Algorithm of Suffix Tree

by farseerfc@gmail.com

Jiachen Yang[1]

November 16, 2011

1 What is Suffix Tree

What can we do with suffix tree?

Linear algorithms for exact string matching

- like KMP

Search for strings

1. Check if a string P of length m is a substring in $O(m)$ time.
2. Find all z occurrences of the patterns $P_1, \cdots, P_q$ of total length m as substrings in $O(m+z)$ time.
3. Search for a regular expression P in time expected sublinear in n .
4. Find for each suffix of a pattern P, the length of the longest match between a prefix of $P[i \ldots m]$ and a substring in D in $\theta(m)$ time .
5. …

Find properties of the strings

1. Find the longest common substrings of the string S_i and S_j in $\theta(n_i + n_j)$ time .
2. *Find all maximal pairs, maximal repeats or supermaximal repeats in* $\theta(n+z)$ *time, if there are* z *such repeats .*
3. Find the Lempel-Ziv decomposition in $\theta(n)$ time .
4. Find the longest repeated substrings in $\theta(n)$ time.
5. Find the most frequently occurring substrings of a minimum length in $\theta(n)$ time.
6. Find the shortest strings from $\sum$ that do not occur in D, in $O(n+z)$ time, if there are z such strings.
7. Find the shortest substrings occurring only once in $\theta(n)$ time.

8. Find, for each i, the shortest substrings of S_i not occurring elsewhere in D in $\theta(n)$ time.
9. ...

Trie, Radix Tree, Suffix Trie & Suffix Tree

trie[1] (AKA prefix tree) is a dictionary tree.

- stores a set of words.
- each node represents a character except that root is empty string.
- words with common prefix share same parent nodes.
- minimal *deterministic finite automaton* that accepts all words.

radix tree (AKA patricia trie or radix trie) is a trie with compressed chain of nodes.

- Each internal node has at least 2 children.

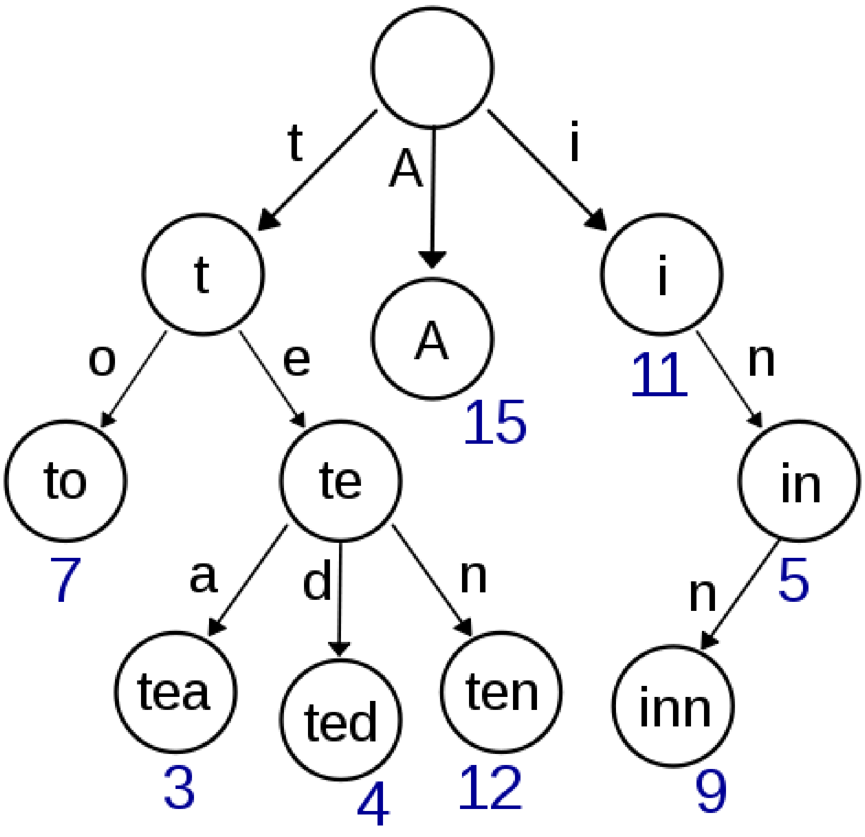
suffix trie is a trie which stores all suffix of a given string.

suffix tree is a suffix radix tree.

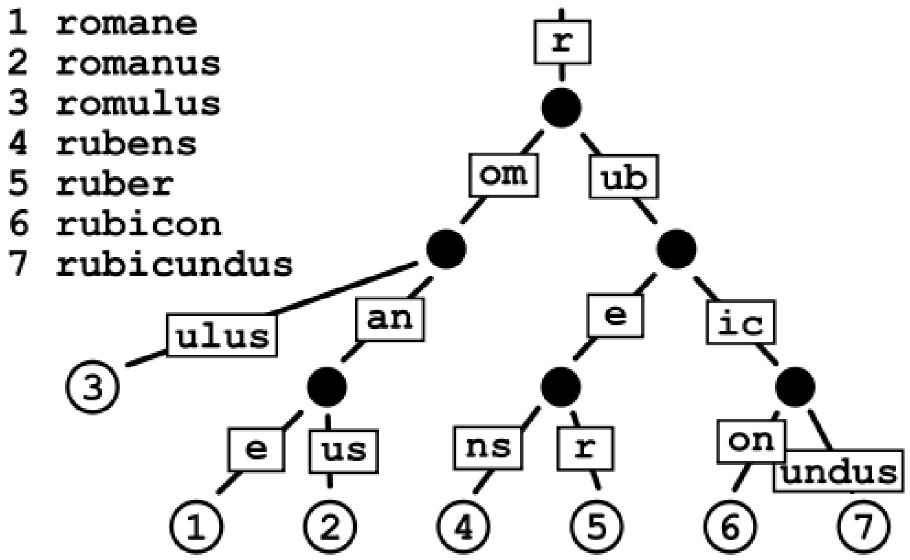
- that enables *linear time* construction and *fast* algorithms of other problems on a string.

Trie & Radix Tree

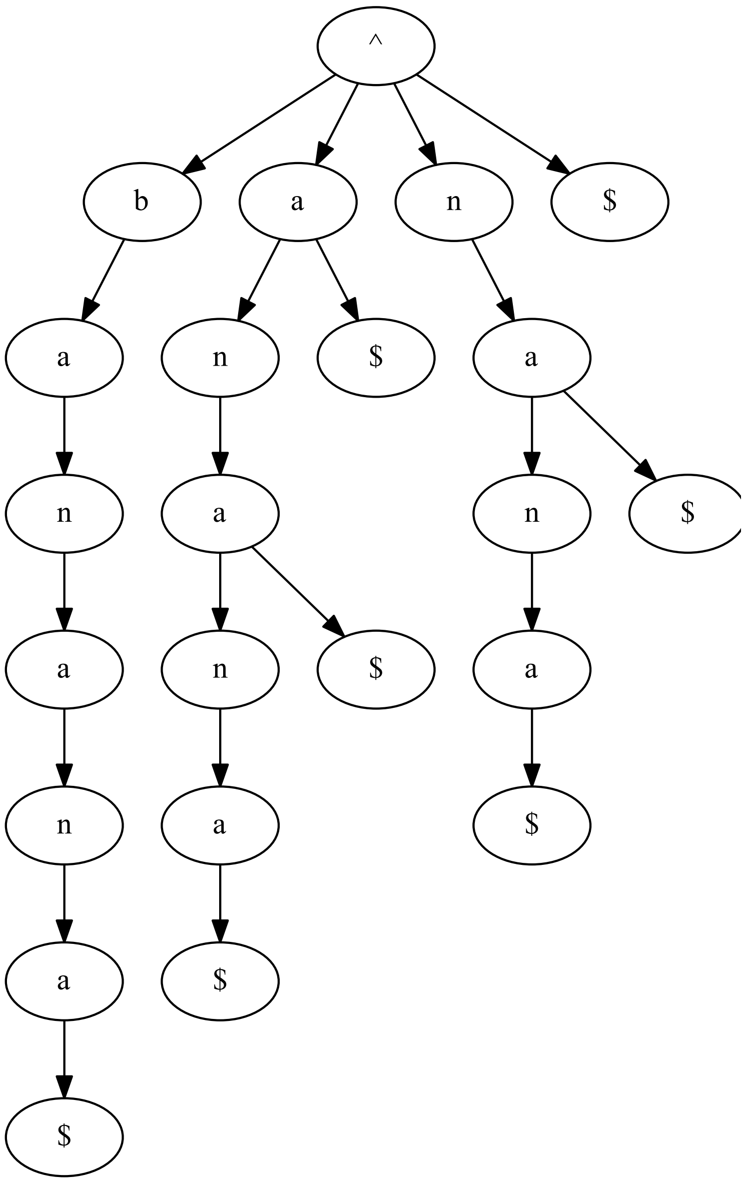
Trie of “A to tea ted ten i in inn”

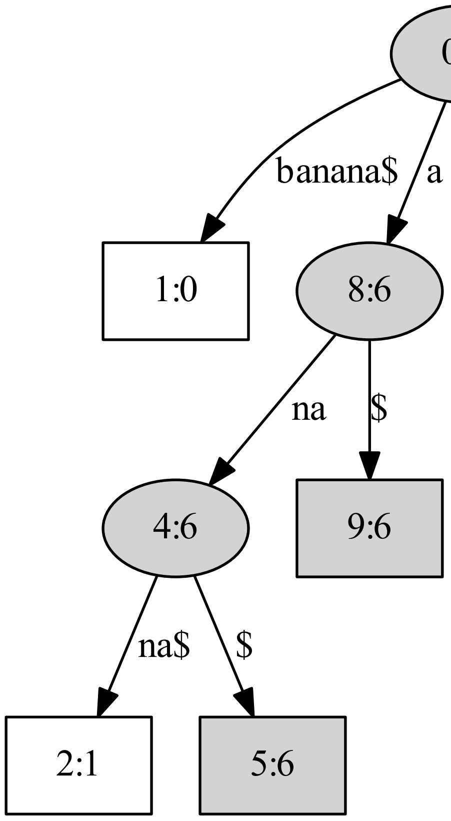


Radix tree example

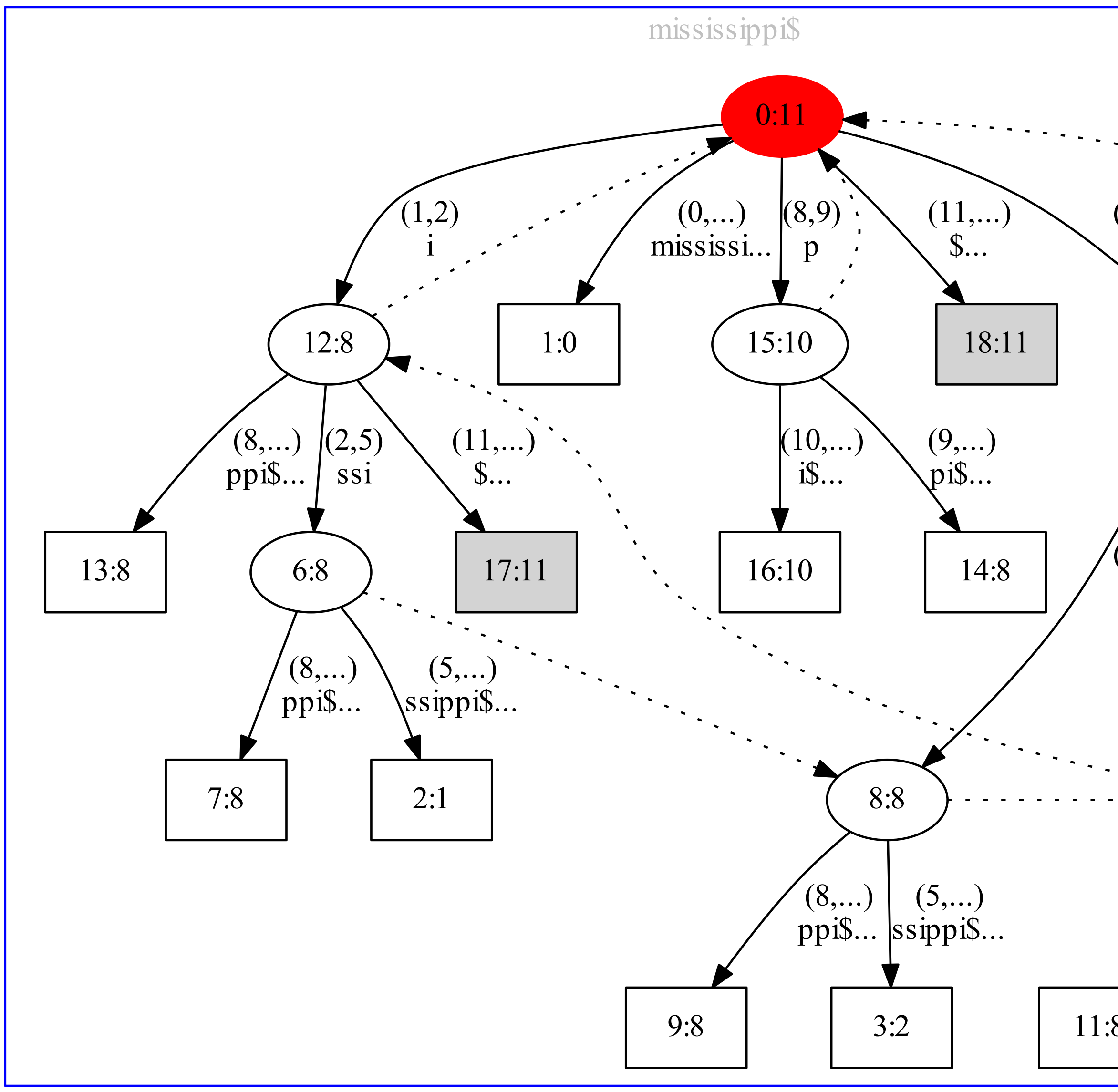


Suffix Trie & Suffix Tree of "banana"





Suffix Tree of “mississippi”



2 History & Naïve Algorithm

History of Suffix Tree Algorithms

- First linear algorithm was introduced by Weiner 1973 as position tree. Awarded by Donald Knuth as "Algorithm of the year 1973".
- Greatly simplified by McCreight 1976.

Above two algorithms are processing string *backward*.

- First online construction by Ukkonen 1995, which is easier to understand.

Above algorithms assume size of *alphabet as fixed* constant .

- Limitation was break by Farach 1997, optimal for all alphabets.

Further study are continued to scale to scenarios when the whole suffix tree or even input string cannot fit into memory.

References

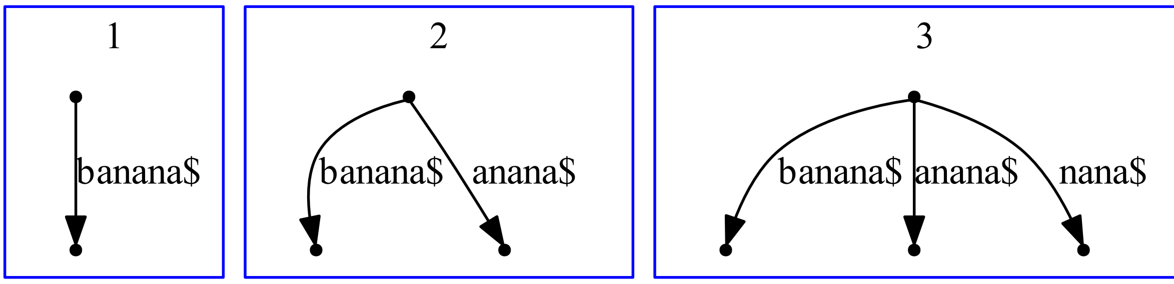
M. Farach. Optimal suffix tree construction with large alphabets. In *focs*, page 137. Published by the IEEE Computer Society, 1997.

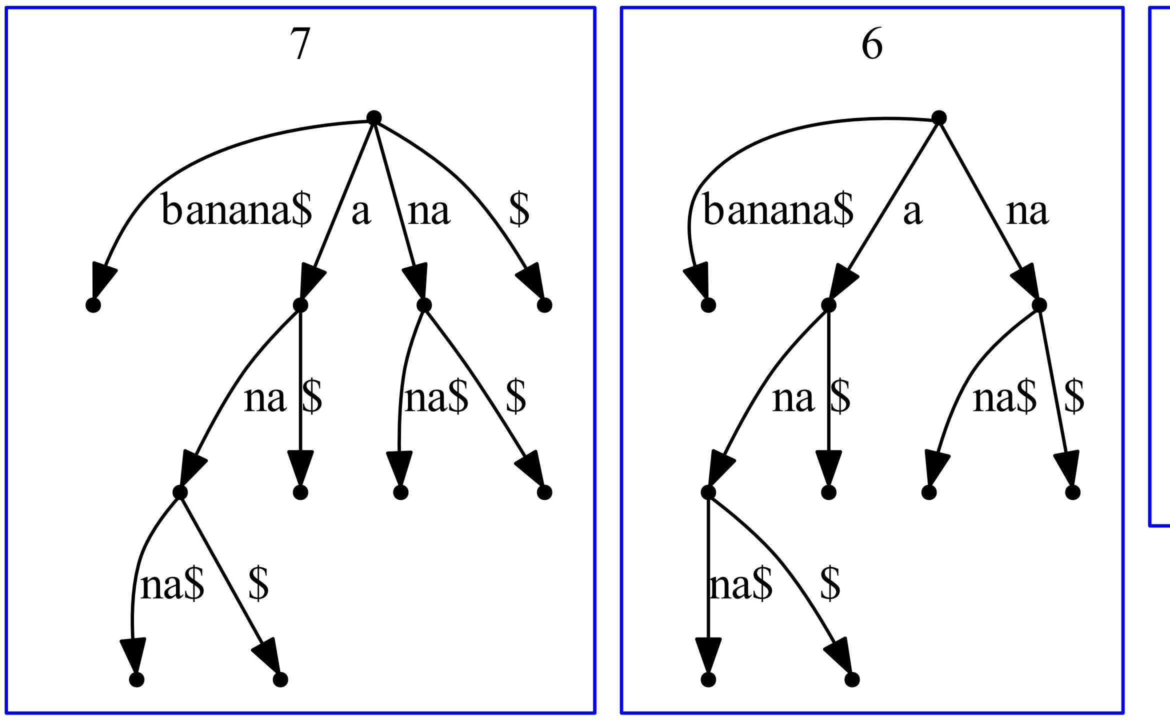
E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, April 1976. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321941.321946. URL `http://doi.acm.org/10.1145/321941.321946`.

E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995. ISSN 0178-4617. URL `http://dx.doi.org/10.1007/BF01206331`. 10.1007/BF01206331.

P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 1 –11, oct. 1973. doi: 10.1109/SWAT.1973.13.

Backward Construction of Suffix Tree





Construct Suffix Tree by Sorting Suffix

Suffix :

```
mississippi
ississippi
ssissippi
sissippi
issippi
ssippi
sippi
ippi
ppi
pi
i
```

Sorted suffix :

```
i
ippi
issippi
ississippi
mississippi
```

```
pi
ppi
sippi
sissippi
ssippi
ssissippi
```

Tree of sorted suffix :

```
|-i->|-
|    |-ppi
|    |-ssi->|-ppi
|           |-ssippi
|-mississippi
|-p->|-i
|    |-pi
|-s->|-i-->|-ppi
     |     |-ssippi
     |-si->|-ppi
           |-ssippi
```

Time complexity will be $O(N^2 \log N)$.
Space complexity will be $O(N^2)$.

Naïve Algorithm

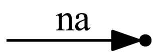
```
SUFFIXTREE(string) 1  for $i \leftarrow 1$ to length(string)
                   2      do UPDATE($tree_i$)          ▷ Phrase i

UPDATE($tree_i$) 1  for $j \leftarrow 1$ to $i$
                 2      do $node \leftarrow tree_i$.FIND($suffix[j$ to $i-1]$)
                 3         EXTEND($node, string[i]$)      ▷ Extension j
```

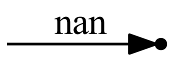
Time complexity will be $O(N^3)$.
Space complexity will be $O(N^2)$.
The challenge is to make sure $tree_i$ is updated to $tree_{i+1}$ *efficiently*.

Suffix Extend Cases of Naïve Algorithm

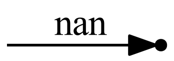
Case I If path $S_{j...i}$ ends at leaf, append a char S_{i+1} to end of edge into leaf. $S_{j...i} = \dots na$



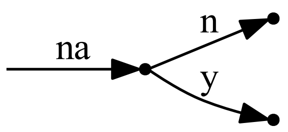
➔ $S_{j...i+1} = \dots nan$



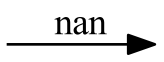
Case II If path $S_{j...i}$ ends in the middle of an edge , and next char S_{i+1} is *not equal* to the next char in the edge, split that edge, create a internal node, add a new edge to a new leaf. $S_{j...i} = \ldots na$



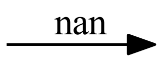
➔ $S_{j...i+1} = \ldots nay$



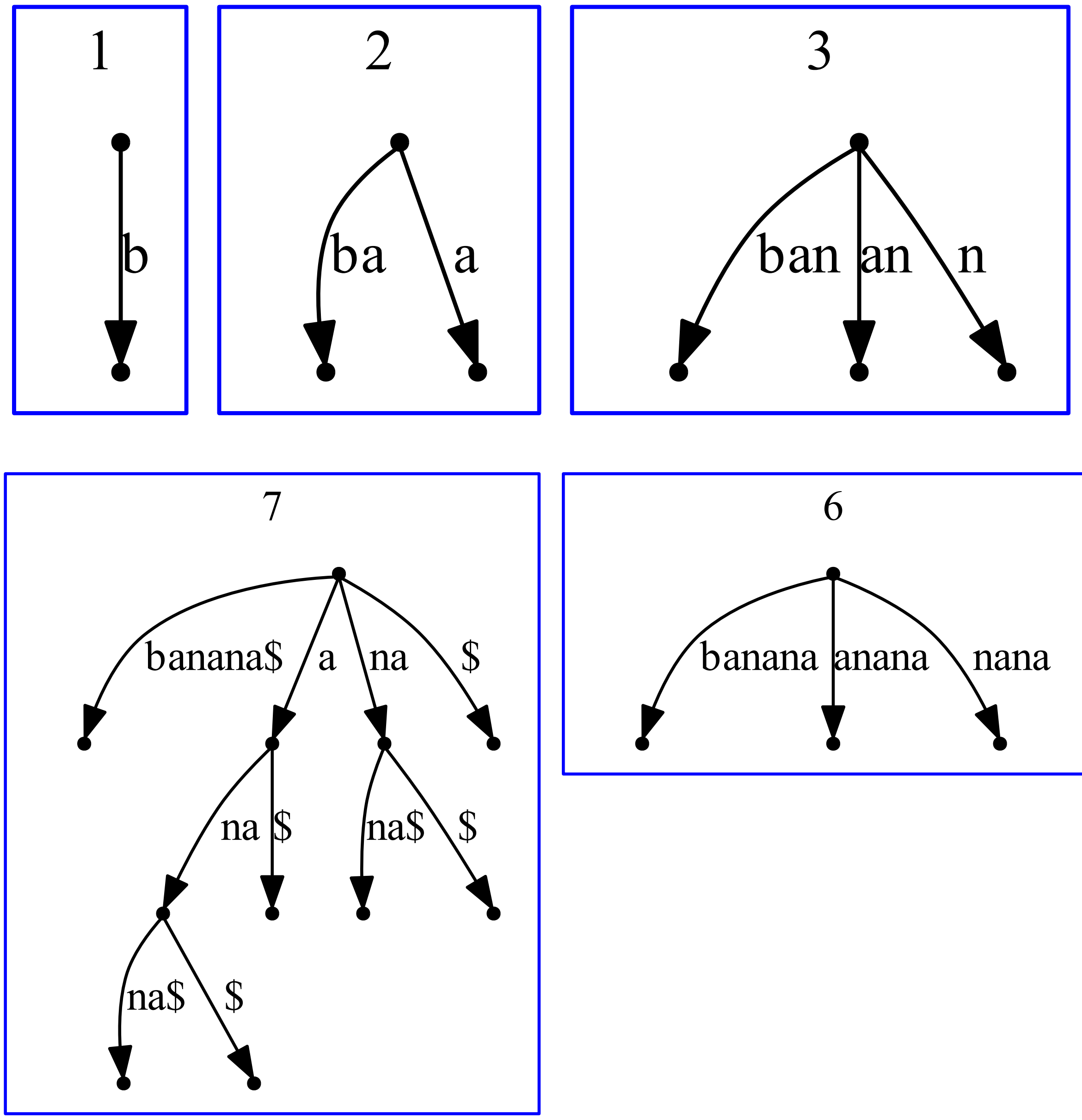
Case III If path $S_{j...i}$ ends in the middle of an edge , and next char S_{i+1} is *equal* to the next char in the edge, do nothing, extenstion has done. $S_{j...i} = \ldots na$



➔ $S_{j...i+1} = \ldots nan$



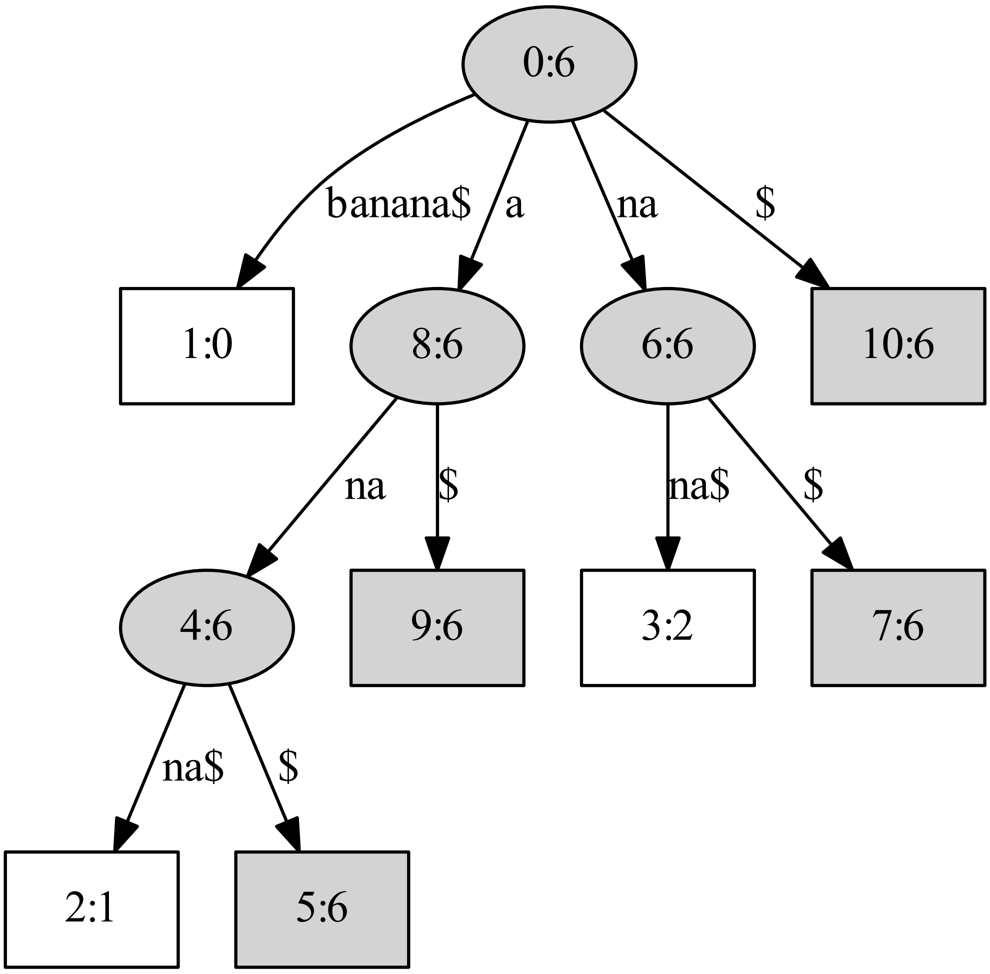
Naïve Online Construction of Suffix Tree



Properties of Suffix Tree

1. Each update will add exactly 1 leaf node .
   - $nr\_leaf = N$

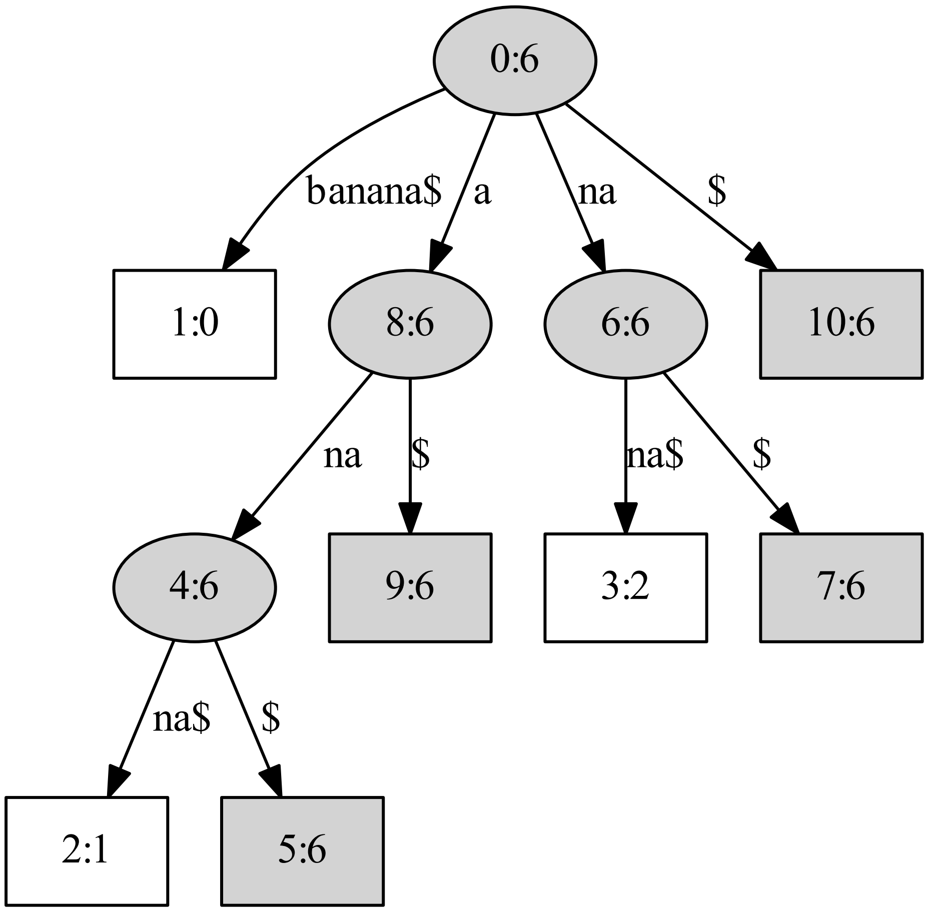
2. Suffix tree is full tree.
    - Each internal node has at least 2 children.
    - *nr_internal* $< N$
    - *nr_node* $< 2N$
3. Worst case Fabonacci word
    - abaababaabaab
4. Suffix is either explicit or implicit.
    - Explicit when it ends at a node.
    - Implicit when it ends in the middle of an edge.

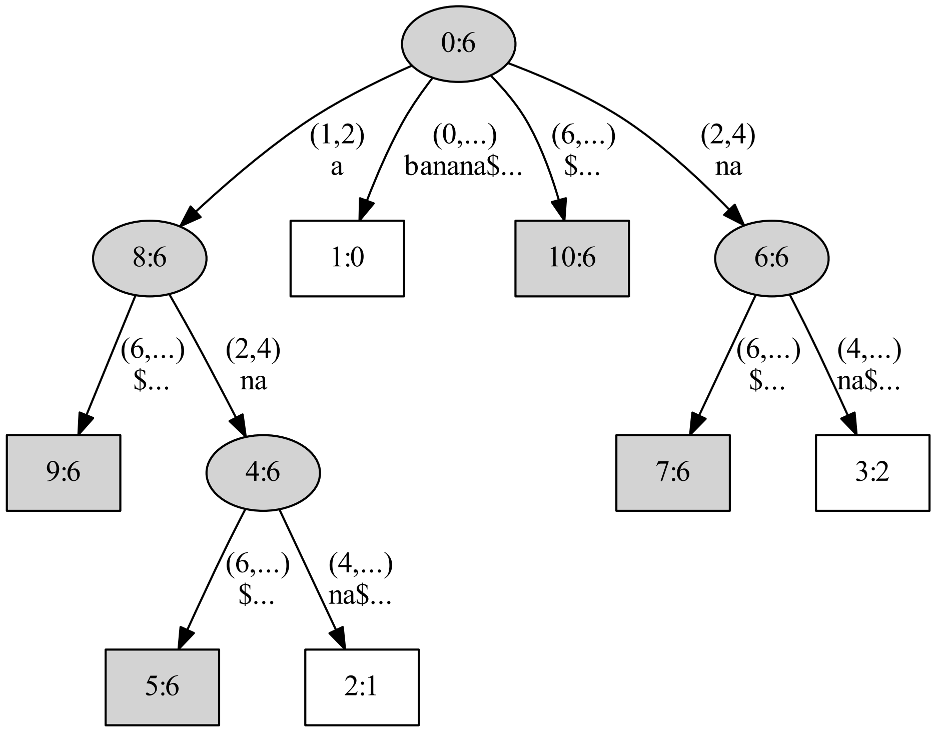


3 Optimization of Naïve Algorithm

Optimization of Naïve Algorithm

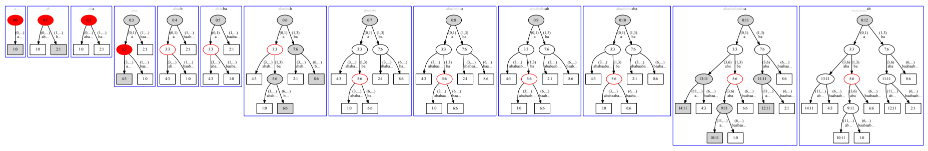
1. Substrings can be represented as (start, end) pair of their index in orignal string.
    - Reduce space complexity to $O(N)$ if size of alphabet is fixed constant.
2. Once a leaf, Always a leaf
    - Represent edge that links to a leaf as (start, $\cdots$).
    - Extend leaf nodes *for free*. We do not need Extend Case I.





Active Point

- During a phrase, if we meet Extend Case III, that is if we found $S[i+1]$ already exists in $suffix[j\dots i]$ then $S[i+1]$ will exists in $\forall suffix[k\dots i], k \in j\dots i$.
- Thus Case III is a sign that means update of this phrase is finished.
- During phrase i if we stopped at $suffix[k\dots i]$ by Case III, then in next phrase we can start from $suffix[k\dots i+1]$ because all suffix start with $1\dots k-1$ will end at Case I.
- We called this point(current internal node, current position k in string) as *Active Point*.

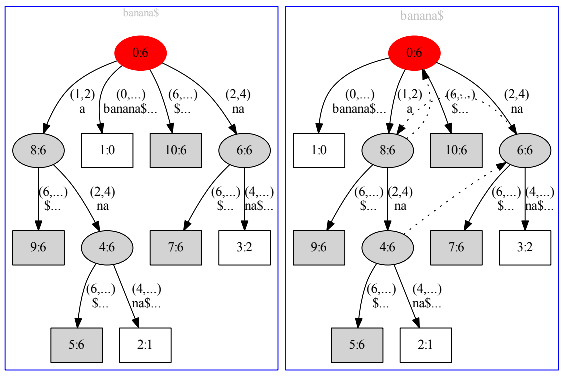


Ukk's Update using Active Point

```
UPDATE($tree_i$)
 1  $current\_suffix \leftarrow active\_point$
 2  $next\_char \leftarrow string[i]$
 3  while True
 4      do if there exists edge start with $next\_char$
 5             then break          ▷ Case III
 6             else
 7                 split current edge if implicit
 8                 create new leaf with new edge labelled $next\_char$
 9          if $current\_suffix$ is empty
10             then break
11             else  $current\_suffix \leftarrow$ next shorter suffix
12  $active\_point \leftarrow current\_suffix$
```

Suffix Link to find next shorter suffix

- Suffix link
  - Internal node of suffix $X\alpha$ has a link to node α.
  - If α is empty, suffix link points to root.
- How to create suffix link
  - Link together every internal nodes that are created by splitting in same phrase.



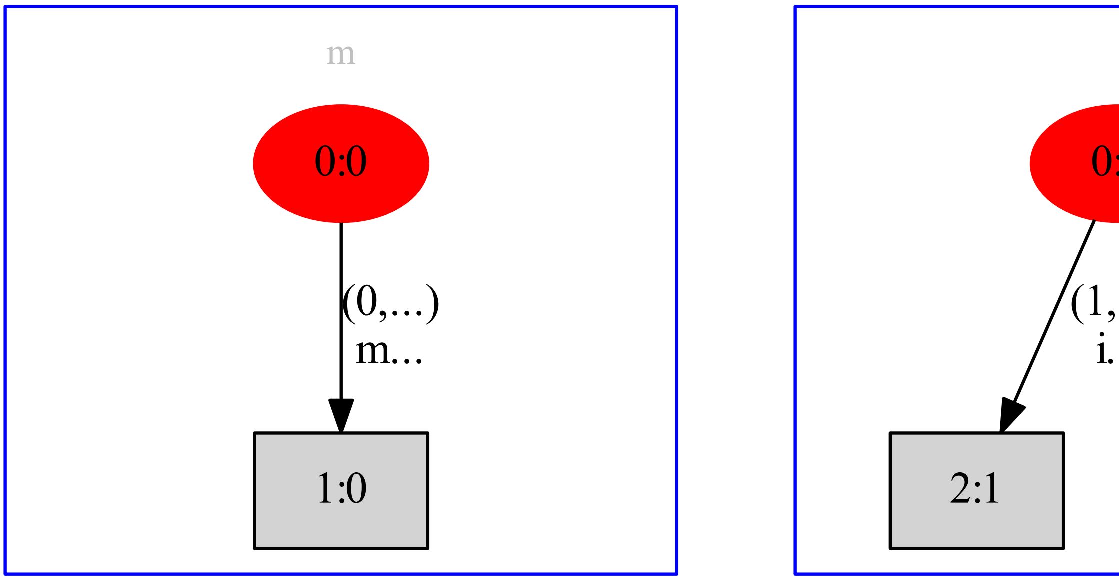
Fast jump using Suffix Link

- Assume we are at Suffix $X\alpha\beta$, whose parent internal node represent $X\alpha$.
  1. Go back to parent internal node,
  2. Jumping follow the node's suffix link to the node represent Suffix α
  3. Go down to Suffix $\alpha\beta$.

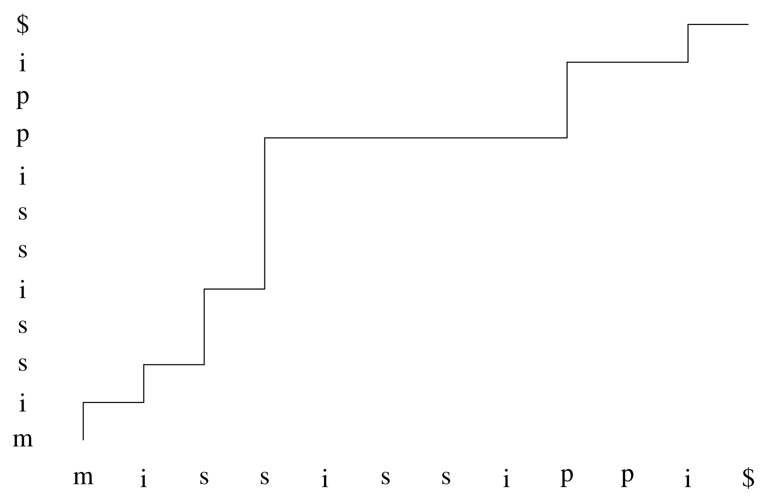
- Even jump down in step 3 because we already know length of β. (Skip/Count trick)
- Combining all these tricks we can Extend a character in $O(1)$ time.

4 Examples & Analysis

Experiment – mississippi

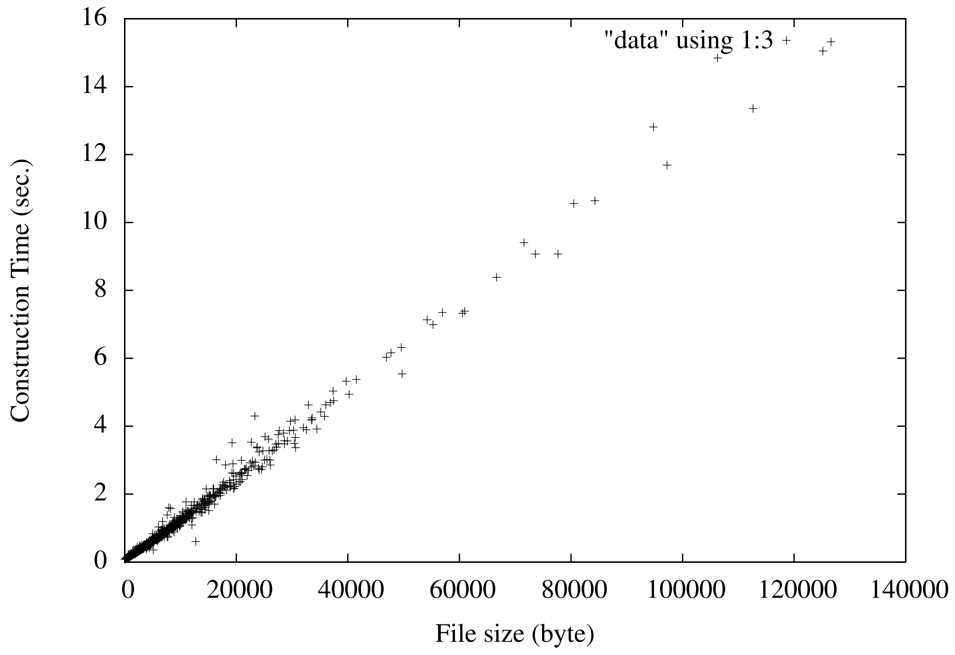


Time Complexity Analysis



Time complexity is $2N = O(N)$.

Experiment – English text


"data" using 1:3
Construction Time (sec.)
File size (byte)
0
2
4
6
8
10
12
14
16
0
20000
40000
60000
80000
100000
120000
140000