Applications of Suffix Trees

Charles Yan

2008

1. Exact String Matching

|P|=n, |T|=m

- P and T are both known at the same time
  - Boyer-Moore, or Suffix trees. O(n+m)
- T is known and kept fixed. P varies.
  - Suffix trees, O(m) in preprocess, O(n+k) in searching
- P is known and kept fixed. T varies.
  - Boyer-Moore, O(n) in preprocess, O(m) in searching

2. Exact Set Matching

$|T|=m$, $P=\{p_1, P_2, \ldots, p_i\}$, $\Sigma|p_i|=n$

- Aho-Corasick
  $O(m+n+k)$

- Suffix trees.
  $O(m)$ in building suffix tree
  $O(n_i+k_i)$ in searching for p_i
  $O(m+\Sigma n_i+ \Sigma k_i)$ for all P, i.e. $O(m+n+k)$

3. Substring problem for a set of texts

Motivation 1:

T is a DNA database containing millions of DNA sequences that have been previously sequenced.

Given a new DNA sequence, to determine whether it has been previously sequenced.

(1) Concatenate all T together, then use Boyer-Moore O(m+n+k) for searching each P, m is huge!

(2) Build a suffix tree for each T_i
O(m) for total preprocessing, but O(i* n+k) for searching each P, i is in the order of 10^6!

Substring problem for a set of texts

Motivation 2:

To identify the remains of military personnel

For each soldier, a set of DNA sequences (T= $\{T_1, T_2, \ldots, T_i\}$) is kept when he/she joins the army. (The whole genome sequence is very difficult to obtain for technical reasons.)

A DNA sequence (P) is extracted from the remains of personnel that have been killed.

To determine whether the remains belong to soldier A, we just need to see whether P matches any sequence in the T of A.

3. Substring problem for a set of texts

Given $T=\{T_1, T_2, \ldots, T_i\}$, $\Sigma|T_i|=m$, $|P|=n$, set T is fixed, P varies. O(m) preprocessing time is allowed. For each coming P, to find all occurrences of P in all T with O(n+k) time

For each given P, this a the reverse of exact set matching problem.

(1) Concatenate all T together, then use Boyer-Moore

O(m+n+k) for searching each P

(2) Build a suffix tree for each T_i,

O(m) for total preprocessing, but O(i* n+k) for searching each P

(3) Build a suffix tree (generalized suffix tree) for the set T,

the searching will take O(n+k) time

but how to build the a generalized suffix tree in O(m)?

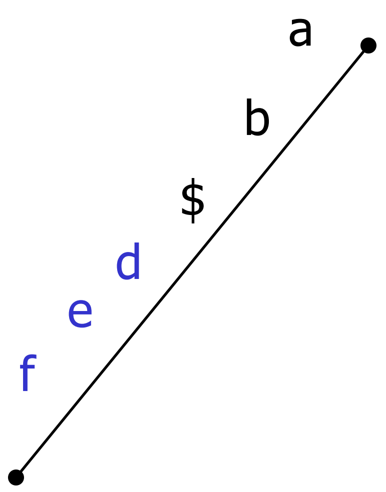
Generalized Suffix Trees

How to build the generlized suffix tree for a set T = {T_1, T_2, …, T_i) in O(m)?

(1) Append a marker to the end of each string and concatenated them together to build a new string S.

(2) Build a suffix tree for S.

(3) But, suffixes span multiple T_i,



Generalized Suffix Trees

Minor subtleties

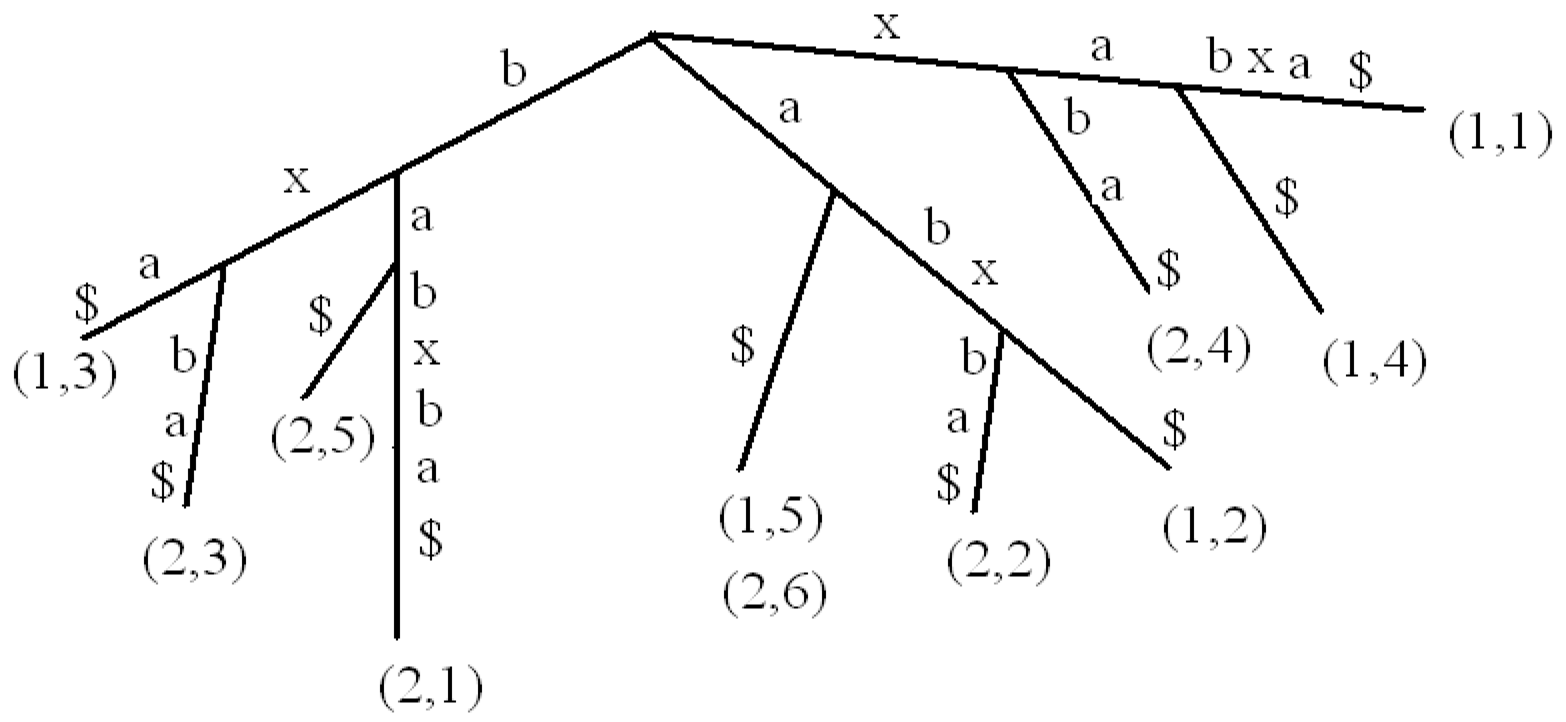
(1) Each edge is associated with three indices (i,p,q), where indicates that the substring come from T_i. p and q are the begin and end positions.

(2) Suffixes from two texts may be identical. Thus, each leaf is associated with labels indicating all of the strings and starting positions of the associated suffix.

Generalized Suffix Trees

T_1: xabxa

T_2: babxba



Generalized Suffix Trees

How to build the suffix tree for a set T = {T_1, T_2, …, T_i) in O(m)?

(1) Build a suffix tree for T_1\$

(2) Start from the root of the tree search for T_2. Assume that i characters in T_2 are matched,

- The suffix tree has implicitly encoded every suffix of $T_2[1,..i]$
- The suffix tree contains I_i for T_2
- We can skip phase 1,..,i for T_2

(3) Continue the Ukkonen's algorithm on T_2 in phase i+1

- Walk up from the end of $T_2[1,..i]$, …

(4) Until all T_i are included in the suffix tree.

4. Longest Common Substring (LCS) of Two Strings

Given strings S1 and S2, find the LCS of them.

Different from *longest common subsequence problem.*

S_1: *xabxa*

S_2: *babxba*

LCS is *abx*

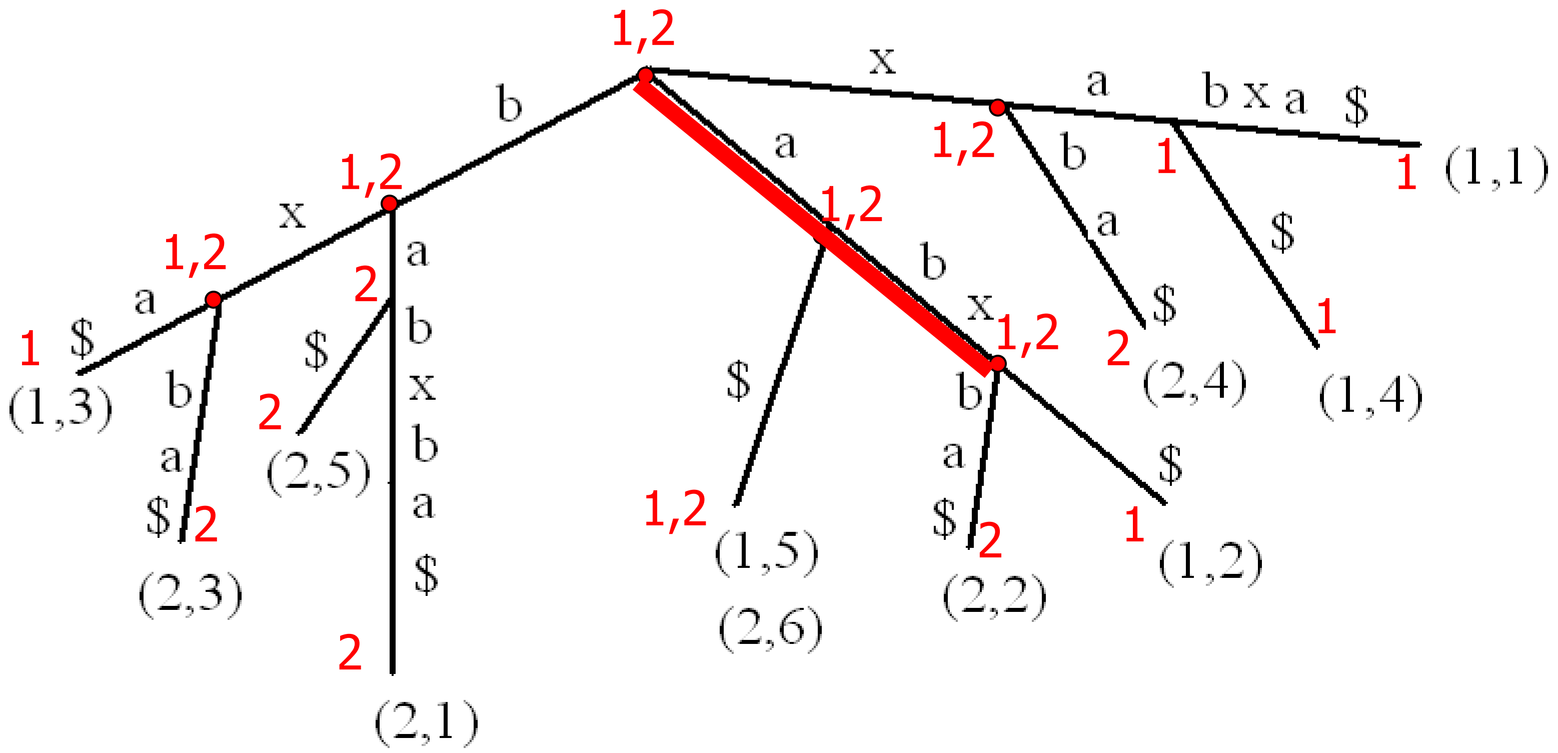
4. Longest Common Substring (LCS) of Two Strings

- Build a generalized suffix tree for S1 and S2
- If a leave is from S_1, then mark all its ancestors with 1.
- If a leave is from S_2, then mark all its ancestors with 2.
- The path-label of any node that is marked with both 1 and 2 is a common substring of S_1 and S_2.
- Find the node that is labeled with 1 and 2, and has the greatest string-depth (number of characters on the path to it).

Generalized Suffix Trees

T_1: xabxa

T_2: babxba

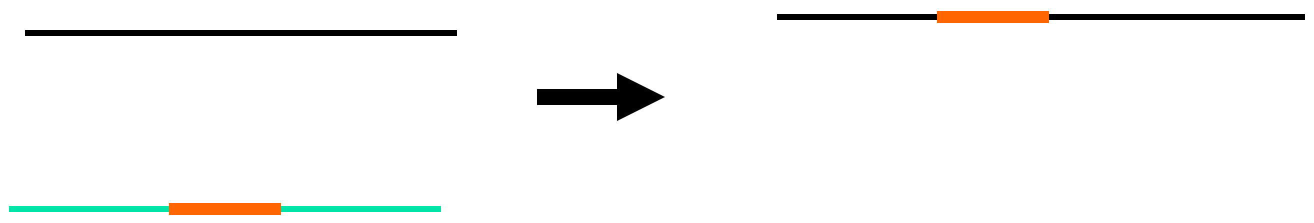


4. Longest Common Substring (LCS) of Two Strings

(1) O(m) for building generalized suffix tree
(2) O(m) for calculating the string-depth of each node (e.g. Breadth first)
(3) O(m) for marking node with 1 or 2 (e.g. Depth first)
(4) O(m) finding the longest.

5. DNA Contamination Problem

DNA contamination: During laboratory processes, unwanted DNA inserted into the DNA of interest.



Contamination sources: Human, bacteria,…

DNA from Dinosaur bone: More similar to human DNA than to bird and crockodilian DNA

5. DNA Contamination Problem

S: DNA of interest

P: DNA of possible contamination source

If S and P share a common substring longer than ℓ, then S has been contaminated by P.

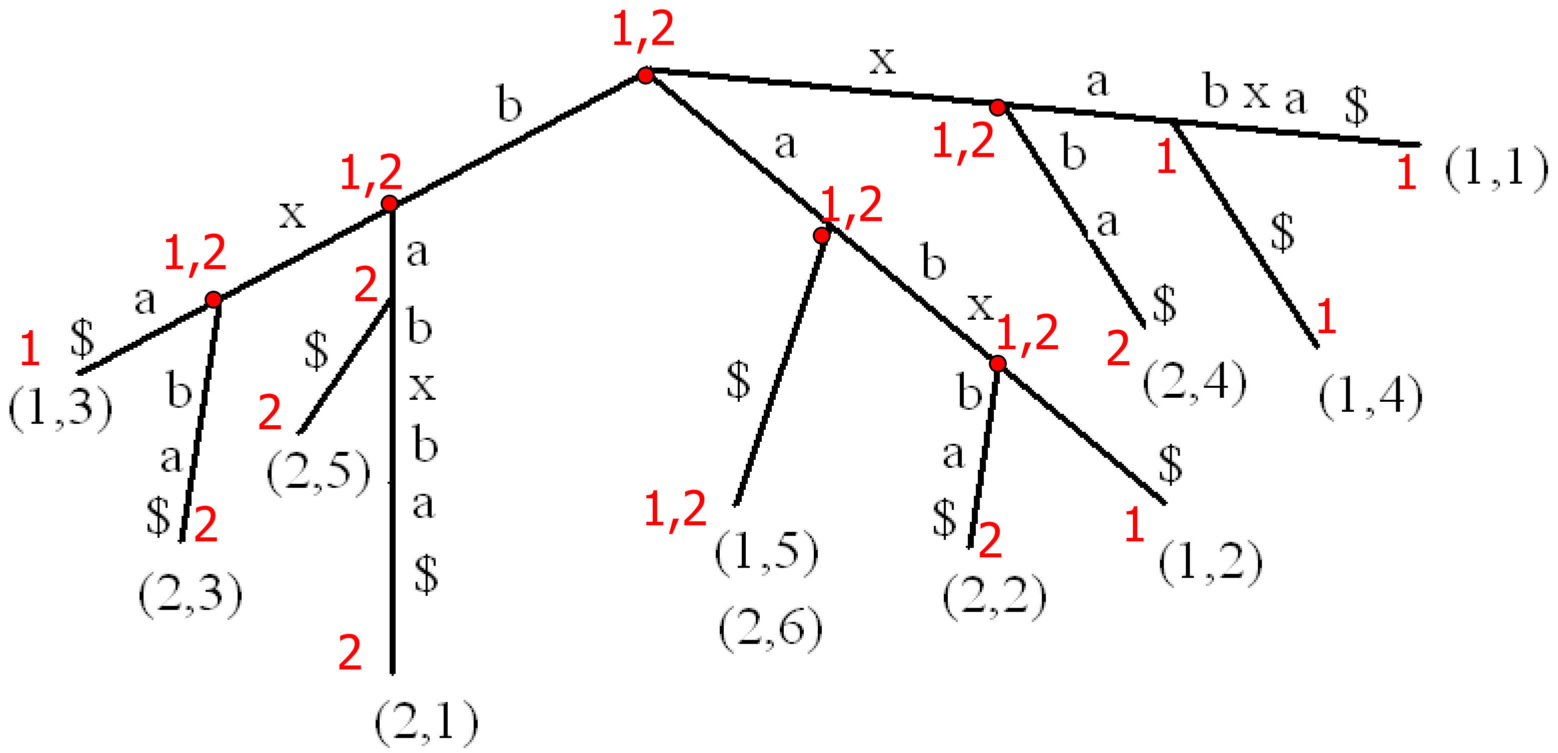
To find all common substrings of S and P that are longer than ℓ.

In general, P is set of DNA that are potential contamination sources.

Generalized Suffix Trees

T_1: xabxa

T_2: babxba



6. Common Substrings Of More Than Two Strings

Motivation

alignment

```
ALRDFATHDDF
SMTAEATHDSI
ECDQAATHEAS
```

regular expression

A-T-H-[D,E]

6.Common Substrings Of More Than Two Strings

Problem statement: Given K strings whose lengths sum to n, let $l(\mathrm{i})$ be the length of the longest substring common to at least i strings, to compute a table of K-1 entries, where entry i give $l(\mathrm{i})$ and one of the common substrings of that length (and that is shared by at least i strings)

{sandollar, sandlot, handler, grand, pantry}

i	$l(\mathrm{i})$	substring
2	4	sand
3	3	and
4	3	and
5	2	an

6. Common Substrings Of More Than Two Strings

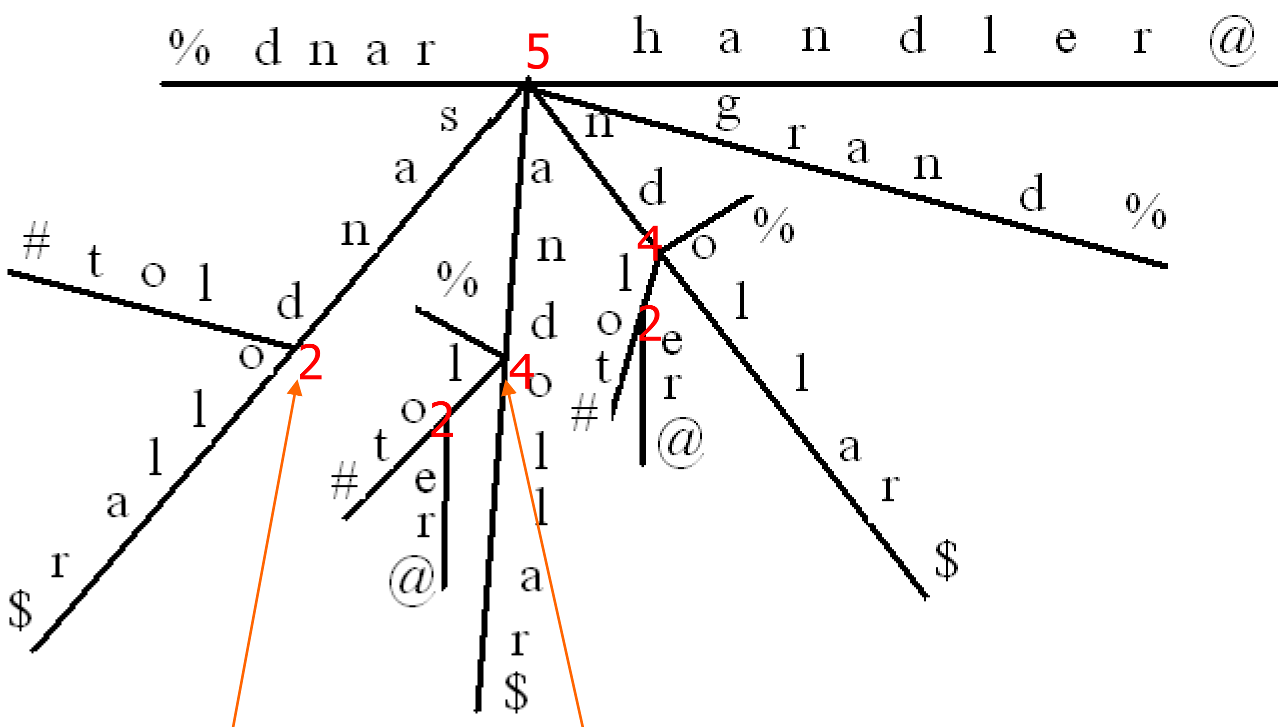
It can be solve in O(n) time.

But, an easy algorithm that uses O(kn) time first.

- Build a generalized suffix tree for the k strings giving each string a unique end marker.
- Each leaf belong to only one string
- For a node (v), let c(v) be the number of distinct string identifiers that appear at the subtree below it.
- V is a vector with V(i) denoting the length of the longest substring that occurs exactly in i strings (and a pointer to the node).
- From V(i) compute *l* (i),

```
for i=k; i>1; i—
        if (V(i)<V(i+1)), then l(i)= V(i+1)
        else l(i)= V(i)
```

6. Common Substrings Of More Than Two Strings



V		4	2	3	2
l		4	3	3	2

6. Common Substrings Of More Than Two Strings

Calculating c(v) is the bottle neck.

Can’t just count the number of leaves below it.

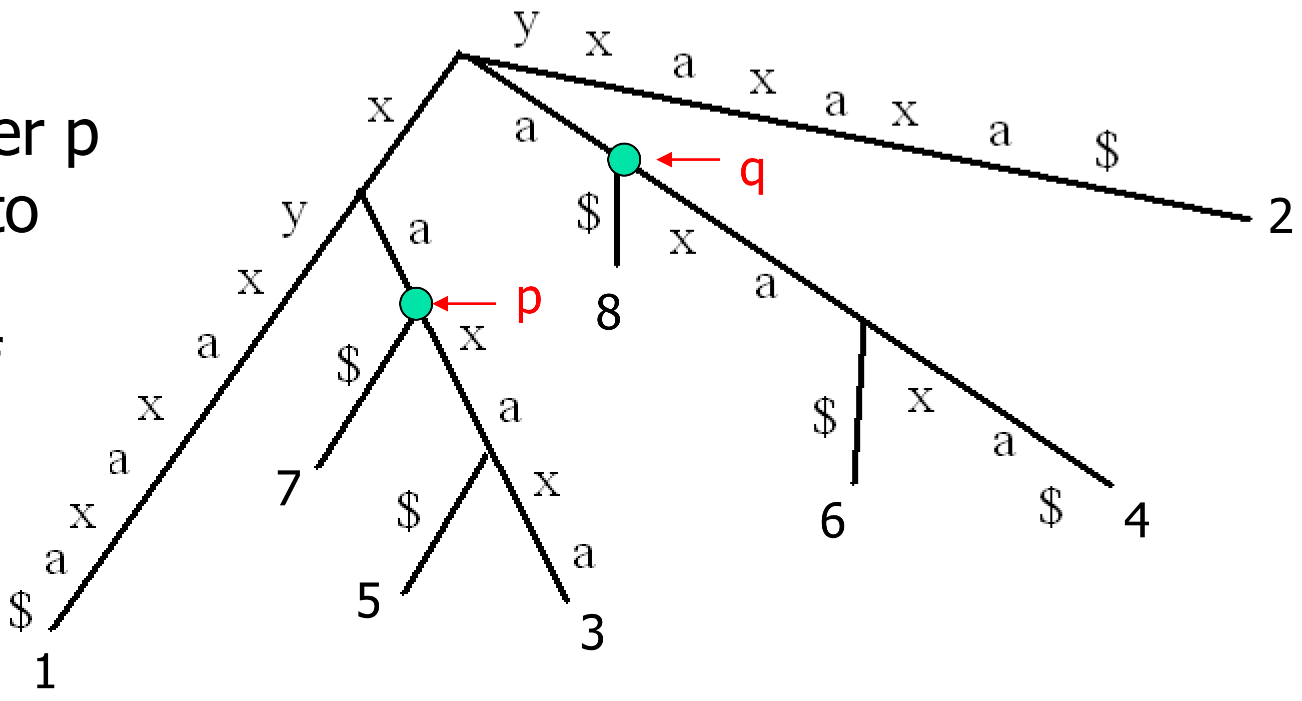
- For each node keep a C vector of k bits, with one bit correspond to one string.
- ith is set to 1 if a leave that belongs to ith string appear below the node
- The V vector of a parent is obtained by ORing the vectors of its children.
- n nodes.
- O(Kn) in calculating c(v).

Suffix Trees to DAGs

Space is a big problem for suffix trees.

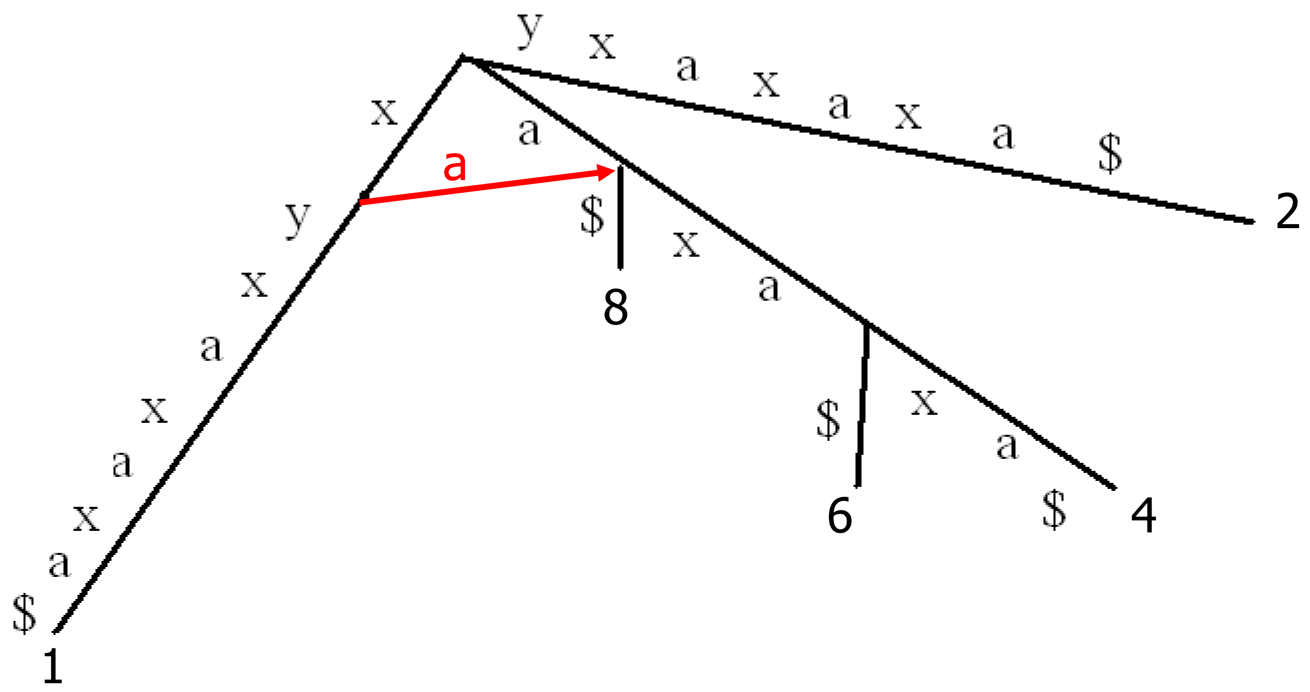
S: xyxaxaxa$

The subtree under p is isomorphic to that under q except for leaf labels



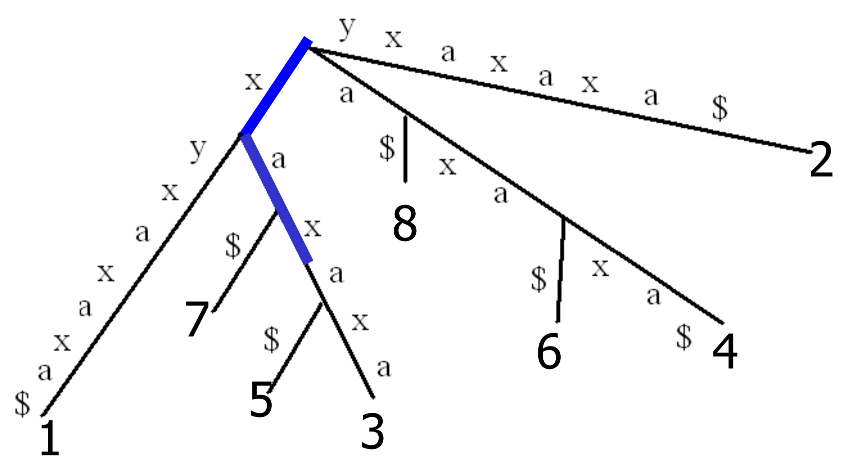
Suffix trees to DAGs

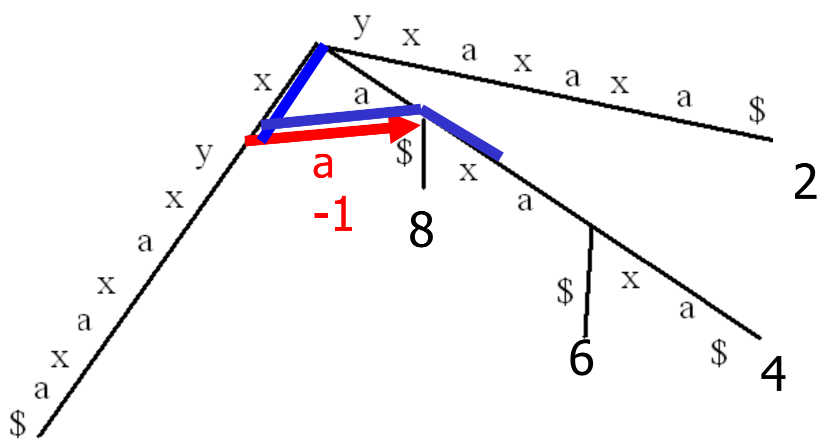
Directed acyclic graph (DAG)



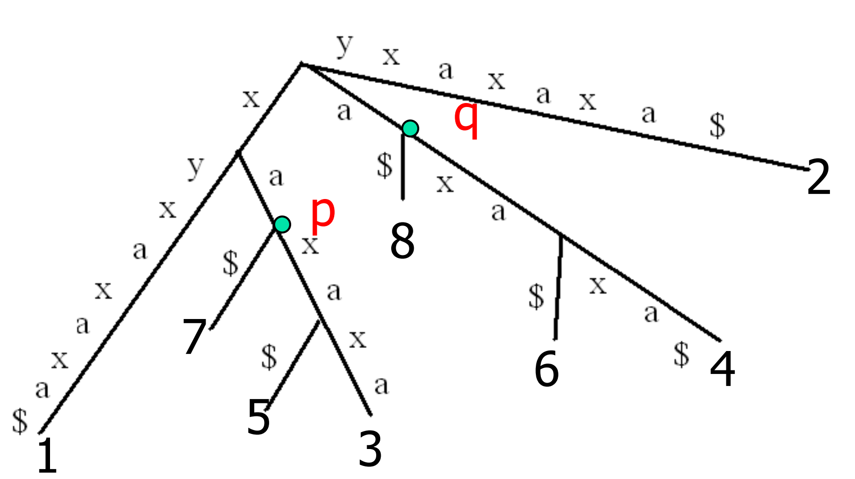
Suffix Trees to DAGs

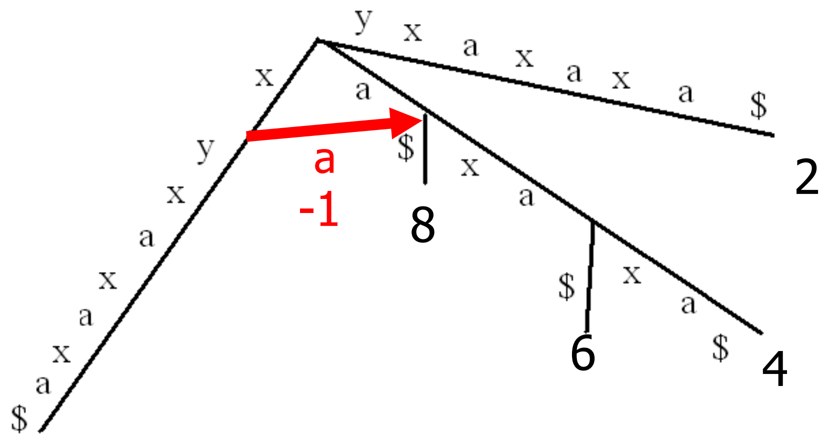
```
S: xyxaxaxa$
P:   xax
```





Suffix Trees to DAGs





If the subtrees under p and q are isomorphic (except leaf lables) and stringdepth(p)> stringdepth(q), then

- Merge p into q, by adding a direct edge from parent(p) to q
- Associated the directed edge with d=stringdepth(q)- stringdepth(p)
- When search for P in the S (text), let i be the leaf below the path labeled with P, if the directed edge is traversed then P occurs at i+d, otherwise P occurs at i.

Suffix Trees to DAGs

How to determine whether a subtree is isomorphic to another one?

Theorem 7.7.1

In suffix tree T the subtree below a node p is isomorphic to the subtree below a node q if and only if

- there is a directed path of suffix links from one node to the other node and
- the numbers of leaves in the two subtrees are equal.

A if and only if B

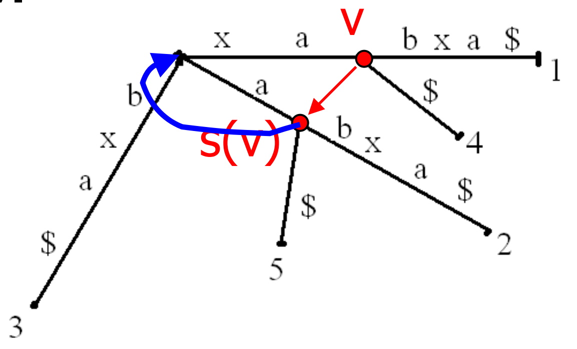
B→A

A→B

Ukkonent Algorithm

Suffix links

Let xα denote an arbitrary string, where x denotes a single character and α denotes a (possible empty) substring. For an internal node v with path-label xα, if there is another node s(v) with path-label α, then a pointer from v to s(v) is called a suffix link, denoted as (v,s(v)).



The root has no suffix link from it.

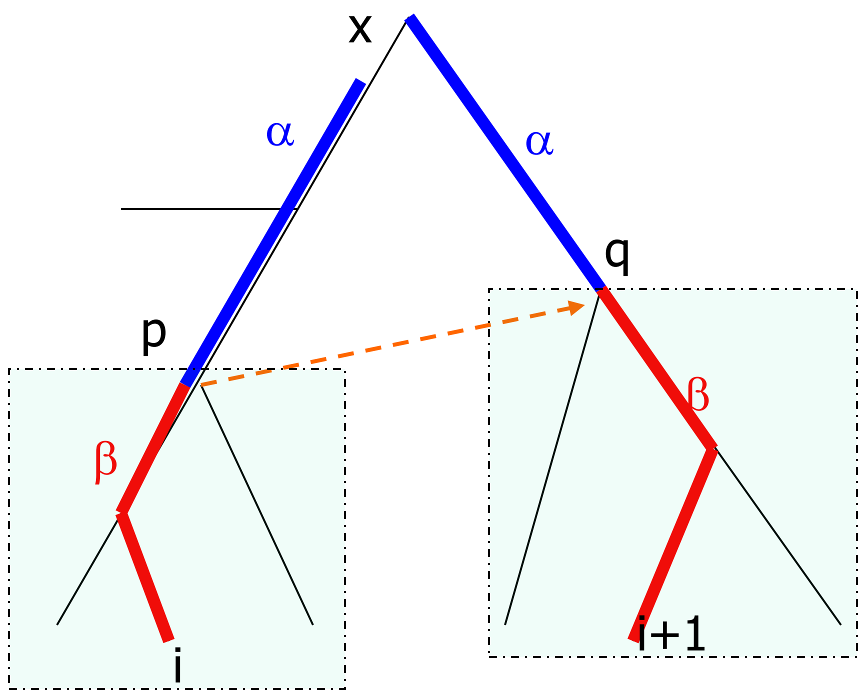
If α is empty, then the suffix link points the root.

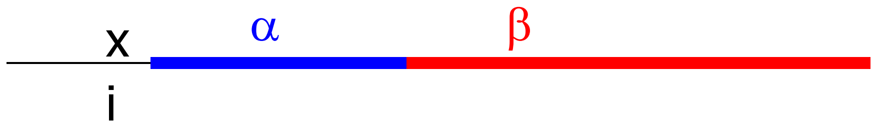
Suffix Trees to DAGs

B→A

Only one suffix link

For every path from p to a leaf in its subtree, there is an identical path from q to a leaf in its subtree.



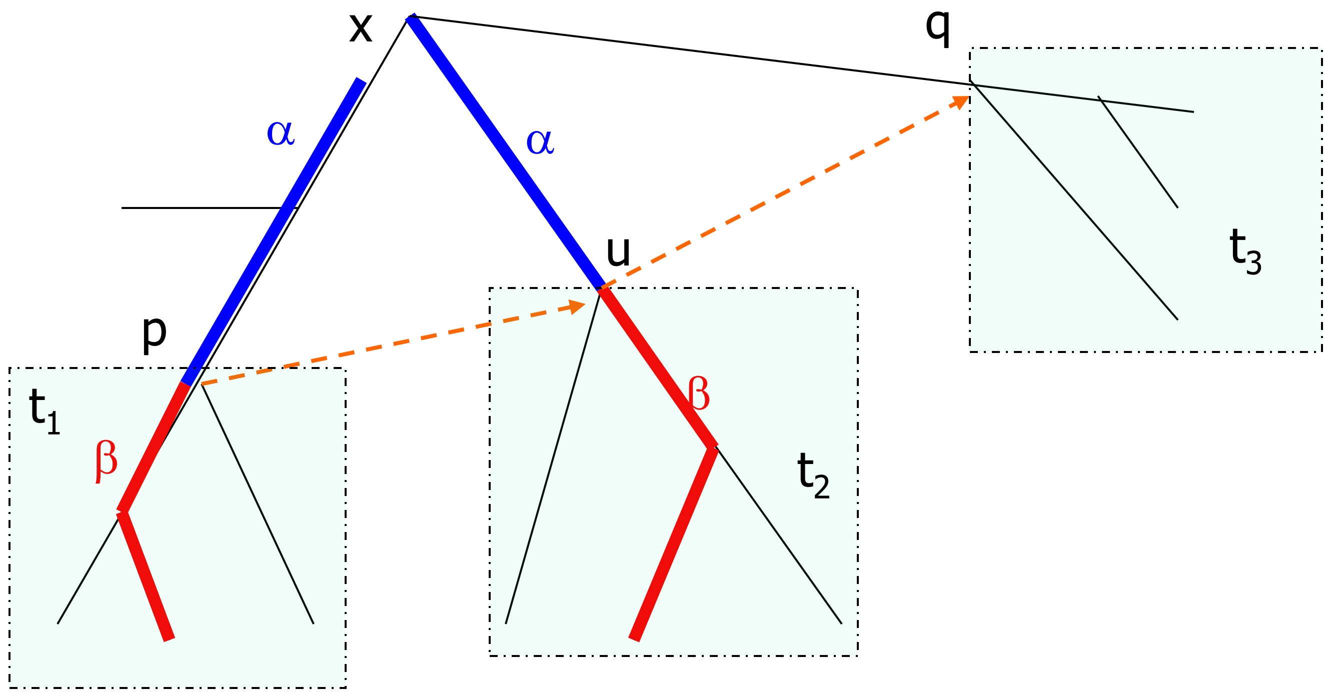


Suffix Trees to DAGs

B→A

A path of suffix links

For every path from p to a leaf in its subtree, there is an identical path from q to a leaf in its subtree.

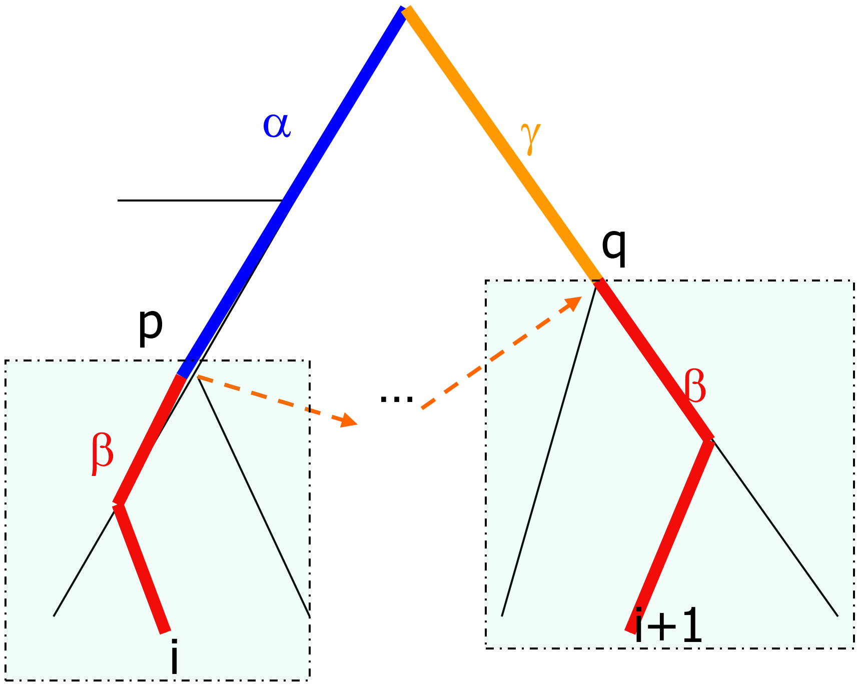


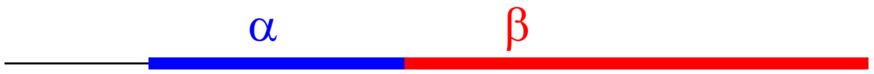
Suffix Trees to DAGs

A→B

Either α is a proper suffix of γ or γ is a proper suffix of α

There is a directed path of suffix links from one node to the other.





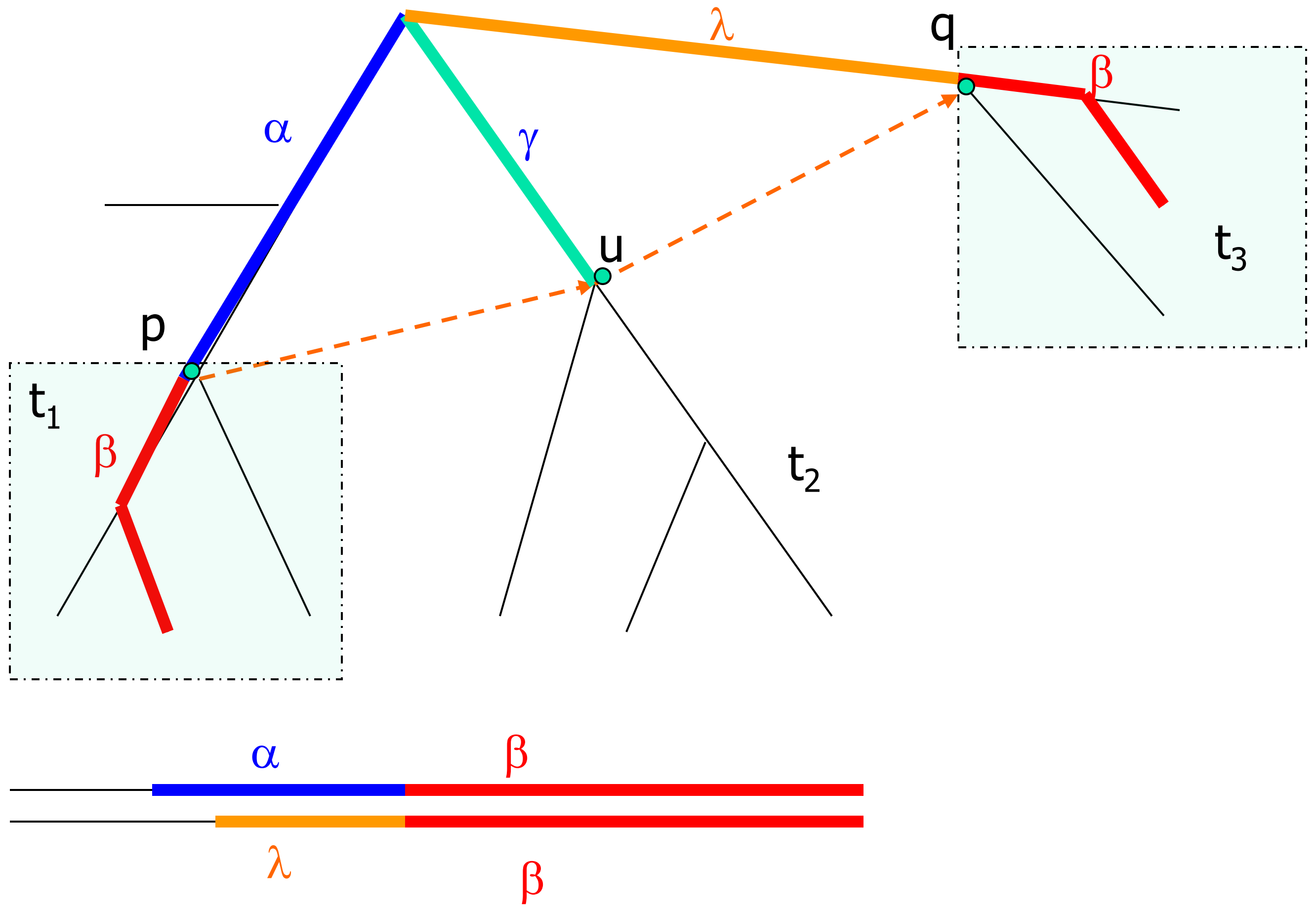
Suffix Trees to DAGs

A→B

Either α is a proper suffix of γ

or γ is a proper suffix of α

There is a directed path of suffix links from one node to the other.



Suffix Trees to DAGs

Let Q be the set of all pairs (p,q) such that there is a suffix link from p to q.

```
While there is a pair (p,q) in Q
    Merge p into q;
    Remove (p,q);
```

The merge of the pairs can be done in arbitrary order.

In practice, we can start merge in a top-down approach (depth-first).

Suffix Arrays—more space reduction

Given a m-character string T, a suffix array for, called Pos, is an array of integers in the range 1 to m, specifying the lexicographic order of the m suffixes of string T.

Pos[i] lexically less than Pos[i+1]

mississippi

pos 11,8,5,2,1,10,9,7,4,6,3

Suffix tree to suffix array

- In O(m) time
- Lexical depth-first search

Pattern searching using suffix arrays

Observation: If p occurs in T then all the locations of those occurrences will be grouped consecutively in Pos.

P=issi

T=mississipi

Pattern searching using suffix arrays

Basic idea: Binary search

O(nlogm) (worst)

O(n+logm) (expected)

A simple accelerant

L and R are left and right boundaries of the "current search interval".

Query will be made at M=(L+R)/2 of Pos.

l: the length of the longest prefix of Pos(L) that match a prefix of P

r: the length of the longest prefix of Pos(R) that match a prefix of P

lmr=min{l,r}

Compare P and Pos(M) starting from position lmr+1 of the two string.

O(nlogm)

A super accelerant

- Lcp (i,j): length of the longest prefix of Pos(i) and Pos(j)

- Use Lcp(L,M), Lcp(M,R)
  - Suppose l>r,
  - If Lcp(L,M) >l, L←M, and l, r unchanged
  - If Lcp(L,M) <l, R←M, r=Lcp(L,M)
  - If Lcp(L,M)=l, comparison of P and Pos(M) starting at l+1.

- O(n+logm)

To obtain Lcp (i,j)

- Lcp (i,i+1) for i=1 to m-1
  - Lexical depth first search
- For any i<j, Lcp(i,j) is the smallest value of Lcp(k,k+1), where, k=i to j-1
  - Lexical depth first search in a complete binary tree