CS481/CS583: Bioinformatics Algorithms

Can Alkan
EA509
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/

Multiple Pattern Matching → Multiple patterns and/or multiple texts

Keyword Tree → if there are more than

COMBINATORIAL PATTERN MATCHING

Genomic Repeats

- Example of repeats:
  - AT<u>GGTC</u>TA<u>GGTC</u>CTAGT<u>GGTC</u>
- Motivation to find them:
  - Genomic rearrangements are often associated with repeats
  - Trace evolutionary secrets
  - Many tumors are characterized by an explosion of repeats

Genomic Repeats

- The problem is often more difficult:
  - AT<u>GGTC</u>TA<u>GGAC</u>CTAGT<u>GTTC</u>
- Motivation to find them:
  - Genomic rearrangements are often associated with repeats
  - Trace evolutionary secrets
  - Many tumors are characterized by an explosion of repeats

ℓ-mer Repeats

- Long repeats are difficult to find
- Short repeats are easy to find (e.g., hashing)

- Simple approach to finding long repeats:

  - Find exact repeats of short ℓ-mers (ℓ is usually 10 to 13)

  - Use ℓ-mer repeats to potentially extend into longer, *maximal* repeats

ℓ-mer Repeats (cont'd)

- There are typically many locations where an ℓ-mer is repeated:

GCTTACAGATTCAGTCTTACAGATGGT

- The 4-mer TTAC starts at locations 3 and 17

Extending *l*-mer Repeats

GCTTACAGATTCAGTCTTACAGATGGT

- Extend these 4-mer matches:

GC<u>TTACAGAT</u>TCAGTC<u>TTACAGAT</u>GGT

- Maximal repeat: TTACAGAT

Maximal Repeats

- To find maximal repeats in this way, we need ALL start locations of all ℓ-mers in the genome
- **Hashing** lets us find repeats quickly in this manner

Hashing DNA sequences

- Each ℓ-mer can be translated into a binary string (**A**, **T**, **C**, **G** can be represented as **00**, **01**, **10**, **11**)
- After assigning a unique integer per ℓ-mer it is easy to get all start locations of each ℓ-mer in a genome

ACG encoding = 001011 i = 11

CGC encoding = 101110 = 46

123456

Genome = ACGCGACG..

h[11] = 1,7

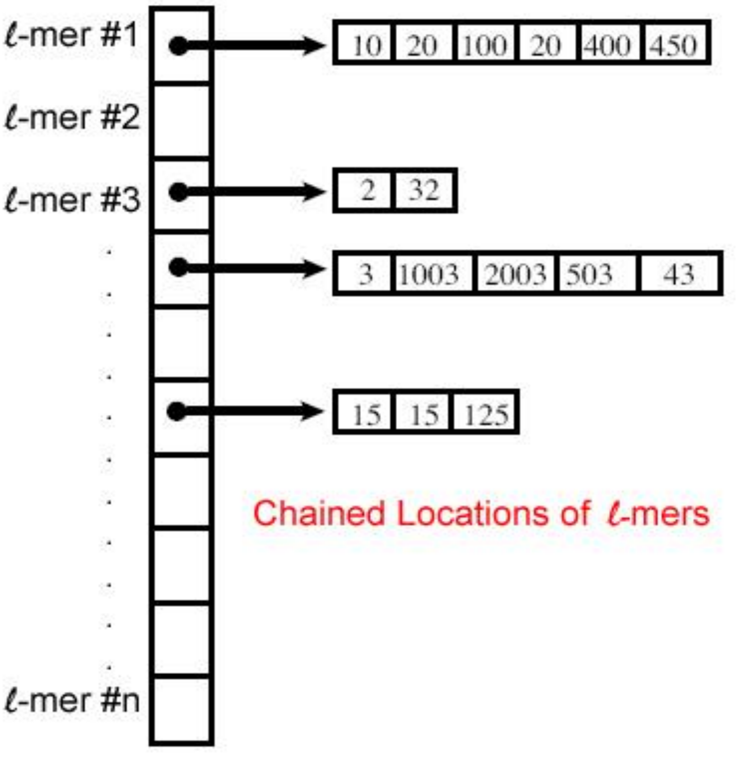
h[46] = 2

Hashing: Maximal Repeats

- To find repeats in a genome:
  - For all l-mers in the genome, note the start position and the sequence
  - Generate a hash table index for each unique l-mer sequence
  - In each index of the hash table, store all genome start locations of the l-mer which generated that index
  - Extend l-mer repeats to maximal repeats

Hashing: Collisions

- Dealing with collisions:
  - “Chain” all start locations of ℓ-mers (linked list)



Hashing: Summary

- When finding genomic repeats from l-mers:
  - Generate a hash table index for each l-mer sequence
  - In each index, store all genome start locations of the l-mer which generated that index
  - Extend l-mer repeats to maximal repeats

Pattern Matching

- What if, instead of finding repeats in a genome, we want to find all sequences in a database that contain a given pattern?

- This leads us to a different problem, the ***Pattern Matching Problem***

Pattern Matching Problem

- Goal: *Find all occurrences of a pattern in a text*
- Input: Pattern $\boldsymbol{p} = p_1 \ldots p_n$ and text $\boldsymbol{t} = t_1 \ldots t_m$
- Output: All positions $1 \leq i \leq (m - n + 1)$ such that the n-letter substring of $\boldsymbol{t}$ starting at i matches $\boldsymbol{p}$
- **Motivation**: Searching database for a known pattern

Exact Pattern Matching: A Brute-Force Algorithm

<u>PatternMatching(**p**,**t**)</u>

$m \leftarrow$ length of pattern **p**

$n \leftarrow$ length of text **t**

for $i \leftarrow 1$ to $(n - m + 1)$

    if $\boldsymbol{t}_i \ldots \boldsymbol{t}_{i+m-1} = \mathbf{p}$

        output i

Exact Pattern Matching: An Example

- *PatternMatching* algorithm for:
  - Pattern GCAT
  - Text CGCATC

```
GCAT
CGCATC

 GCAT
CGCATC

 GCAT
CGCATC

 GCAT
CGCATC

 GCAT
CGCATC
```

Exact Pattern Matching: Running Time

- *PatternMatching* runtime: O(*nm*)
- Better solution: suffix trees
  - Can solve problem in O(*n*) time
  - Conceptually related to keyword trees (=trie)
    - Multiple T, single P; or
    - Single T, multiple P

Multiple Pattern Matching Problem

- Goal: *Given a set of patterns and a text, find all occurrences of any of patterns in text*

- Input: k patterns $\mathbf{p}^1,\ldots,\mathbf{p}^k$, and text $\mathbf{t} = t_1\ldots t_m$

- Output: Positions $1 \leq i \leq m$ where substring of $\mathbf{t}$ starting at i matches $\mathbf{p}_j$ for $1 \leq j \leq k$

- **Motivation**: Searching database for known multiple patterns

Multiple Pattern Matching: Straightforward Approach

- Can solve as *k* “Pattern Matching Problems”
  - Runtime:

    O(kmn)

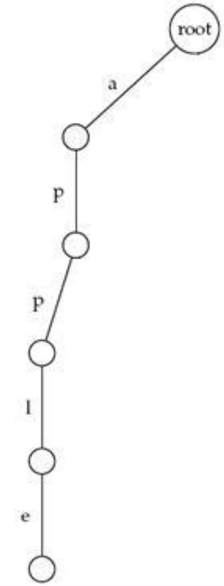
    using the *PatternMatching* algorithm *k* times
  - *m* - length of the text
  - *n* - average length of the pattern

Multiple Pattern Matching: Keyword Tree Approach

- Or, we could use keyword trees:
  - Build keyword tree in $O(N)$ time; N is total length of all patterns
  - With naive threading: $O(N + nm)$
  - Aho-Corasick algorithm: $O(N + m)$

Keyword Trees: Example

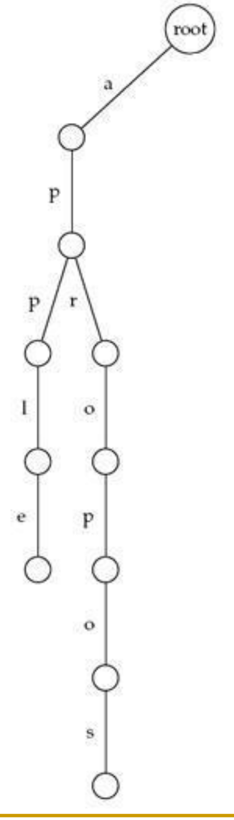
- ***Keyword tree***:
  - Apple



Also known as “trie”

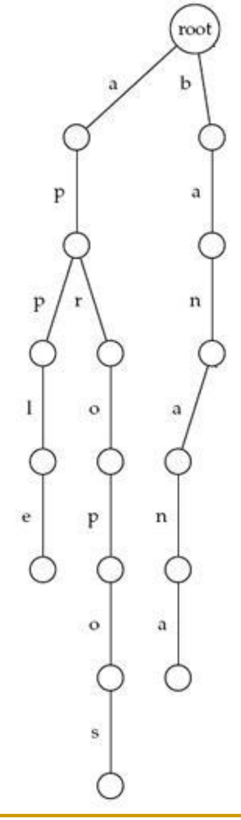
Keyword Trees: Example (cont'd)

- ***Keyword tree***:
  - Apple
  - Apropos



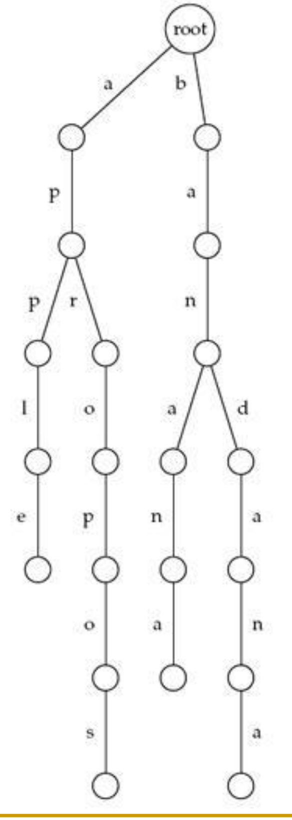
Keyword Trees: Example (cont'd)

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana



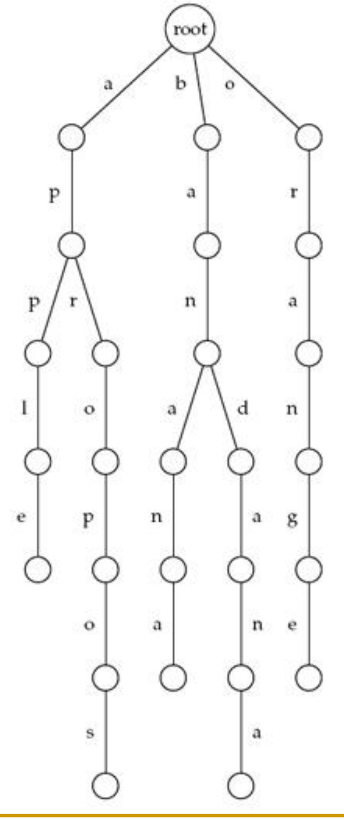
Keyword Trees: Example (cont'd)

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana
  - Bandana



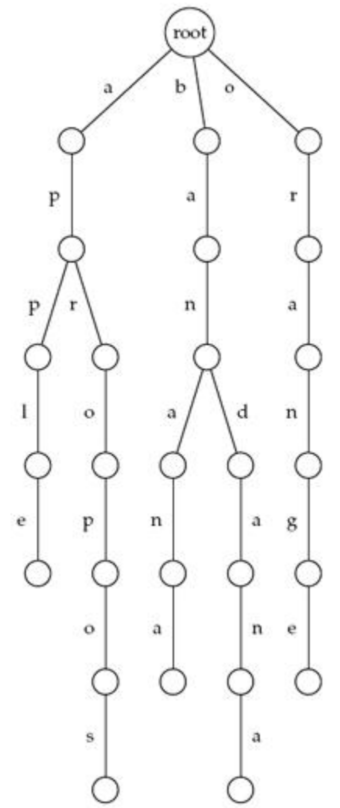
Keyword Trees: Example (cont'd)

- ***Keyword tree***:
  - Apple
  - Apropos
  - Banana
  - Bandana
  - Orange



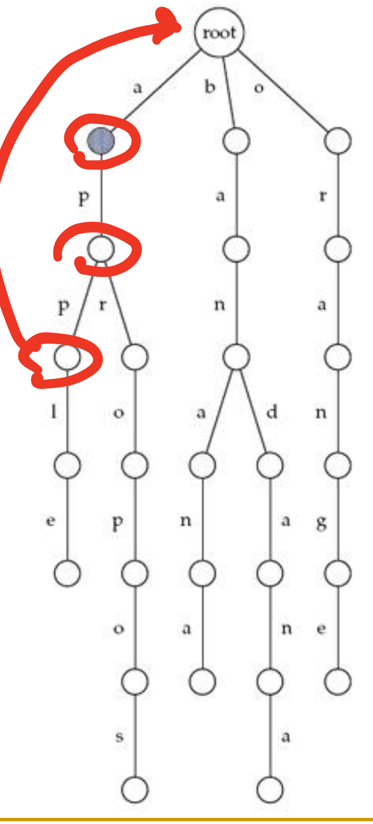
Keyword Trees: Properties

- Stores a set of keywords in a rooted labeled tree
- Each edge labeled with a letter from an alphabet
- Any two edges coming out of the same vertex have distinct labels
- Every keyword stored can be spelled on a path from root to some leaf



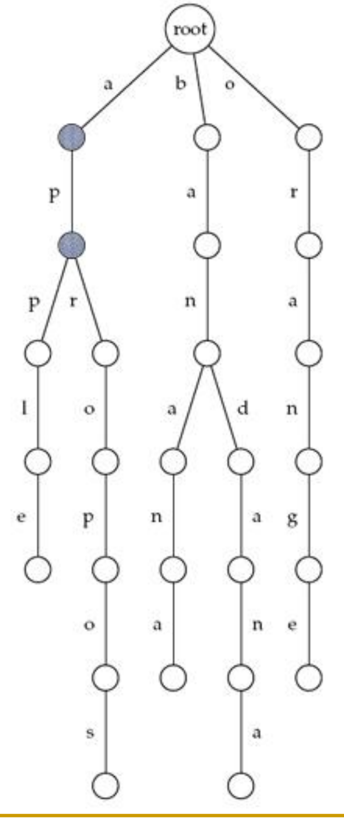
Keyword Trees: Threading (cont'd)

- Thread "appeal"
  - <u>a</u>ppeal



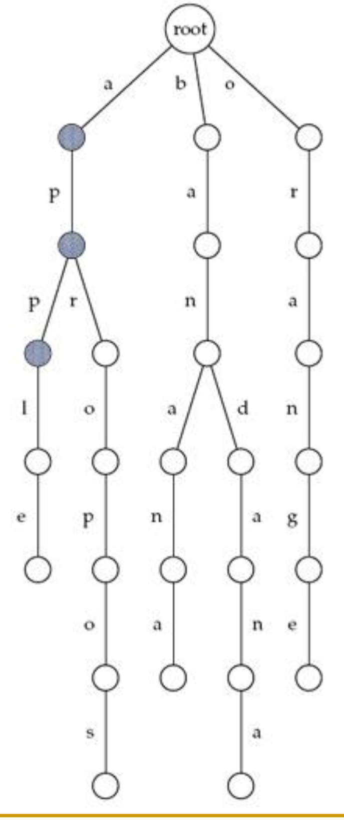
Keyword Trees: Threading (cont'd)

- Thread "appeal"
  - <u>ap</u>peal



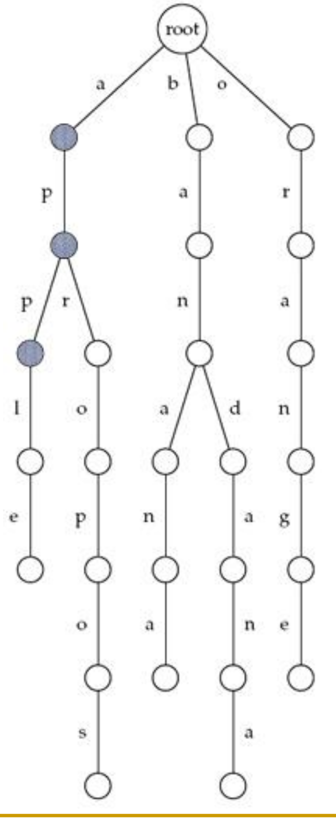
Keyword Trees: Threading (cont'd)

- Thread "appeal"
  - appeal



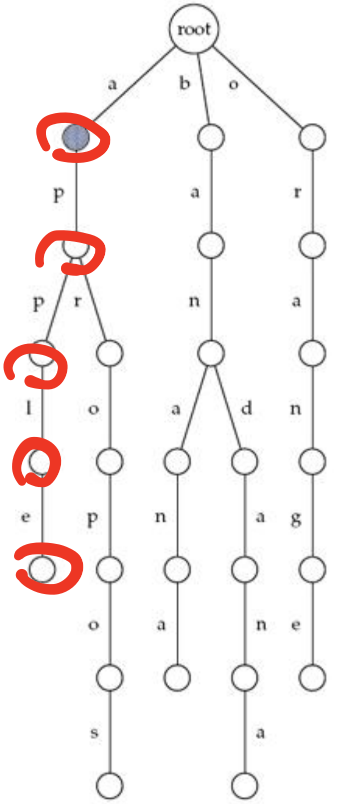
Keyword Trees: Threading (cont'd)

- Thread "appeal"
  - appeal



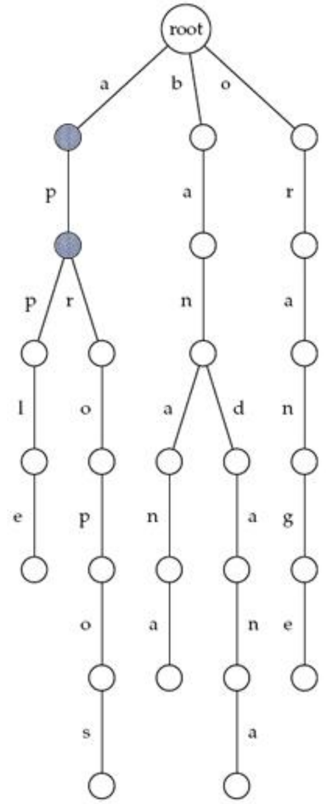
Keyword Trees: Threading (cont'd)

- Thread "apple"
  - <u>a</u>pple



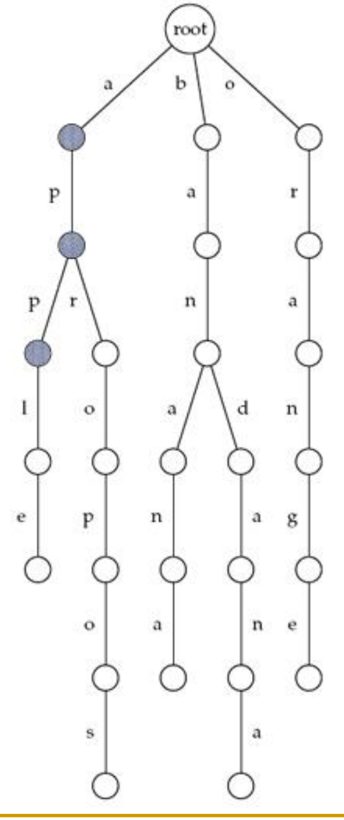
Keyword Trees: Threading (cont'd)

- Thread "apple"
  - <u>ap</u>ple



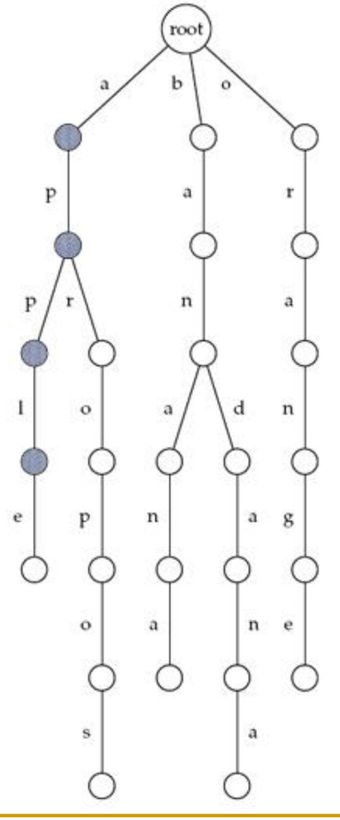
Keyword Trees: Threading (cont'd)

- Thread "apple"
  - <u>app</u>le



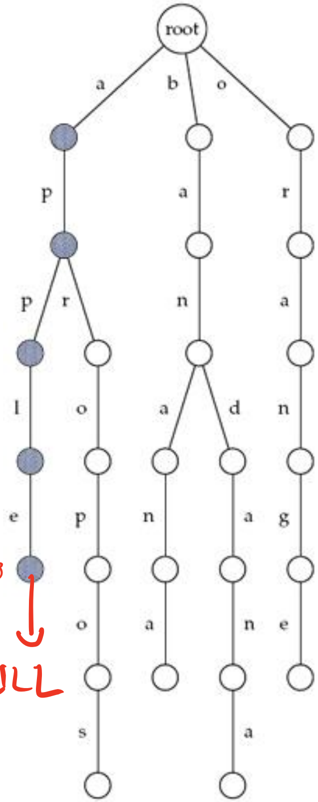
Keyword Trees: Threading (cont'd)

- Thread "apple"
  - <u>appl</u>e



Keyword Trees: Threading (cont'd)

- Thread "apple"
  - apple



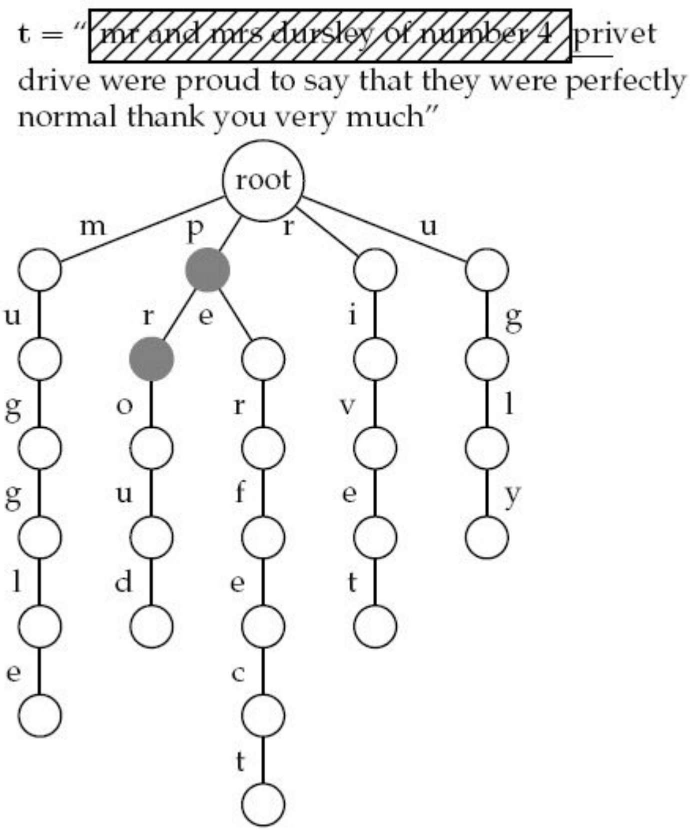
pattern found

leaf

NULL

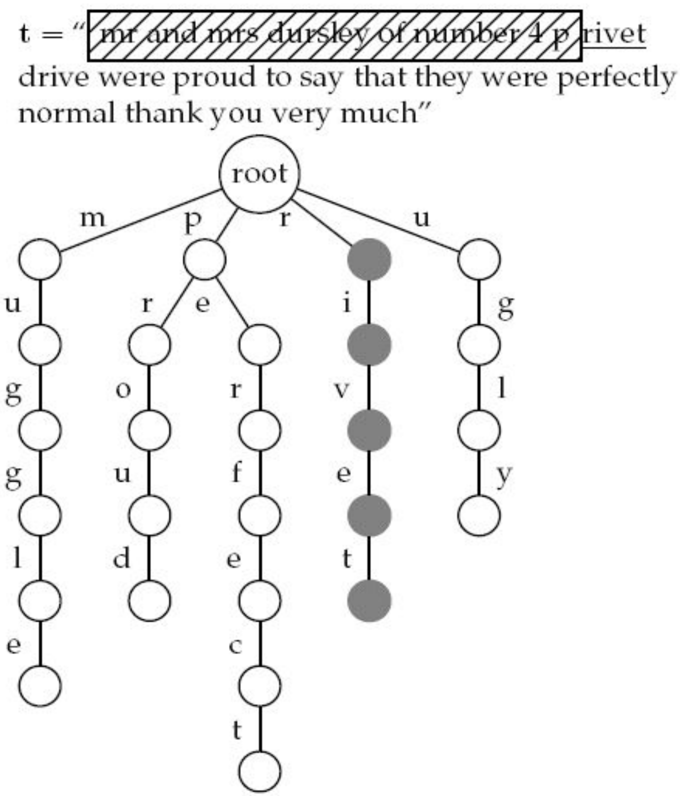
Keyword Trees: Threading

- To match patterns in a text using a keyword tree:
  - Build keyword tree of patterns
  - “Thread” the text through the keyword tree



Keyword Trees: Threading (cont'd)

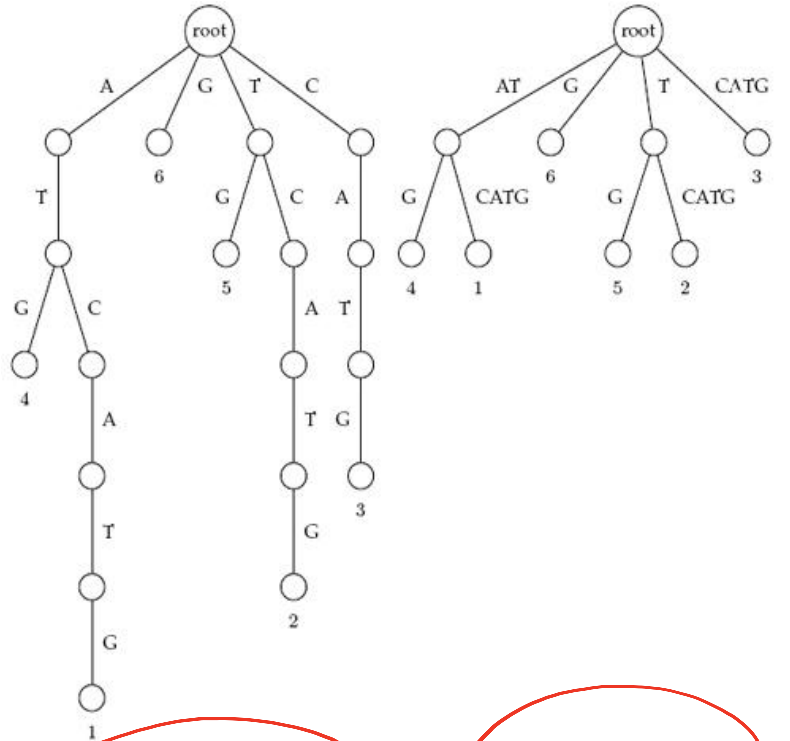
- Threading is "complete" when we reach a leaf in the keyword tree
- When threading is "complete," we've found a pattern in the text



Problem: High memory requirement when N is large

Suffix Trees=Collapsed Keyword Trees

- Similar to keyword trees, except edges that form paths are collapsed
  - Each edge is labeled with a *substring* of a text
  - All internal edges have at least two outgoing edges
  - Leaves labeled by the index of the pattern.



(a) Keyword tree

(b) Suffix tree

Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

ATCATG
TCATG
CATG
ATG
TG
G

Keyword Tree

Suffix Tree

Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

ATCATG
TCATG
CATG
ATG
TG
G

Keyword Tree

Suffix Tree

How much time does it take?

Suffix Tree of a Text

- Suffix trees of a text is constructed for all its suffixes

ATCATG
TCATG
CATG
ATG
TG
G

quadratic

Keyword Tree

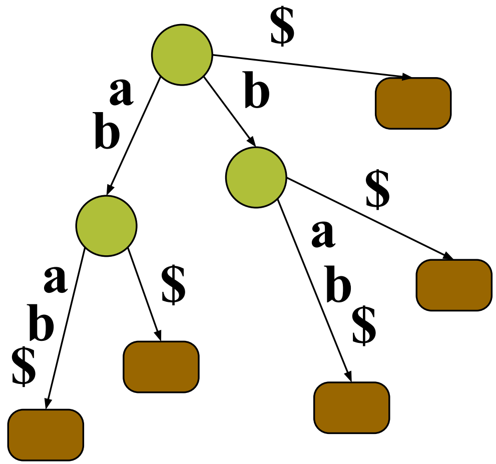
Suffix Tree

Time is linear in the total size of all suffixes, i.e., it is quadratic in the length of the text

Suffix tree (Example)

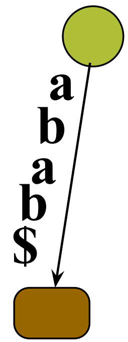
Let s=abab, a suffix tree of s is a compressed trie of all suffixes of s=abab$

{
$
b$
ab$
bab$
abab$
}

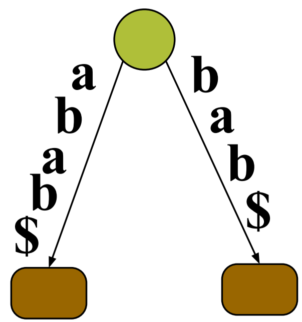


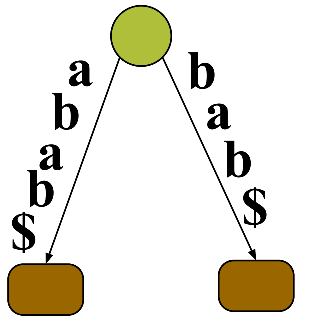
Trivial algorithm to build a Suffix tree

Put the largest suffix in

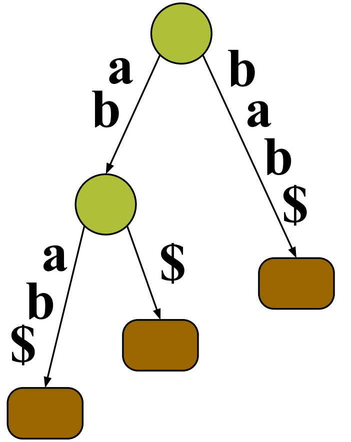


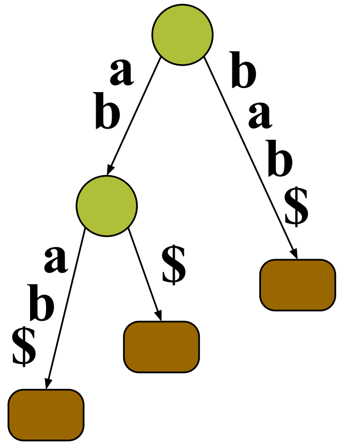
Put the suffix bab$ in



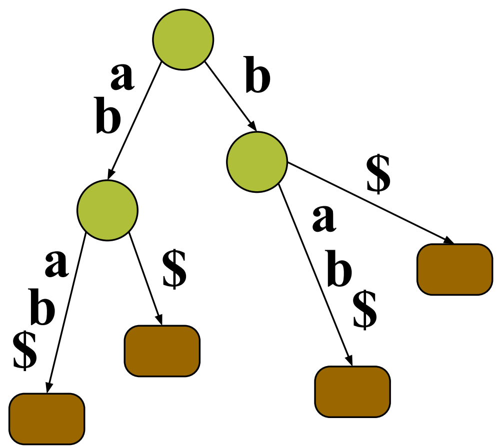


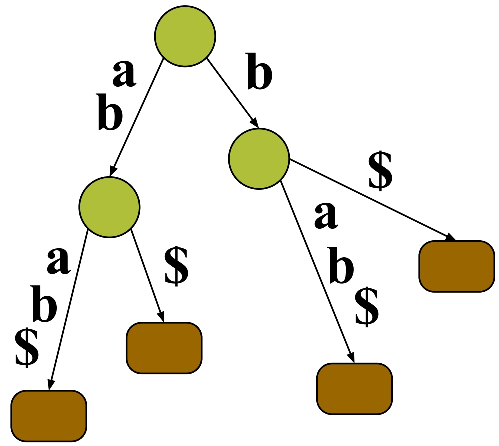
Put the suffix ab$ in



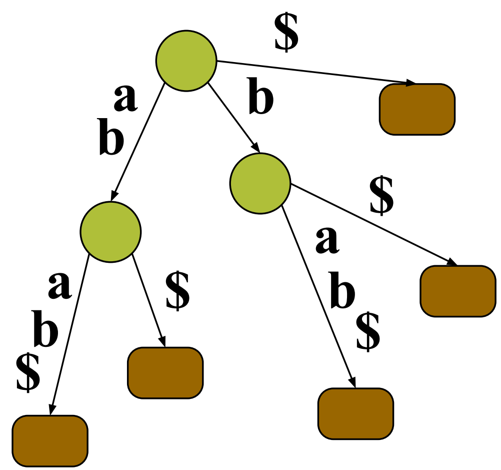


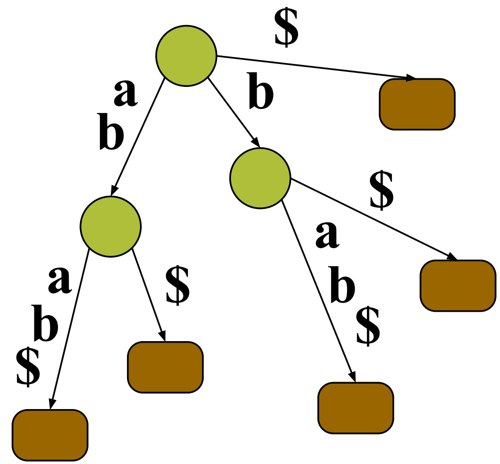
Put the suffix **b$** in





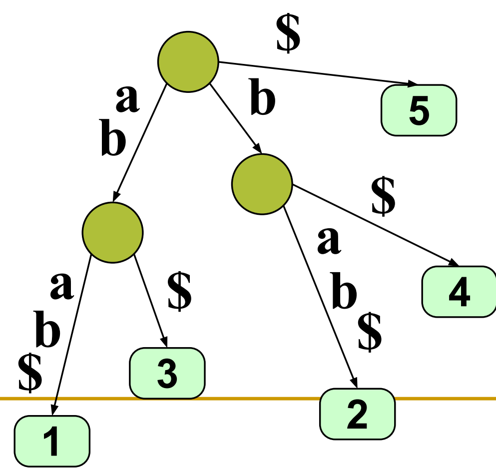
Put the suffix $ in





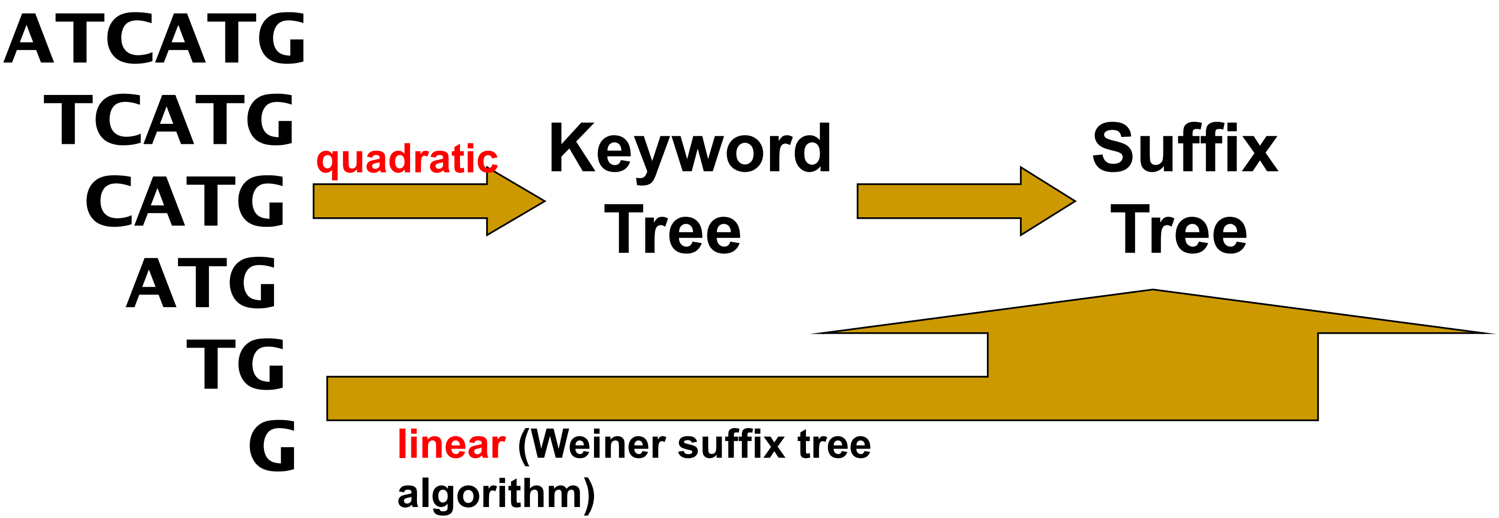
We will also label each leaf with the starting point of the corres. suffix.

Trivial algorithm: $O(n^2)$ time



Suffix Trees: Advantages

- Suffix trees of a text is constructed for all its suffixes
- Suffix trees build faster than keyword trees



Use of Suffix Trees

- Suffix trees hold all suffixes of a text
  - i.e., ATCGC: ATCGC, TCGC, CGC, GC, C
  - Builds in O(*m*) time for text of length *m*
- To find any pattern of length ***n*** in a text:
  - Build suffix tree for text
  - Thread the pattern through the suffix tree
  - Can find pattern in text in O(*n*) time!
- O(*n* + *m*) time for “Pattern Matching Problem”
  - Build suffix tree for T and look up P

Pattern Matching with Suffix Trees

<u>SuffixTreePatternMatching(**p**,**t**)</u>

Build **suffix tree** for text **t**

Thread pattern **p** through **suffix tree**

if threading is complete

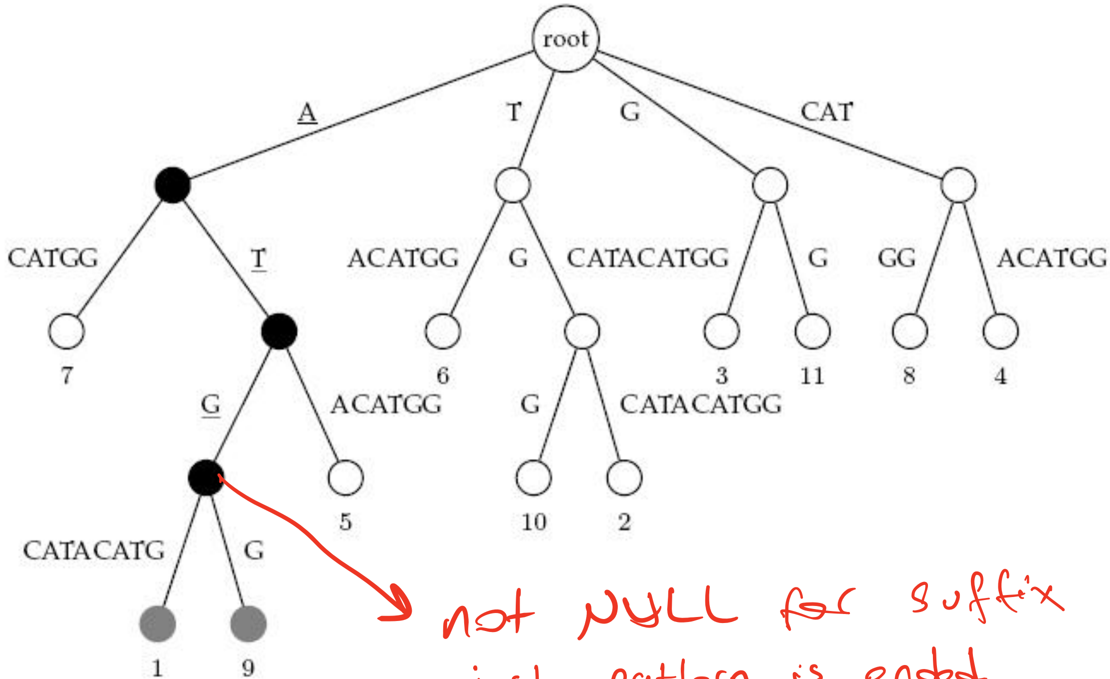
  output positions of all **p**-matching leaves in the tree

else

  output "Pattern does not appear in text"

Suffix Trees: Example

T = ATGCATACATGG   P = ATG



not NULL for suffix
just pattern is ended

Generalized suffix tree

Given a **set** of strings **S** a generalized suffix tree of **S** is a compressed trie of all suffixes of **s ∈ S**

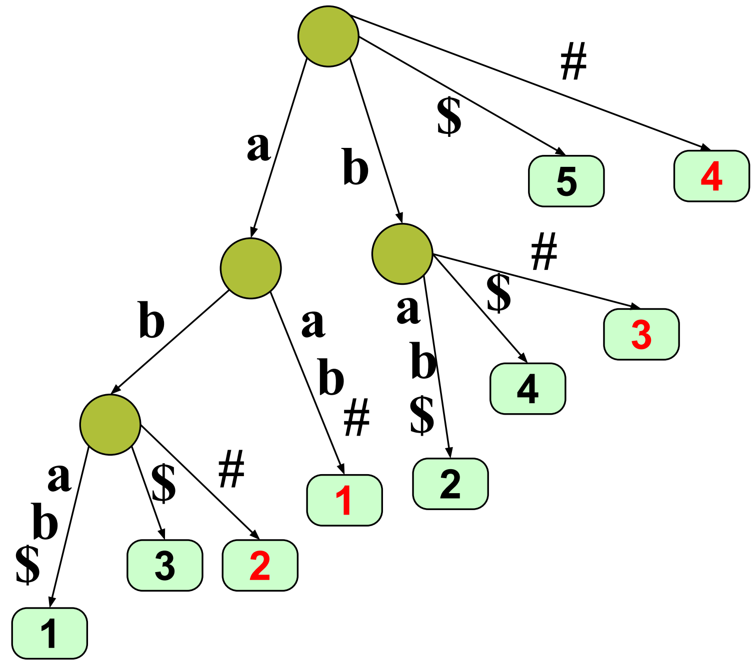
To make these suffixes prefix-free we add a special char, say **$**, at the end of **s**

To associate each suffix with a unique string in **S** add a different special char to each **s**

Generalized suffix tree (Example)

Let s_1=abab and s_2=aab here is a generalized suffix tree for s_1 and s_2

{
$ #
b$ b#
ab$ ab#
bab$ aab#
abab$
}

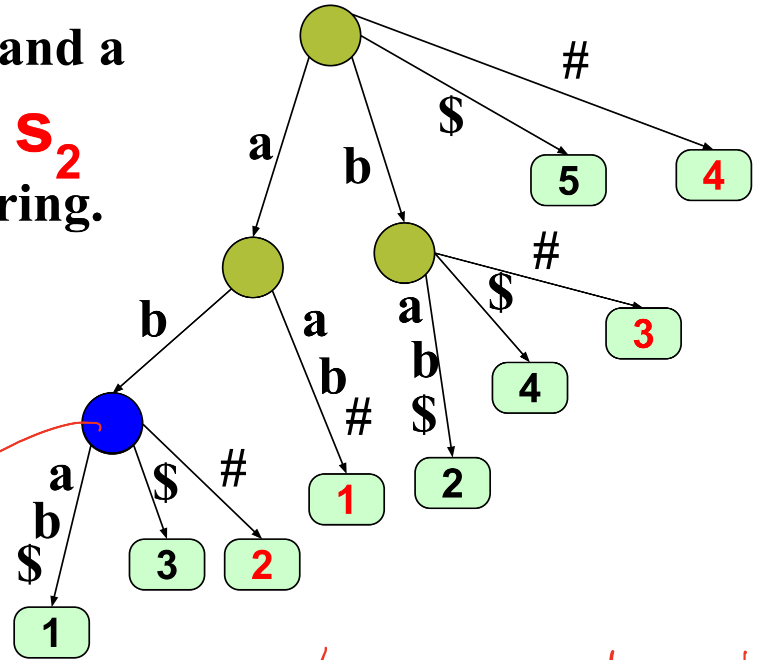


Matching a pattern against a database of strings

Longest common substring of two strings

Every node with a leaf descendant from string s_1 and a leaf descendant from string s_2 represents a common substring.

Find such node with largest "string depth"



Have $ and # nodes at the same time.

Multiple Pattern Matching: Summary

- Keyword and suffix trees are used to find patterns in a text
- ***Keyword trees***:
  - Build keyword tree of patterns, and ***thread text*** through it
- ***Suffix trees***:
  - Build suffix tree of text, and ***thread patterns*** through it

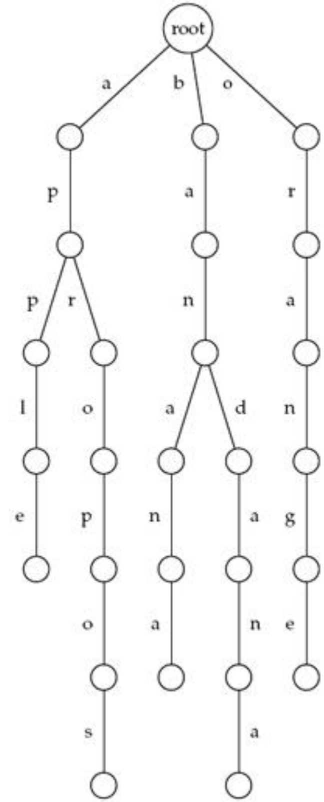
more memory consuming

Slides from Charles Yan

AHO-CORASICK

Search in keyword trees

- Naïve threading in keyword trees do not *remember* the partial matches
- P={apple, appropos}
- T=appappropos
- When threading
  - *app* is a partial match
  - But naïve threading will go back to the root and re-thread *app*
- Define *failure links*



Failure Link

v: a node in keyword tree K

L(v): the label on v, that is, the concatenation of characters on the path from the root to v.

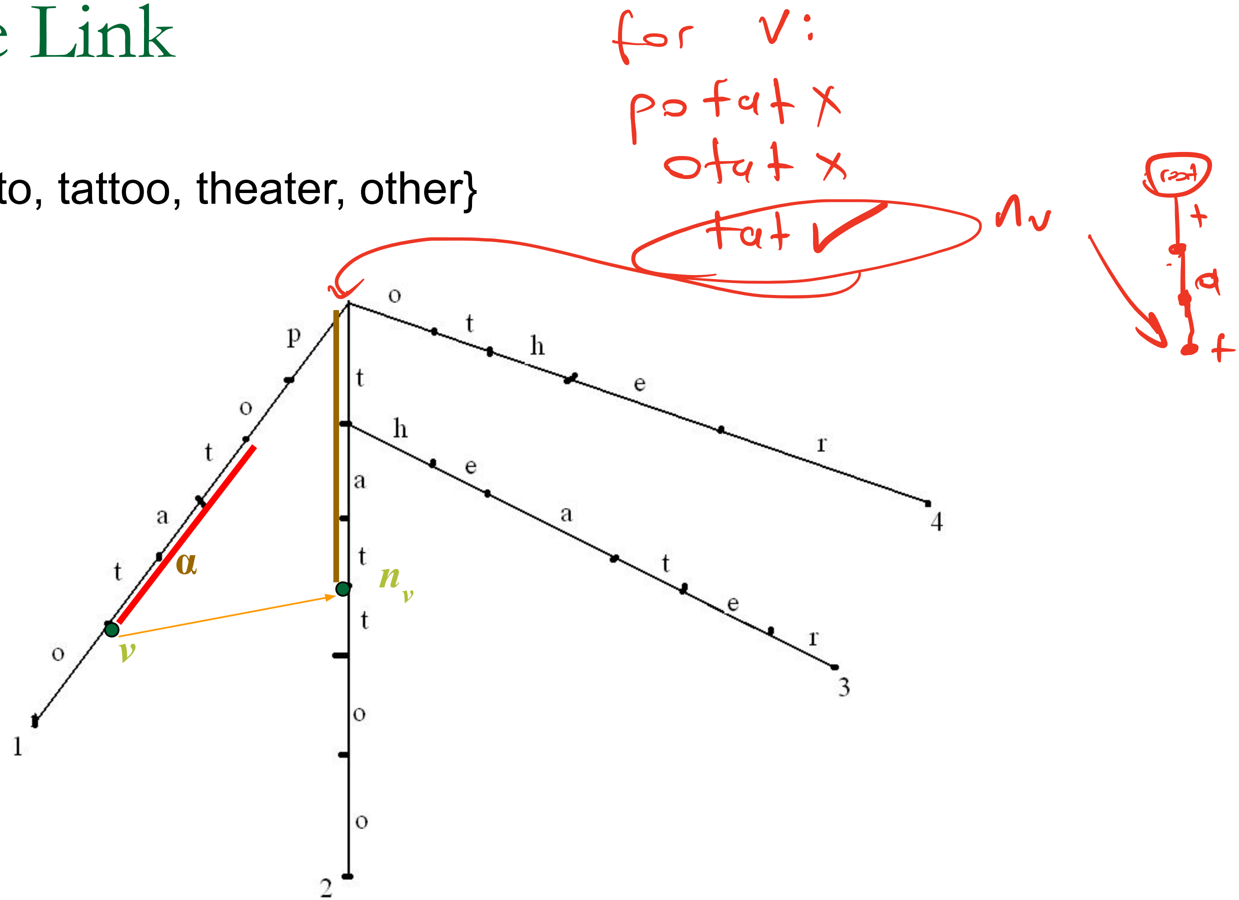
lp(v): the length of the longest proper suffix of string L(v) that is a prefix of some pattern in P. Let this substring be α.

Lemma. There is a unique node in the keyword tree that is labeled by string α. Let this node be n_v. Note that n_v can be the root.

The ordered pair (v, n_v) is called a **failure link**.

Failure Link

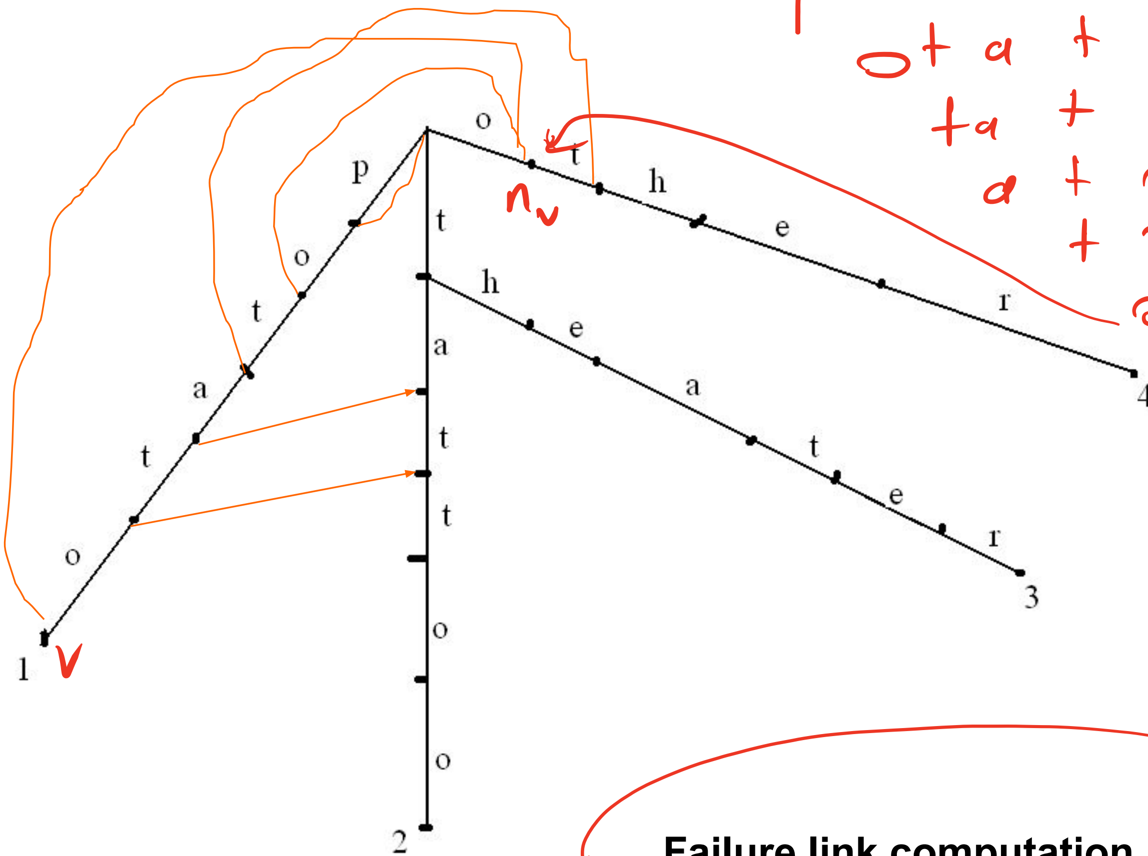
P={potato, tattoo, theater, other}



Failure Link

for potato

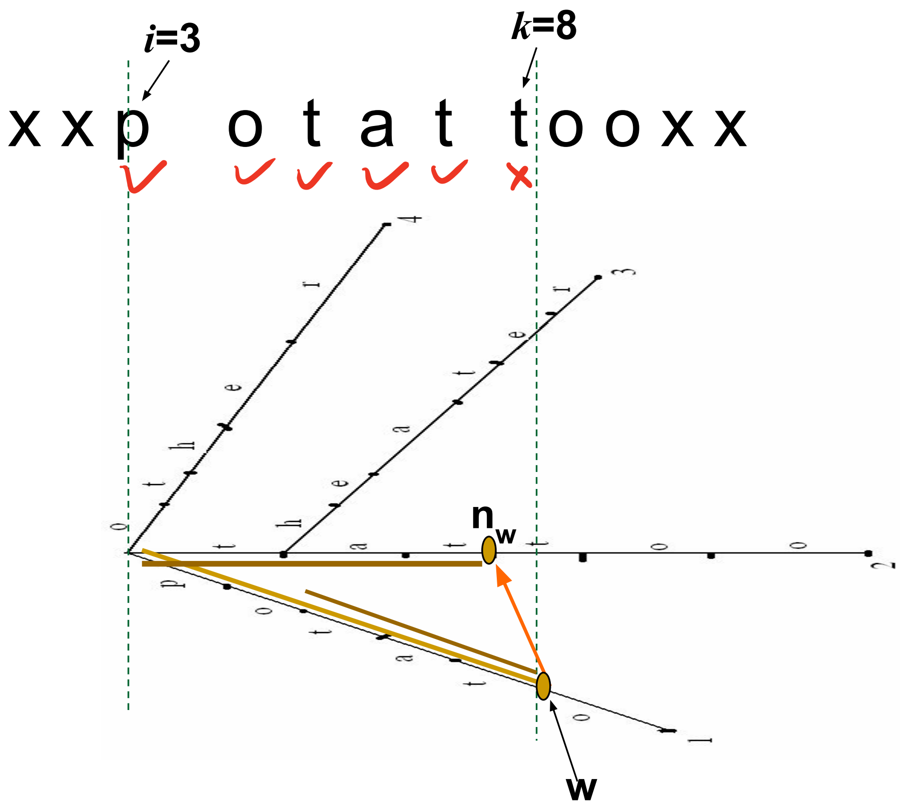
potato ×
otato ×
tato ×
ato ×
to ×
o ✓



Failure link computation is O(n)

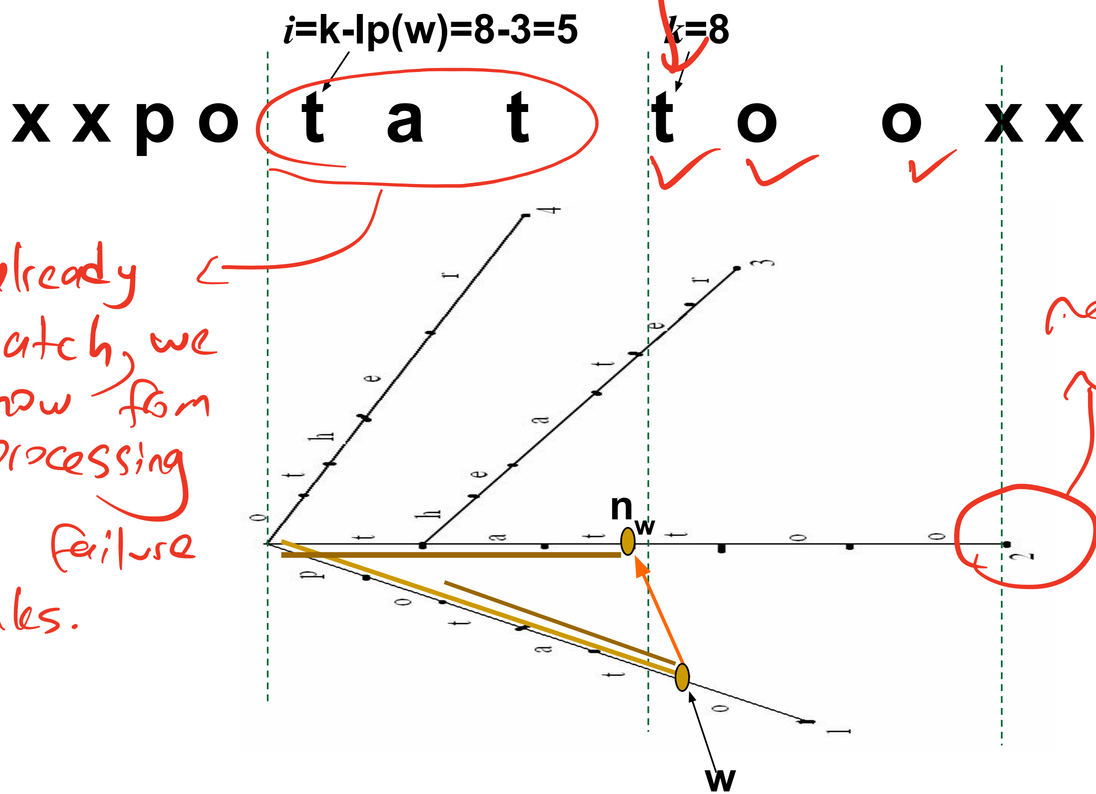
Failure Link

since seond t does not appear in potato
check n_w for it!



Failure Link

check now from here



already match, we know from preprocessing for failure links.

reached pattern no: 2

Failure Link

dynamic programming approach

How to construct failure links for a keyword tree in a linear time?

Let d be the distance of a node (v) from the root r.

When $d \leq 1$, i.e., v is the root or v is one character away from r, then $n_v = r$.

Suppose n_v has been computed for every node (v) with $d \leq k$, we are going to compute n_v for every node with $d = k+1$.

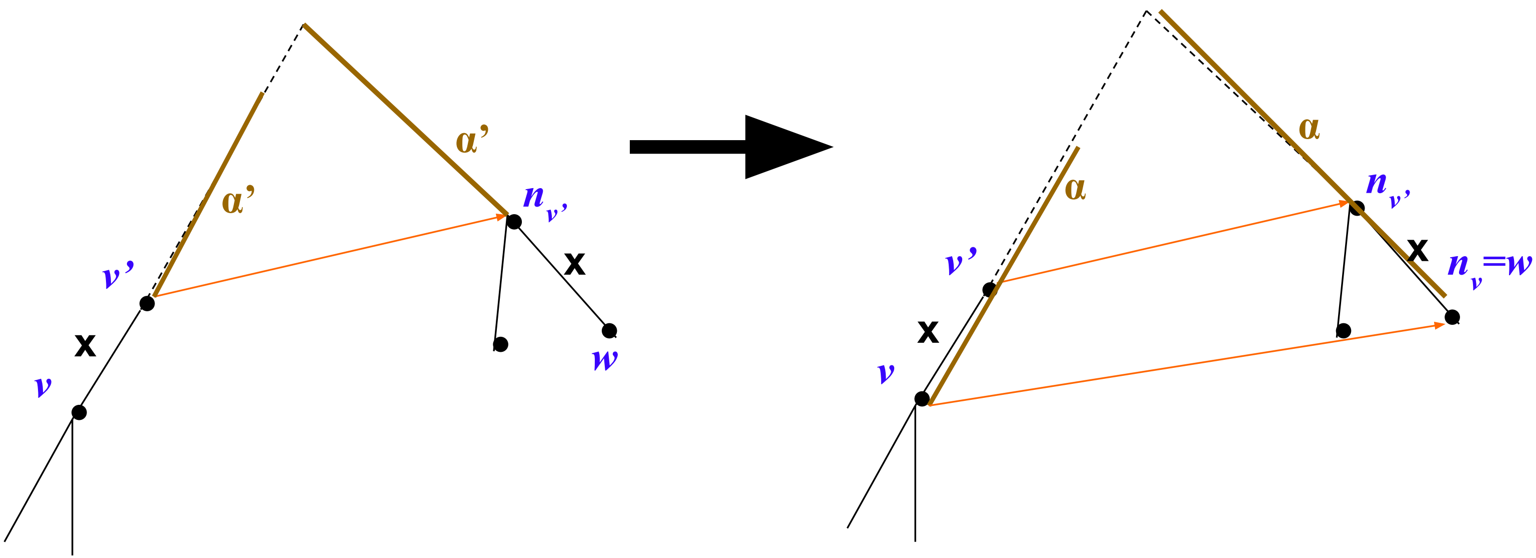
v': parent of v, then v' is k characters from r, that is d=k

thus the failure link for v' ($n_{v'}$) has been computed.

x: the character on edge (v', v)

Failure Link

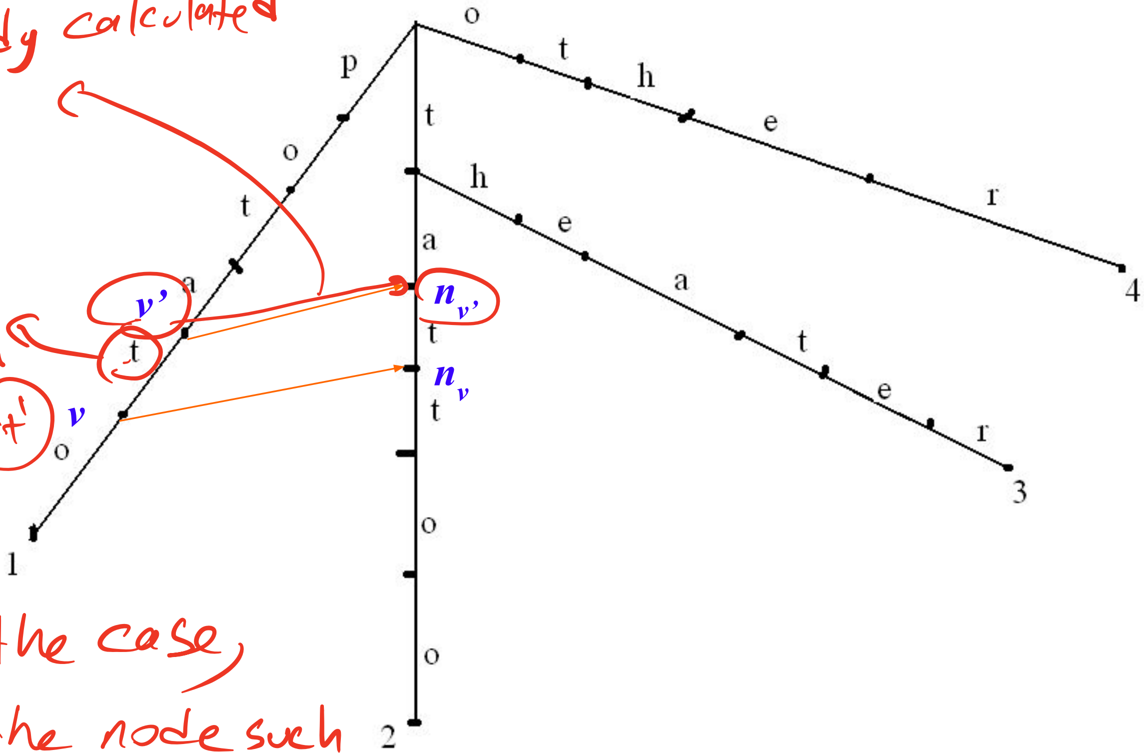
(1) If there is an edge ($n_{v'}$, w) out of $n_{v'}$ labeled with x, then n_v=w.



Failure Link

already calculated

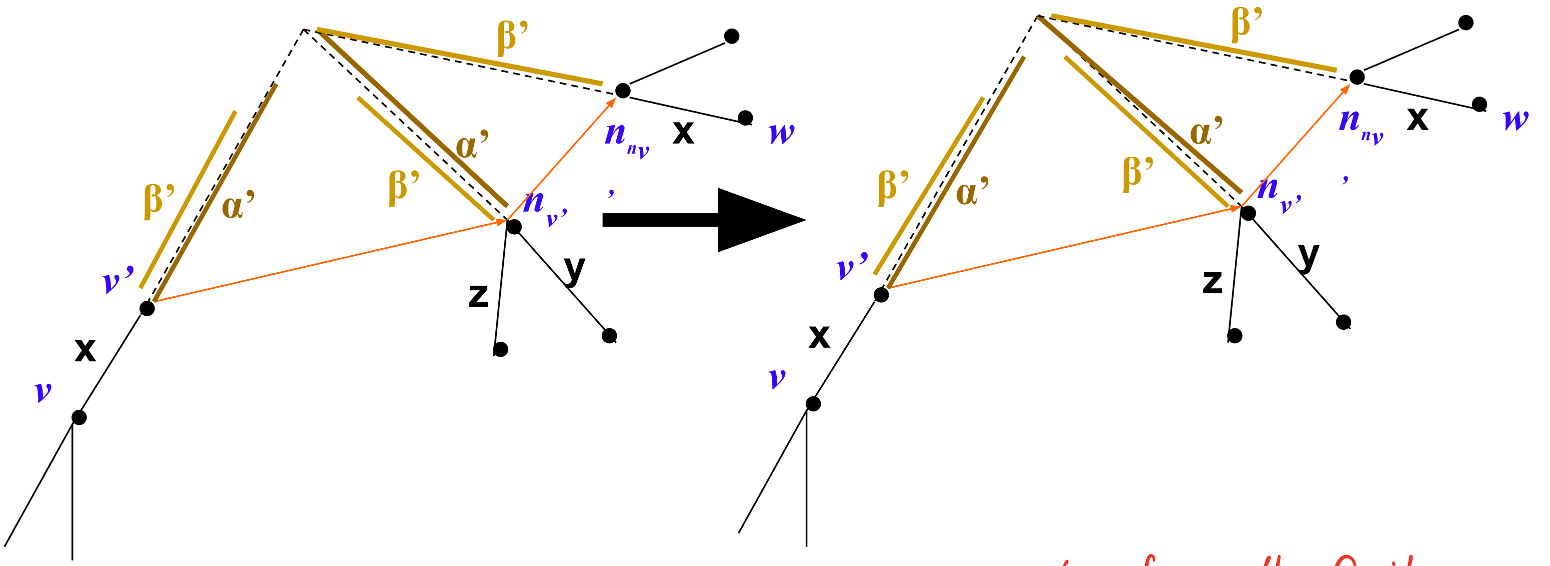
for v
check if
$n_{v'}$ has an
edge with 't'



if it is the case,
n_v is the node such
that $n_{v'} \xrightarrow{t} n_v$

Failure Link

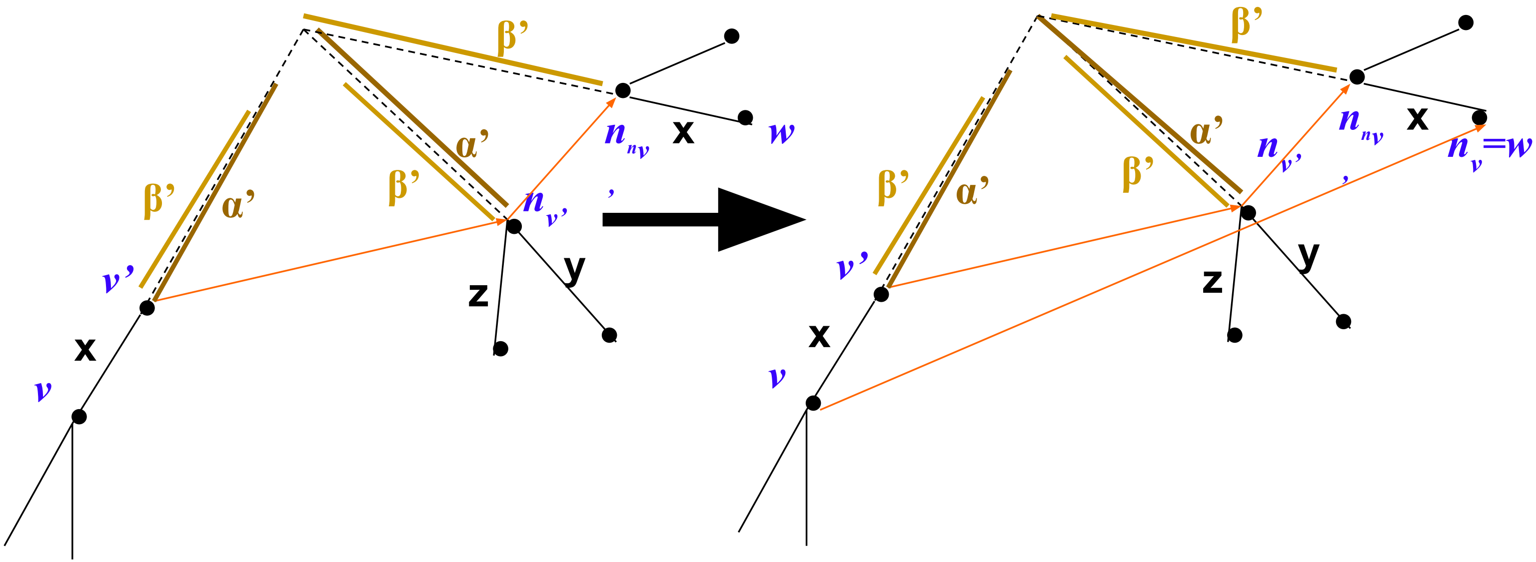
(2) If such an edge does not exist, examine $n_{n_{v'}}$ to see if there is an edge out of it labeled with x. Continue until the root.



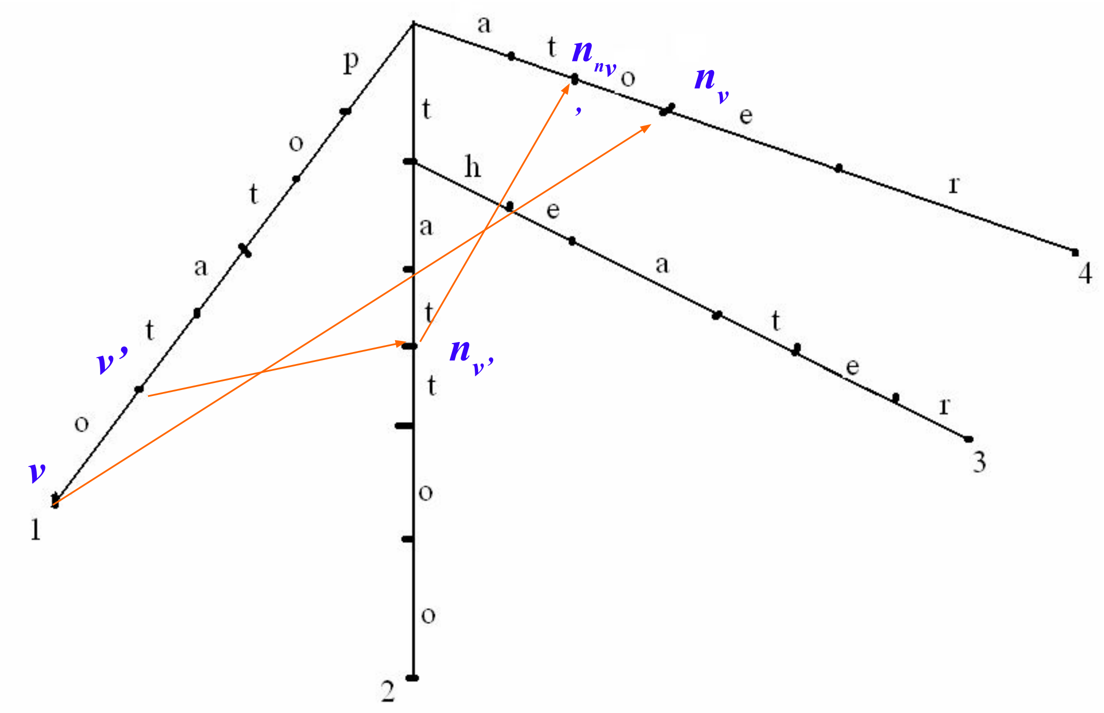
if not x, go through and check all failure links. → n_{n_v}

Failure Link

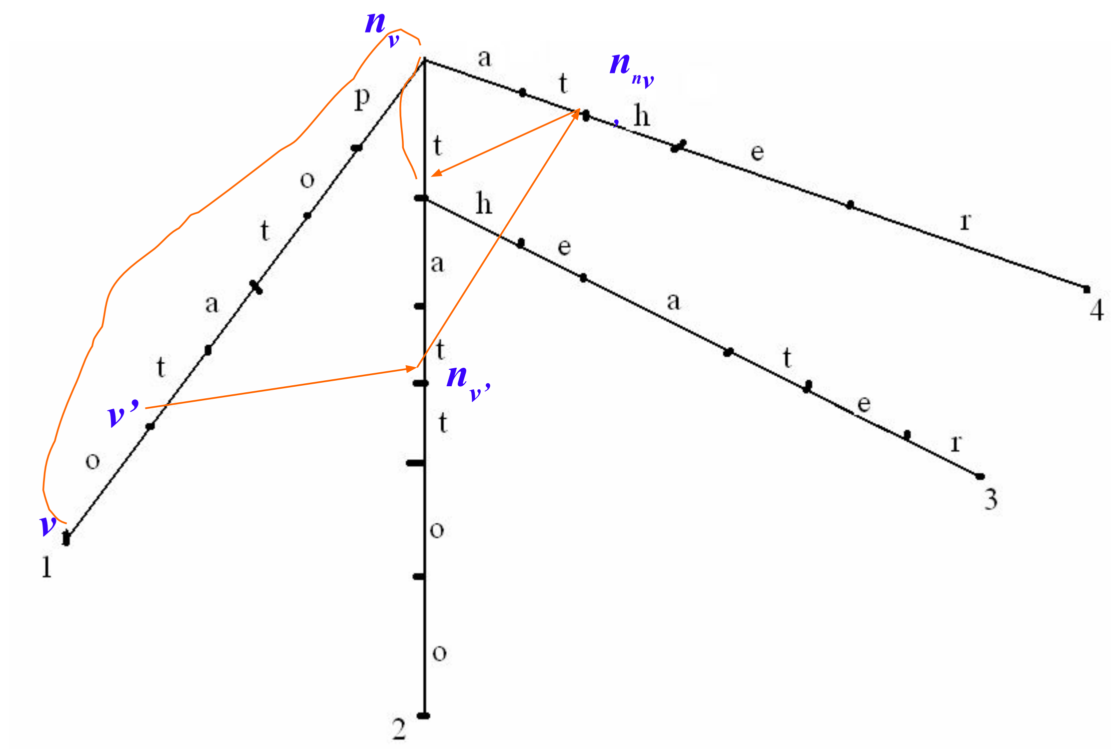
(2) If such an edge does not exist, examine $n_{n_{v'}}$ to see if there is an edge out of it labeled with x. Continue until the root.



Failure Link

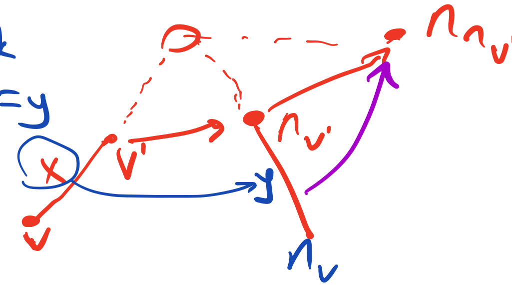


Failure Link



Failure Link

check if $x=y$



if no check n_{n_v} for other links

if yes link $v \rightarrow n_v$

Output: calculate n_v for v

Algorithm n_v

v' is the parent of v in K

x is the character on edge (v', v)

$w=n_{v'}$

while there is no edge out of w labeled with x and w≠r

$w=n_w$

If there is an edge (w, w') out of w labeled x then (w' is (w)s child)

$n_v=w'$

else

$n_v=r$

Aho-Corasick Algorithm

Input: Pattern set P and text T
Output: all occurrences in T any pattern from P
Algorithm Aho-Corasick
l=1;
c=1;
w=root of tree K
Repeat
    while there is an edge (w, w’) labeled with T(c)
        if w' is numbered by a pattern *i* then
            report that p_i occurs in T starting at l;
        w=w’; c++;
    $w=n_w$ and l=c-lp(w);
Until c>m

Slides from Tolga Can

SUFFIX ARRAYS

Suffix arrays

- Suffix arrays were introduced by Manber and Myers in 1993
- More space efficient than suffix trees
- A suffix array for a string x of length *m* is an array of size *m* that specifies the lexicographic ordering of the suffixes of x.

Suffix arrays

Example of a suffix array for acaaacatat$

0	aaacatat$	3
1	aacatat$	4
2	acaaacatat$	1
3	acatat$	5
4	atat$	7
5	at$	9
6	caaacatat$	2
7	catat$	6
8	tat$	8
9	t$	10
10	$	11

Suffix array construction

- Naive in place construction
  - Similar to insertion sort → research about it.
  - Insert all the suffixes into the array one by one making sure that the new inserted suffix is in its correct place
  - Running time complexity:
    - $O(m^2)$ where m is the length of the string
- Manber and Myers give a $O(m \log m)$ construction.

Suffix arrays

- O(*n*) space where *n* is the size of the database string
- Space efficient. However, there's an increase in query time
- Lookup query
  - Based on binary search
  - O(m log n) time; m is the size of the query
  - Can reduce time to O(m + log n) using a more efficient implementation

Searching for a pattern in Suffix Arrays

```
find(Pattern P in SuffixArray A):
    i = 0
    lo = 0, hi = length(A)
     for 0<=i<length(P):
        Binary search for x,y
        where P[i]=S[A[j]+i] for lo<=x<=j<y<=hi
        lo = x, hi = y
     return {A[lo],A[lo+1],...,A[hi-1]}
```

Search example

- Search *is* in *mississippi$*

Examine the pattern letter by letter, reducing the range of occurrence each time.

return them

First letter *i*:
occurs in indices from 0 to 3

So, pattern should be between these indices.

Second letter *s*:
occurs in indices from 2 to 3

Done.
Output: issippi$ and
ississippi$

0	11	i$
1	8	ippi$
2	5	issippi$
3	2	ississippi$
4	1	mississippi$
5	10	pi$
6	9	ppi$
7	7	sippi$
8	4	sissippi$
9	6	ssippi$
10	3	ssissippi$
11	12	$

Suffix Arrays

- They can be built very fast.
- They can answer queries very fast:
  - How many times does ATG appear?
- Disadvantages:
  - Can't do approximate matching
    - Except with some heuristics we will cover later
  - Hard to insert new stuff (need to rebuild the array) dynamically.