *active_edge* is "a" representing that the edge beginning with "a" will be branch and *active length* is 1 representing length of "a" also the distance from the position where to spilt the edge to *active_node* (Fig. 7).

[4]In following figures, the number is not the value of the leaf, just the mark of nodes.

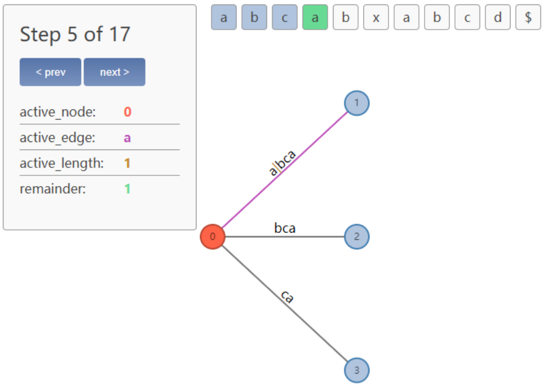


Figure 7: Active point and remainder[1]

Keep going to next character “b”, which is also existing in the tree. Hence, active point is updated to (root, a, 2). *active_edge* is still “a” edge, but *active_length* is 2, the length of “ab”. Th remainder is updated to 2, we have “ab” and “a” to insert (Fig. 8).

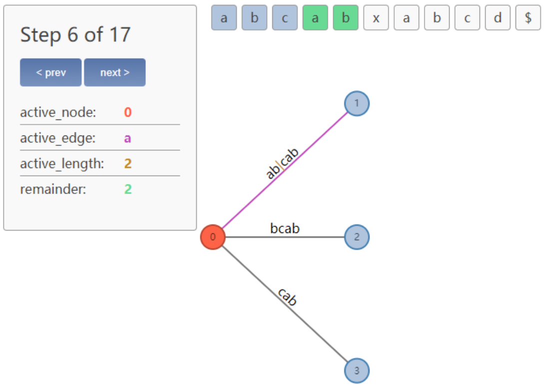


Figure 8: Inserting “b” to the suffix tree[1]

When inserting next character “x”, we find that it is not in the tree, so we split at the mark on the edge (Fig. 8). There is a new leaf in the suffix tree and its value will be assigned as 4. Active point and remainder are updated to (root, b, 1) and 2. But how? Actually, when it is “x”'s turn, remainder is updated to 3, “abx”,“bx” and “x”. After spliting, suffix “abx” is inserted and “bx” and “x” are left. Hence, remainder is 2, we will split at edge “b” next and the split position is after “b” (Fig. 9).

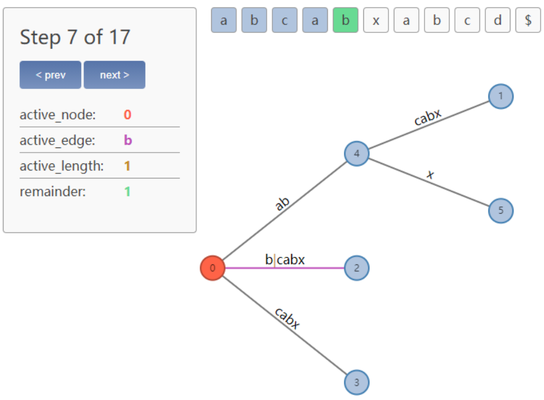


Figure 9: Split at edge of “a”[1]

We keep doing the spliting process on the edge of “b”. After spliting, we establish a link from Node4 to the new node, called **suffix link** (Fig. 10), which is the key to make Ukkonen's construction in time linear. If the new non-leaf node is not the first one to be generated, establish suffix link to the previous non-leaf node. Then update active point and remainder to (root, none, 0) and 1. (root, 0) means to split at root (distance is zero), i.e., create a new edge for the last one suffix "x". For now, the insertion of "x" accomplished.

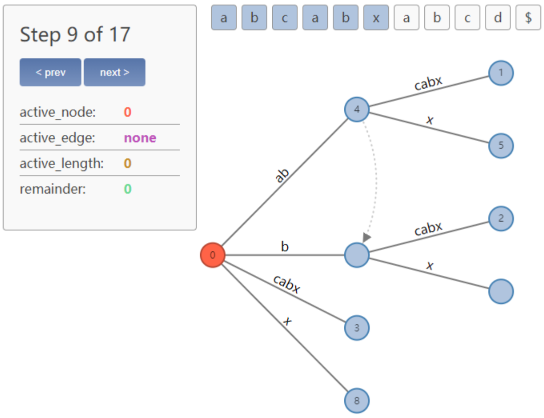


Figure 10: Suffix link[1]

Then we repeat the process for the next "abc", which covers the edge of "ab". Thus we determine *active_ length* based on Node4. Update active point and remainder to (Node4, c, 1) and 3. Then we insert "d", the split process begins. After inserting the suffix "abcd", we could change the *active_ node* by suffix link directly (Fig. 11), which make the algorithm in time linear.

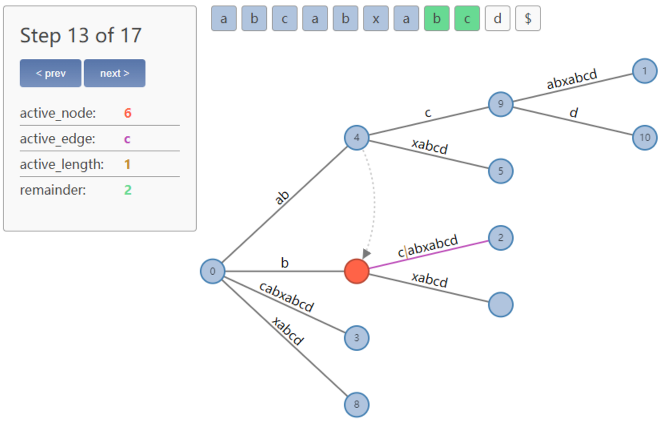


Figure 11: Suffix link[1]

Repeat the process, Ukkonen's suffix tree construction completes (Fig. 12).

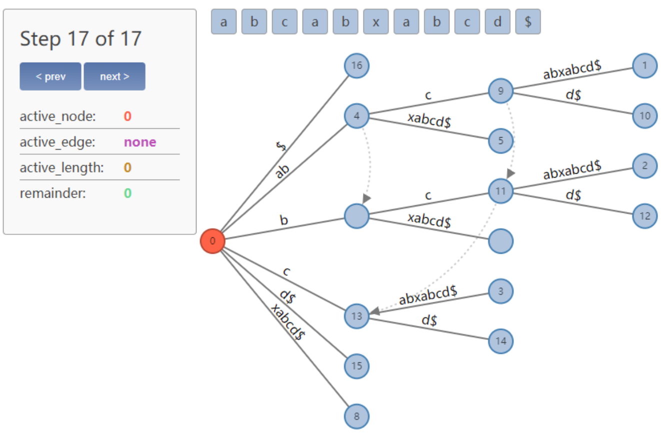


Figure 12: Suffix link[1]

Algorithm 1: Ukkonen's suffix tree construction

Input : A string S

Output: A suffix tree

```
S += "$"
activePosition ← (active_node, active_edge, active_length)
value ← 0                                    /* value to be assigned to leaf nodes */
for i from 1 to len(S) do
    active_node ← root
    active_edge ← none
    active_length ← 0
    remainder ← 0
    Add S[i] to each edge of leaf nodes
    if S[i] is in any edge then
        if active_edge == none then
            active_edge ← S[i]
        end if
        active_length += 1
        if active_length equals to the length of the edge including S[i] then
            active_node ← the node that the edge leads to
            active_edge ← none
            active_length ← 0
        end if
        remainder += 1
        continue
    end if
    ifFirst ← 1
    prevNode ← NULL
    while remainder > 0 do
        Generate a new node split on active_edge at active_length position
        Generate a node leaf linked to split with edge S[i]
        leaf.value ← value
        value += 1
        remainder -= 1
        if ifFirst then
            prevNode ← split
            ifFirst ← 0
        end if
        else
            split linked to prevNode
            prevNode ← split
        end if
        if active_node == root then
            active_edge ← S[index(active_edge) + 1]
            active_length -= 1 continue
        end if
        if active_node has suffix link then
            active_node ← the node active_node links to
        end if
        else
            active_node ← root
        end if
    end while
end for
return root
```

Time complexity: $\mathcal{O}(N)$

Applications

- **Longest Repeated Substring[3]**

  Construct a suffix tree, find the deepest non-leaf node. The path from root to the node is the longest repeated substring of the string.

- **Longest Common Substring[3]**

  Construct a suffix tree of "string1 + # + string2 + $", find the deepest non-leaf node. The path from root to the node is the longest common substring of string1 and string2.

- **Longest Palindromic Substring[2]**

  Construct a suffix tree of "string + # + reverse of string + $", find the deepest non-leaf node. The path from root to the node is the longest palindromic substring of the string.

References.

[1] brenden. *Visualization of Ukkonen's Algorithm*. http://brenden.github.io/ukkonen-animation/ Accessed October 5, 2020 (cit. on pp. 2–5).

[2] GeeksforGeeks. *Suffix Tree Application 6 – Longest Palindromic Substring*. https://www.geeksforgeeks.org/suffix-tree-application-6-longest-palindromic-substring/?ref=lbp Accessed October 5, 2020 (cit. on p. 7).

[3] Dr Steven Halim and Dr Felix Halim. *VisuAlgo.net – visualising data structures and algorithms through animation*. https://visualgo.net/en/suffixtree Accessed October 5, 2020 (cit. on pp. 1, 2, 7).

[4] jogojapan. *Ukkonen's suffix tree algorithm in plain English*. https://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english/9513423#9513423 Accessed October 5, 2020 (cit. on p. 2).

[5] E. Ukkonen. "On–line construction of suffix trees". In: *Algorithmica* 14.249-260 (1995). DOI: 10.1007/BF01206331 (cit. on p. 2).