Text Algorithms (6EAP)

Full text indexing (gentle)

Jaak Vilo

2012 fall

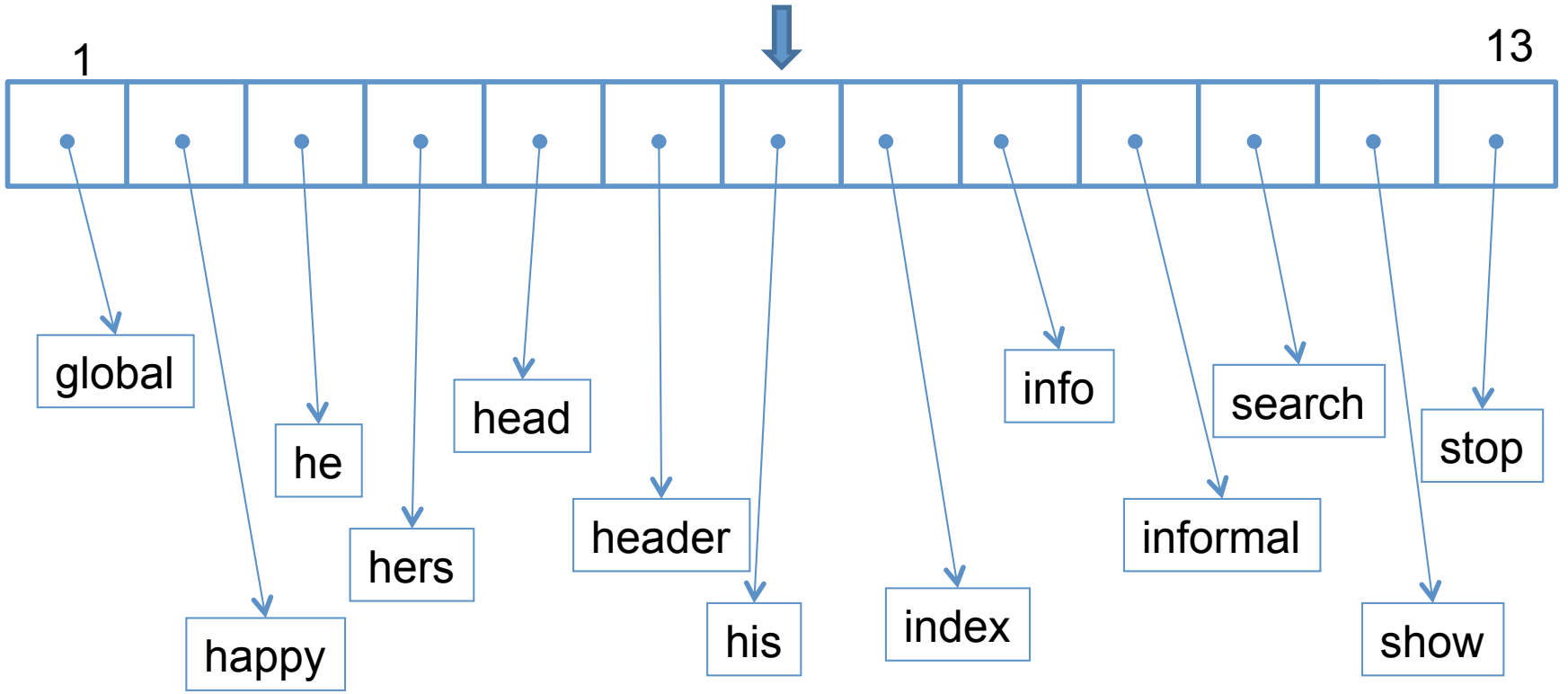
Problem

- Given P and S – find all exact or approximate occurrences of P in S
- You are allowed to preprocess S (and P, of course)
- Goal: to speed up the searches

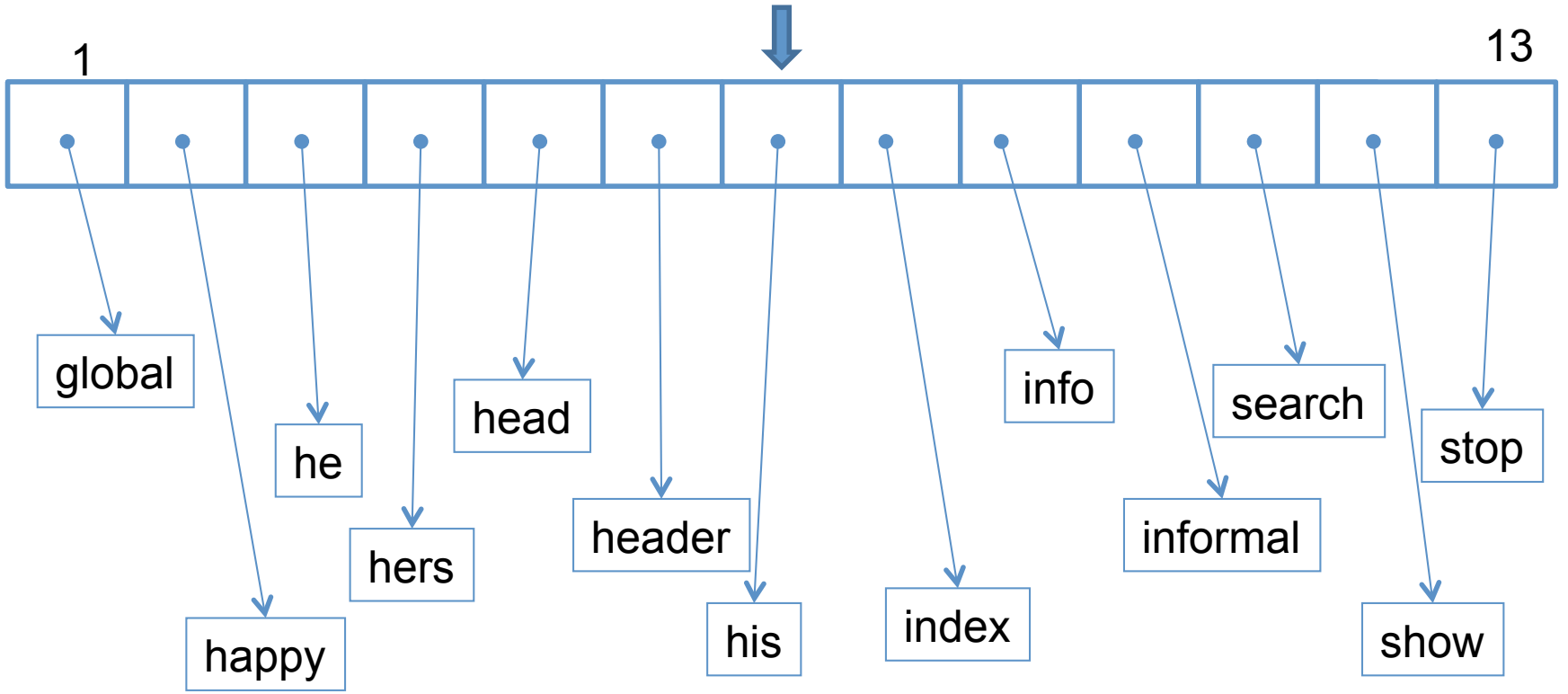
E.g. Dictionary problem

- Does P belong to a dictionary $D=\{d_1,\ldots,d_n\}$
  - Build a binary search tree of D
  - B-Tree of D
  - Hashing
  - Sorting + Binary search
- Build a keyword trie: search in **O(|P|)**
  - Assuming alphabet has up to a constant size c
  - See Aho-Corasick algorithm, Trie construction

Sorted array and binary search

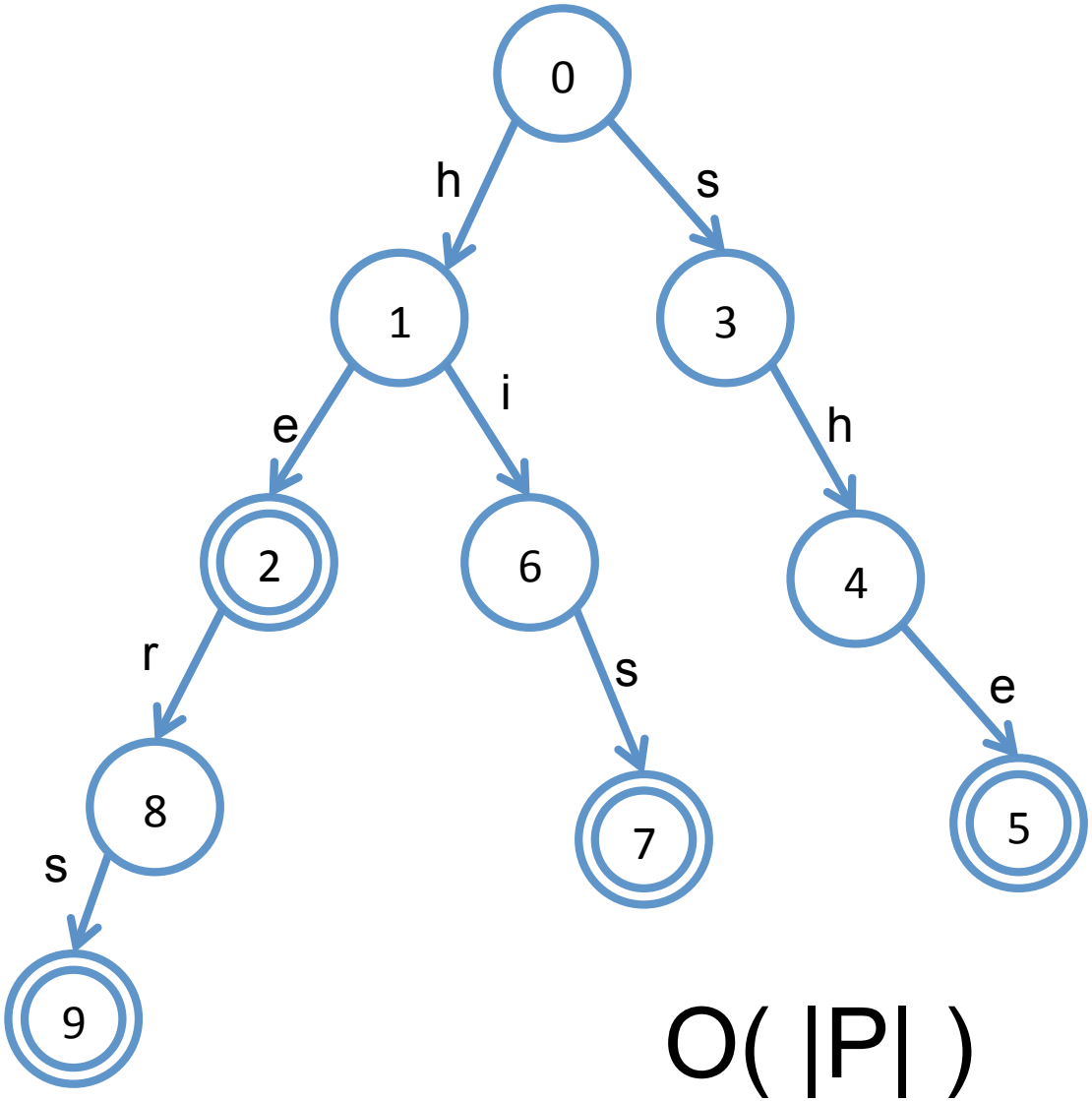


Sorted array and binary search



O(|P| log n)

Trie for D={ he, hers, his, she}



O(|P|)

S != set of words

- S of length n
- How to index?
- **Index from every position of a text**
- Prefix of every possible suffix is important

ttttttttttttgagacggagtctcgctctg
tcgcccaggctggagtgcagtggcggg
atctcggctcactgcaagctccgcctcc
cgggttcacgccattctcctgcctcagcc
tcccaagtagctgggactacaggcgcc
cgccactacgcccggctaatttttgtattt
ttagtagagacggggtttcaccgttttagc
cgggatggtctcgatctcctgacctcgtg
atccgcccgcctcggcctcccaaagtgc
tgggattacaggcgt

E. Ukkonen: http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

Suffix tree

- **Definition:** A compact representation of a trie corresponding to the suffixes of a given string where all nodes with one child are merged with their parents.
- **Definition (suffix tree)**. A suffix tree T for a string S (with n = |S|) is a rooted, labeled tree with a leaf for each non-empty suffix of S. Furthermore, a suffix tree satisfies the following properties:
- Each internal node, other than the root, has at least two children;
- Each edge leaving a particular node is labeled with a non-empty substring of S of which the first symbol is unique among all first symbols of the edge labels of the edges leaving this particular node;
- For any leaf in the tree, the concatenation of the edge labels on the path from the root to this leaf exactly spells out a non-empty suffix of s.
- **Dan Gusfield:** Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Hardcover - 534 pages 1st edition (January 15, 1997). Cambridge Univ Pr (Short); ISBN: 0521585198.

Literature on suffix trees

- http://en.wikipedia.org/wiki/Suffix_tree
- **Dan Gusfield:** Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Hardcover - 534 pages 1st edition (January 15, 1997). Cambridge Univ Pr (Short); ISBN: 0521585198. (pages: 89--208)
- E. Ukkonen. On-line construction of suffix trees. Algorithmica, 14:249-60, 1995. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.751
- Ching-Fung Cheung, Jeffrey Xu Yu, Hongjun Lu. "Constructing Suffix Tree for Gigabyte Sequences with Megabyte Memory," IEEE Transactions on Knowledge and Data Engineering, vol. 17, no. 1, pp. 90-105, January, 2005. http://www2.computer.org/portal/web/csdl/doi/10.1109/TKDE.2005.3
- CPM articles archive: http://www.cs.ucr.edu/~stelo/cpm/
- Mark Nelson. Fast String Searching With Suffix Trees *Dr. Dobb's Journal*, August, 1996. http://www.dogma.net/markn/articles/suffixt/suffixt.htm

- **STXXL : Standard Template Library for Extra Large Data Sets.**
- http://stxxl.sourceforge.net/

- **ANSI C implementation of a Suffix Tree**
- http://yeda.cs.technion.ac.il/~yona/suffix_tree/

- **SeqAn** - http://www.seqan.de/

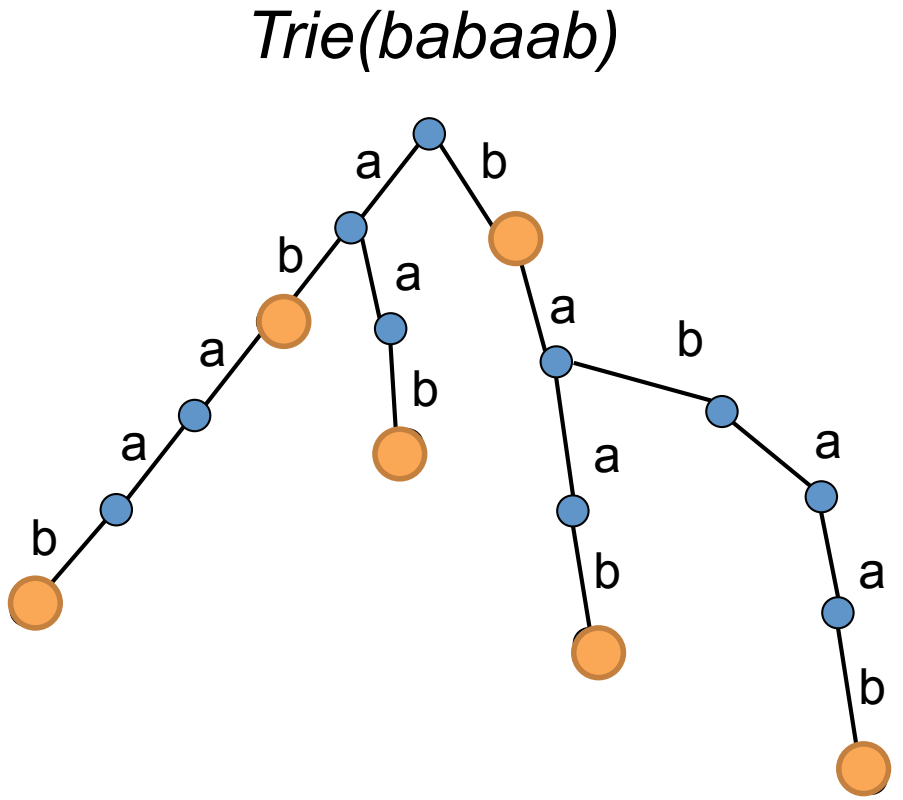
The suffix tree Tree(T) of T

- data structure suffix tree, Tree(T), is compacted trie that represents all the suffixes of string T
- linear size: |Tree(T)| = O(|T|)
- can be constructed in linear time O(|T|)
- has *myriad virtues* (A. Apostolico)
- is well-known: 366 000 Google hits
  - 2012: 665 000 (”suffix tree”)

E. Ukkonen: http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

Suffix Trie

babaab
abaab
baab
aab
ab
b



Suffix trie and suffix tree

abaab

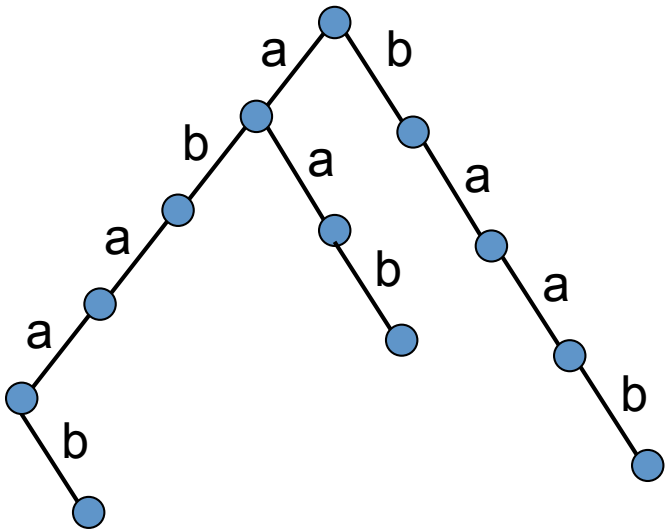
baab

aab

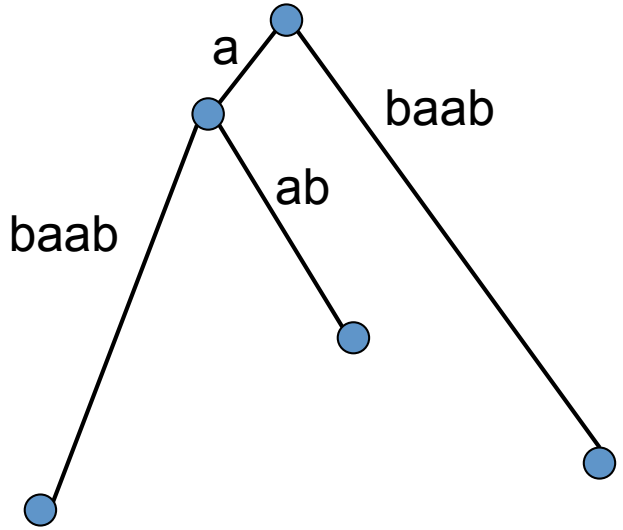
ab

b

Trie(abaab)



Tree(abaab)



E. Ukkonen: http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

Suffix tree and suffix array techniques for pattern analysis in strings

Esko Ukkonen

Univ Helsinki

Erice School 30 Oct 2005

Algorithms for combinatorial string matching?

- deep beauty? +-
- shallow beauty? +
- applications? ++
- intensive algorithmic miniatures
- sources of new problems:

  text processing, DNA, music,…

E. Ukkonen: http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

Analysis of a string of symbols

- T = h a t t i v a t t i *'text'*
- P = a t t *'pattern'*

- Find the occurrences of P in T:

  h a t t i v a t t i

- Pattern synthesis: #(t) = 4 #(atti) = 2 # (t****t) = 2

E. Ukkonen: http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

Pattern finding & synthesis problems

- $T = t_1 t_2 \ldots t_n$, $P = p_1 p_2 \ldots p_n$, strings of symbols in finite alphabet

- Indexing problem: Preprocess T (build an index structure) such that the occurrences of different patterns P can be found fast
  - static text, any given pattern P

- Pattern synthesis problem: Learn from T new patterns that occur surprisingly often

- What is a pattern? Exact substring, approximate substring, with generalized symbols, with gaps, …

1. **Suffix tree**
2. Suffix array
3. Some applications
4. Finding motifs

E. Ukkonen: http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

The suffix tree Tree(T) of T

- data structure suffix tree, Tree(T), is compacted trie that represents all the suffixes of string T
- **linear size: |Tree(T)| = O(|T|)**
- **can be constructed in linear time O(|T|)**
- has *myriad virtues* (A. Apostolico)
- is well-known: 366 000 Google hits

E. Ukkonen: http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt

Suffix trie and suffix tree

abaab

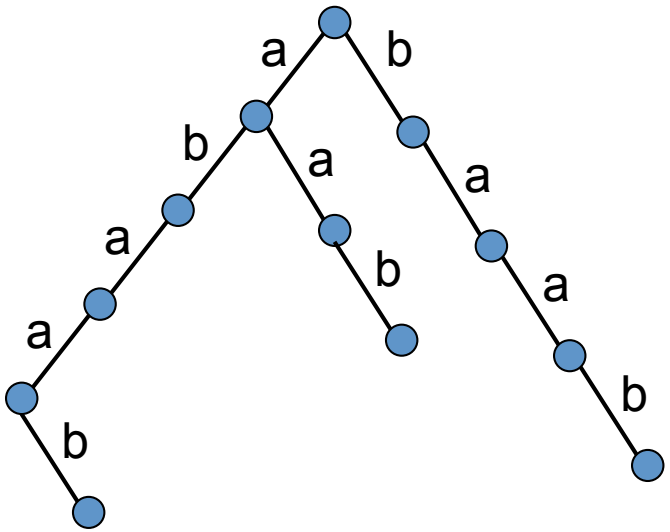
baab

aab

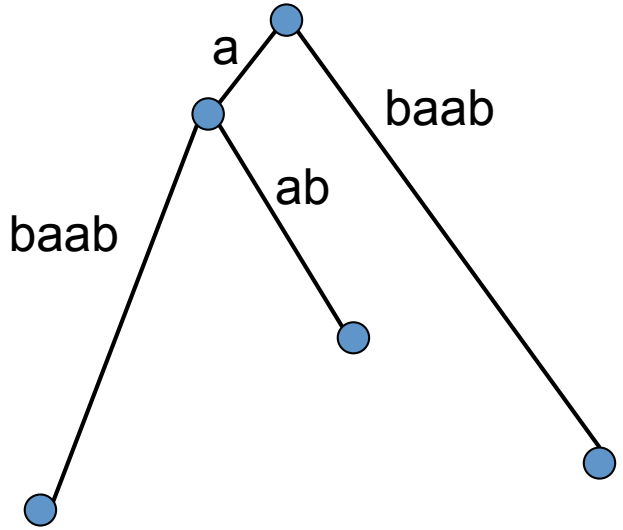
ab

b

Trie(abaab)



Tree(abaab)



Trie(T) can be large

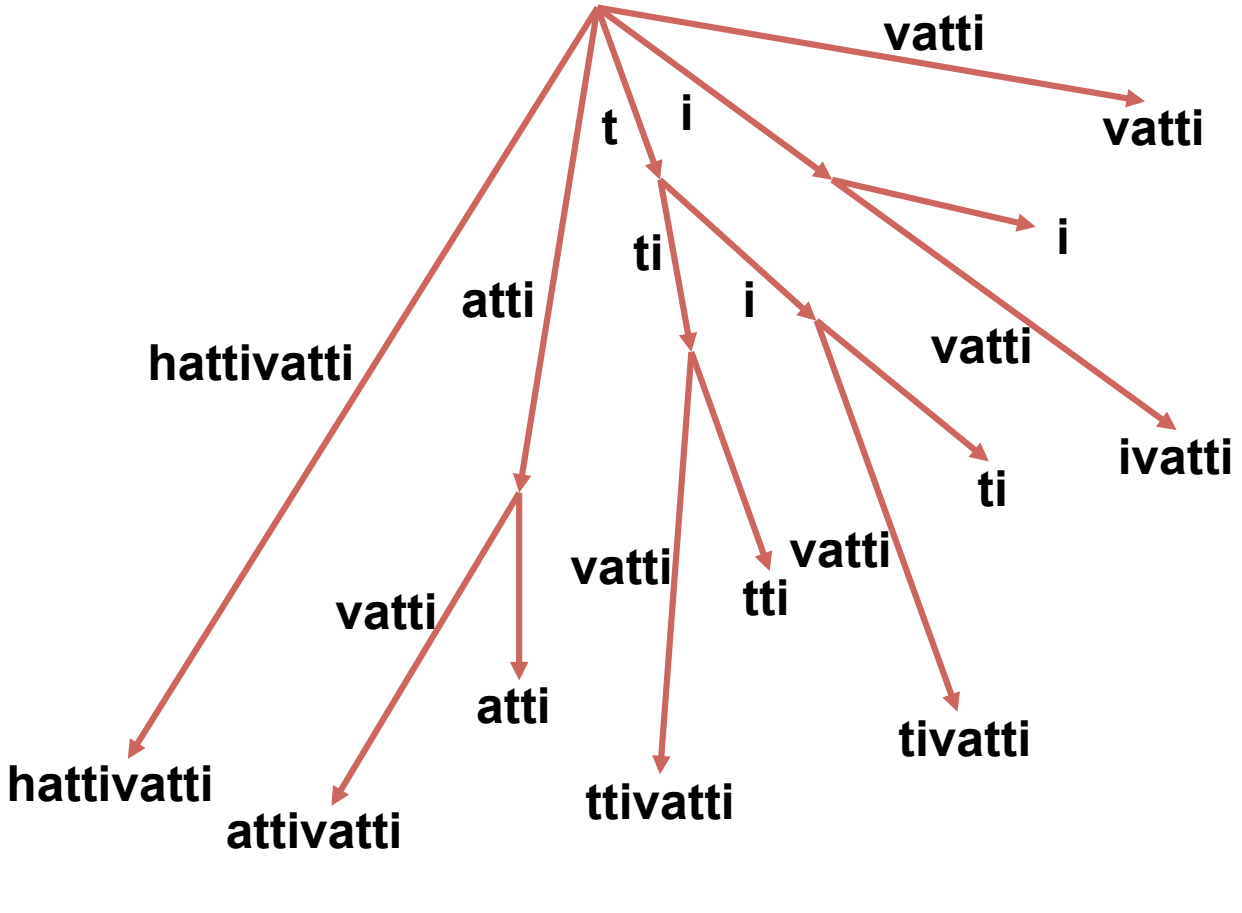
- $|\text{Trie}(T)| = O(|T|^2)$
- bad example: $T = a^n b^n$
- Trie(T) can be seen as a DFA: language accepted = the suffixes of T
- minimize the DFA => directed cyclic word graph ('DAWG')

Tree(T) is of linear size

- only the internal branching nodes and the leaves represented explicitly
- edges labeled by substrings of T
- v = node(α) if the path from root to v spells α
- one-to-one correspondence of leaves and suffixes
- |T| leaves, hence < |T| internal nodes
- |Tree(T)| = O(|T| + size(edge labels))

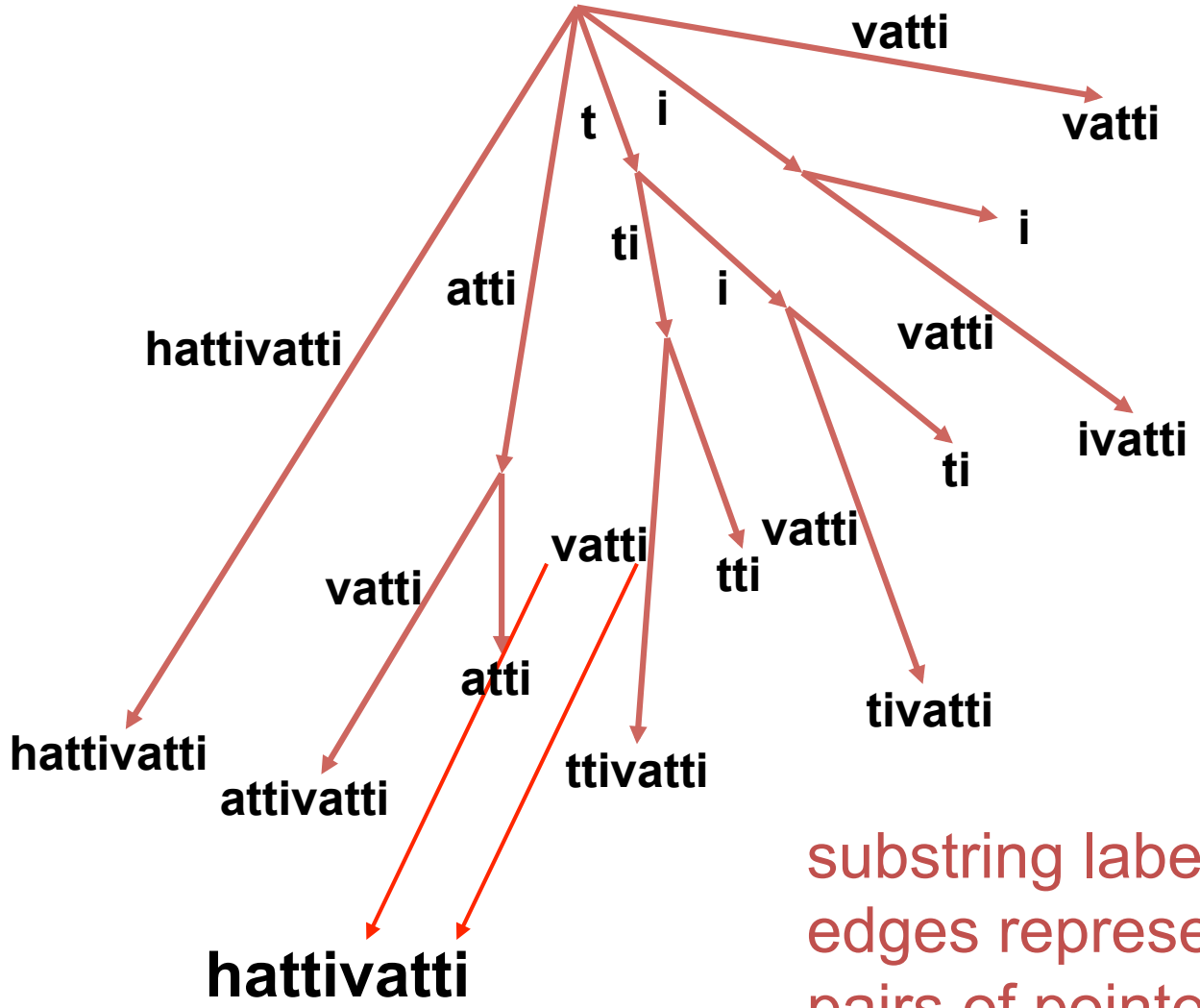
Tree(hattivatti)

hattivatti
attivatti
ttivatti
tivatti
ivatti
vatti
atti
tti
ti
i



Tree(hattivatti)

hattivatti
attivatti
ttivatti
tivatti
ivatti
vatti
atti
tti
ti
i



substring labels of edges represented as pairs of pointers

Tree(hattivatti)

hattivatti

attivatti

ttivatti

tivatti

ivatti

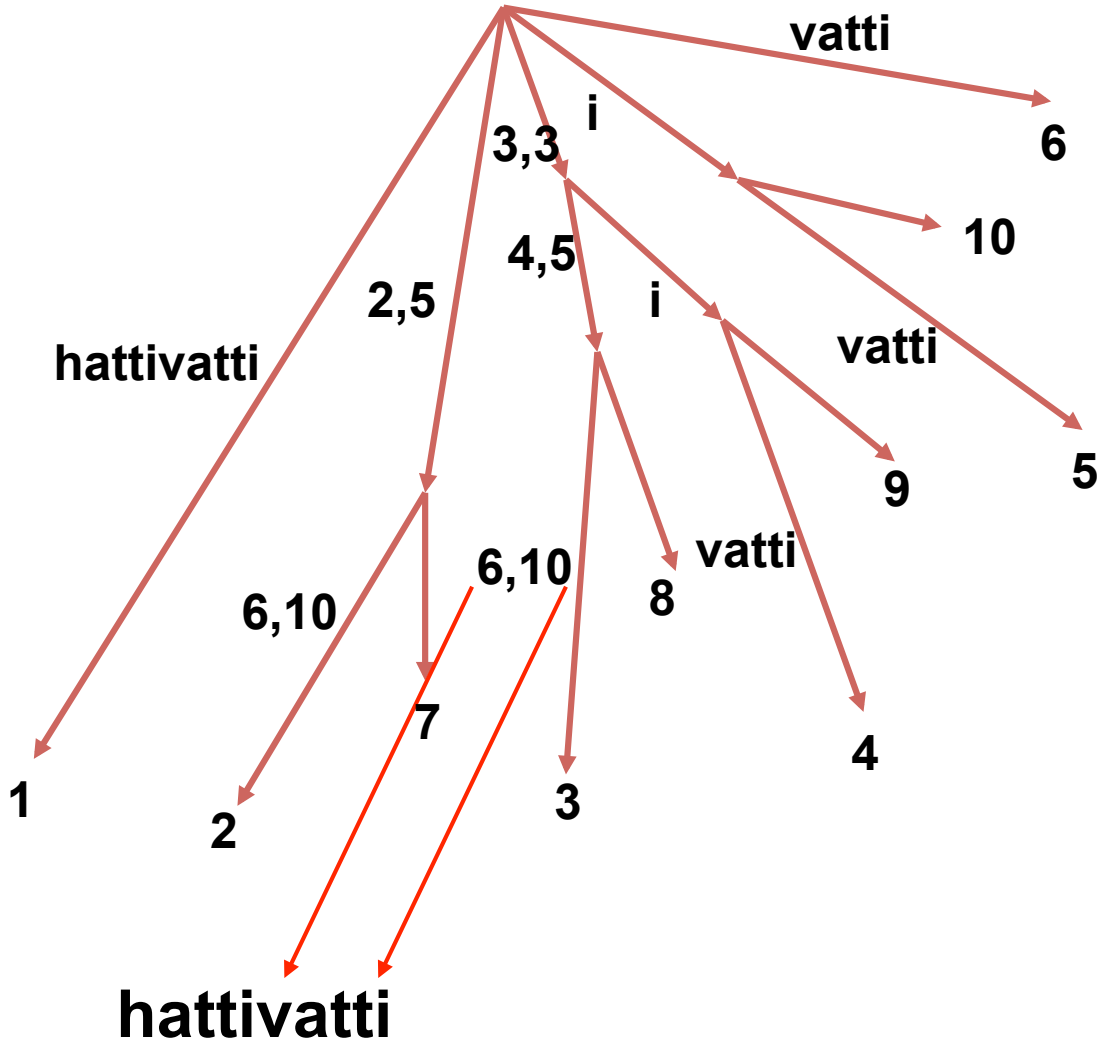
vatti

atti

tti

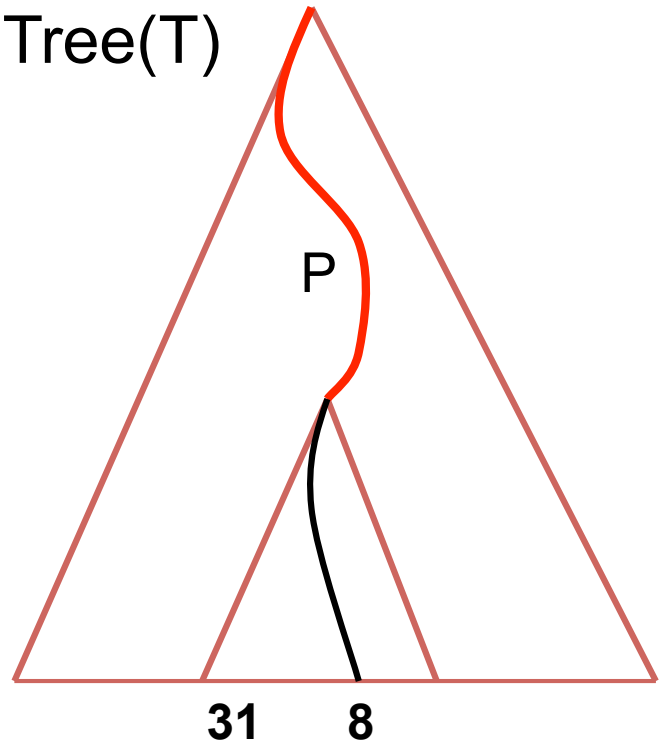
ti

i



Tree(T) is *full* text index

P occurs in T at
locations 8, 31, …

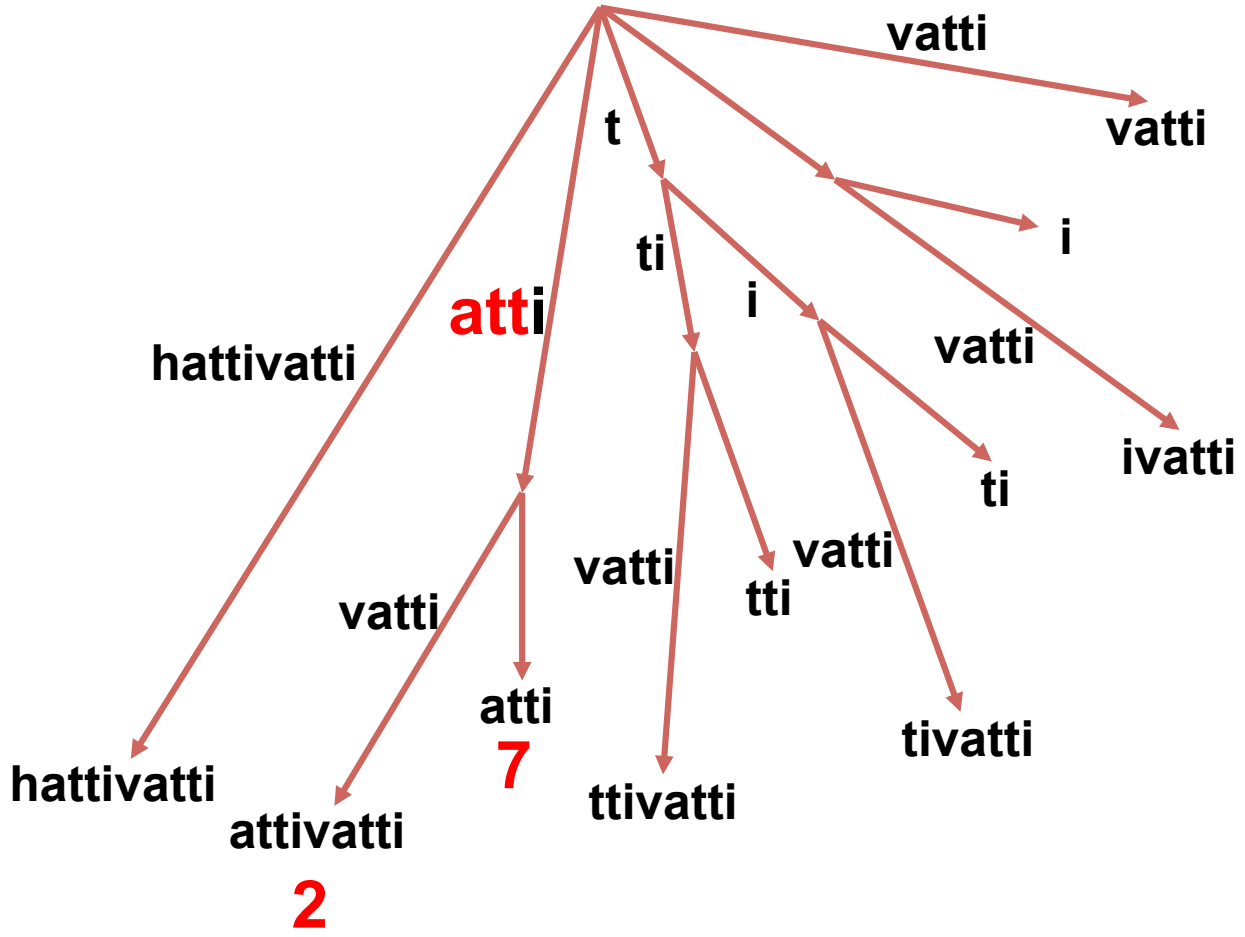


P occurs in T ⇔ P is a prefix of some suffix of T
⇔ Path for P exists in Tree(T)

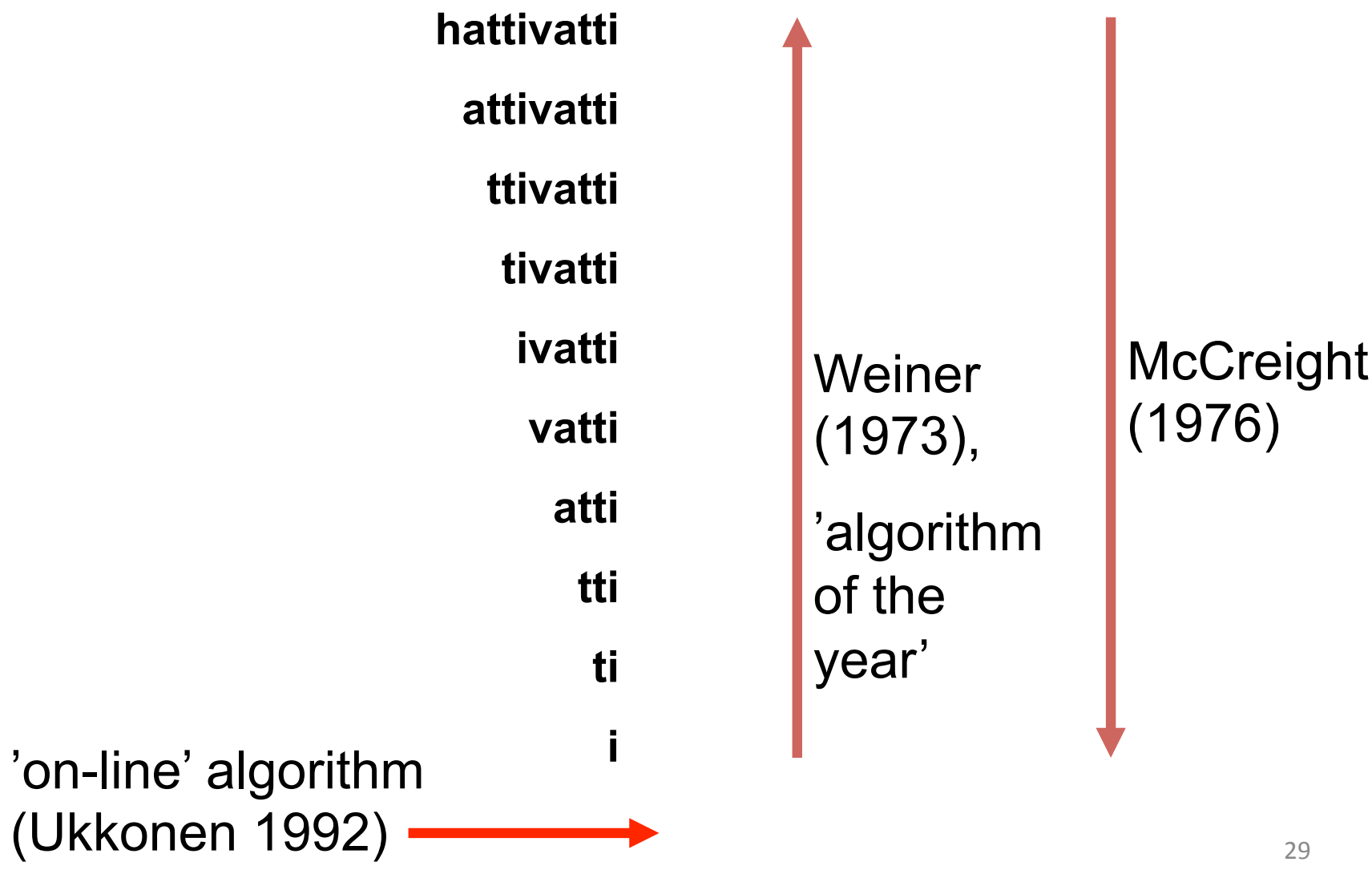
All occurrences of P in time O(|P| + #occ)

Find att from Tree(hattivatti)

hattivatti
attivatti
ttivatti
tivatti
ivatti
vatti
atti
tti
ti
i



Linear time construction of Tree(T)

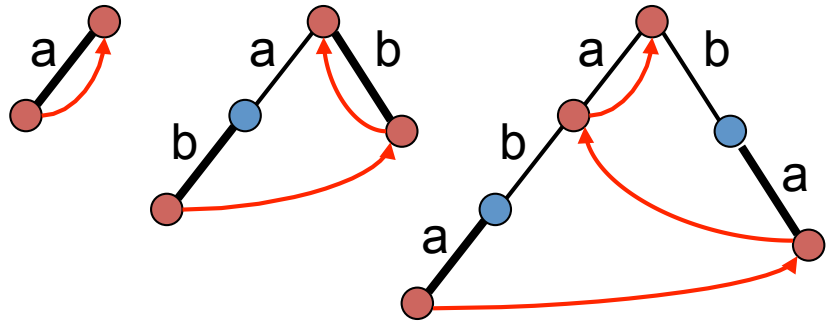


On-line construction of Trie(T)

- $T = t_1 t_2 \ldots t_n\$$
- $P_i = t_1 t_2 \ldots t_i$ i:th prefix of *T*
- on-line idea: update *Trie(P_i)* to *Trie(P_{i+1})*
- => very simple construction

Trie(abaab)

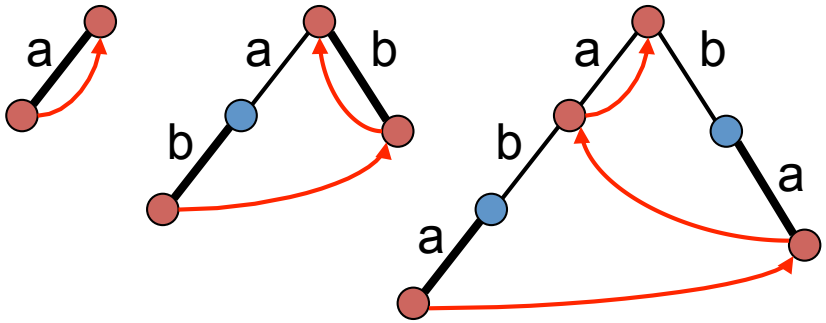
Trie(a) *Trie(ab)* *Trie(aba)*



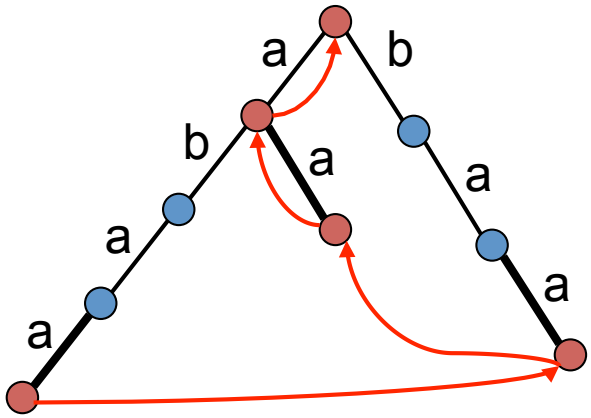
aba**a**
ba**a**
a**a**
ε**a**
ε

chain of links connects the end points of current suffixes

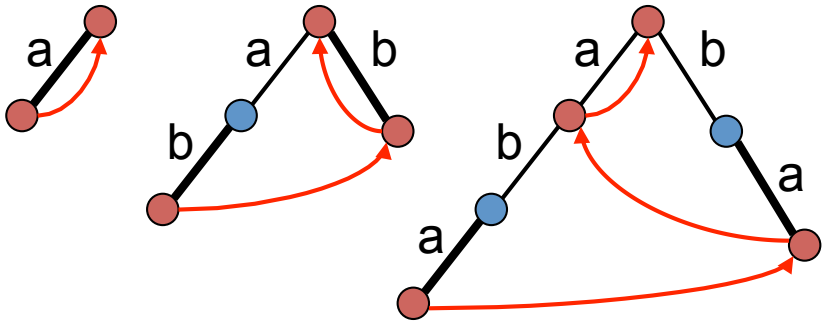
Trie(abaab)



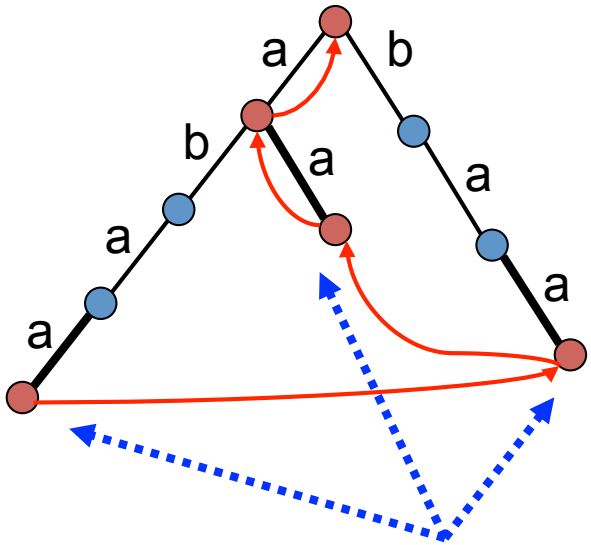
Trie(abaa)



Trie(abaab)

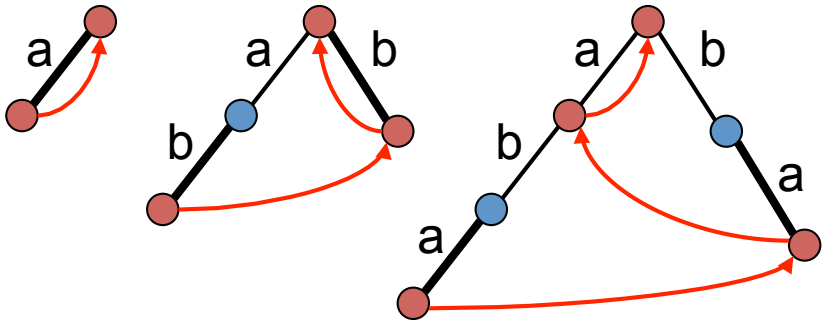


Trie(abaa)

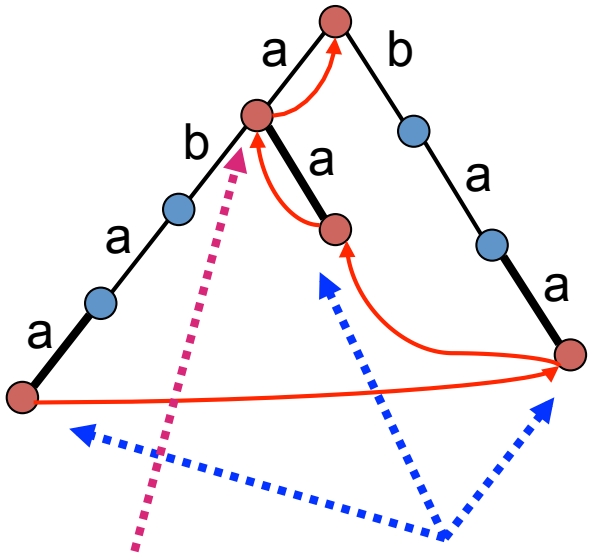


Add next symbol = b

Trie(abaab)



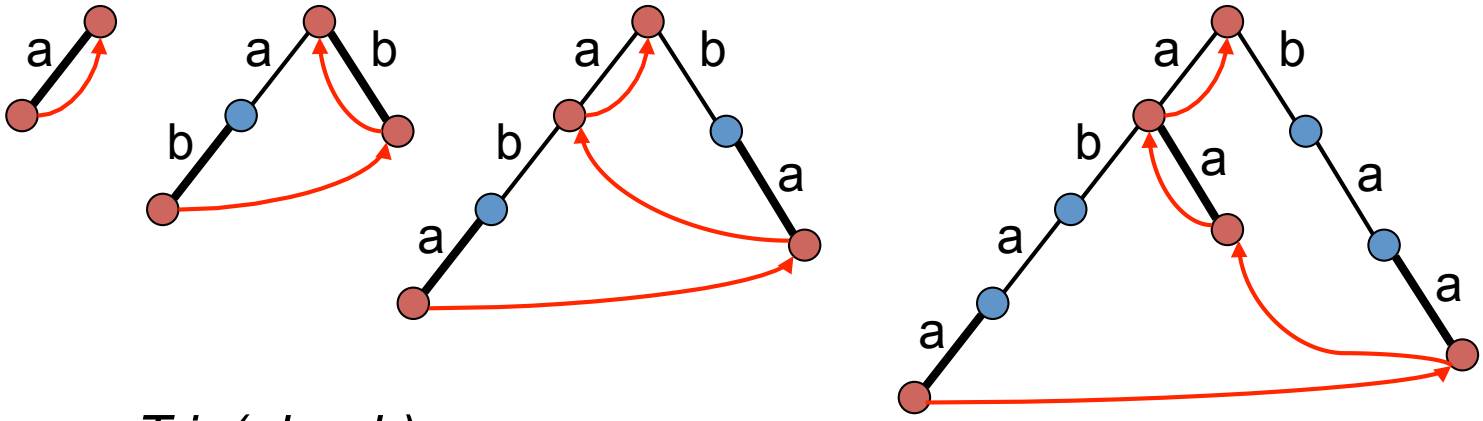
Trie(abaa)



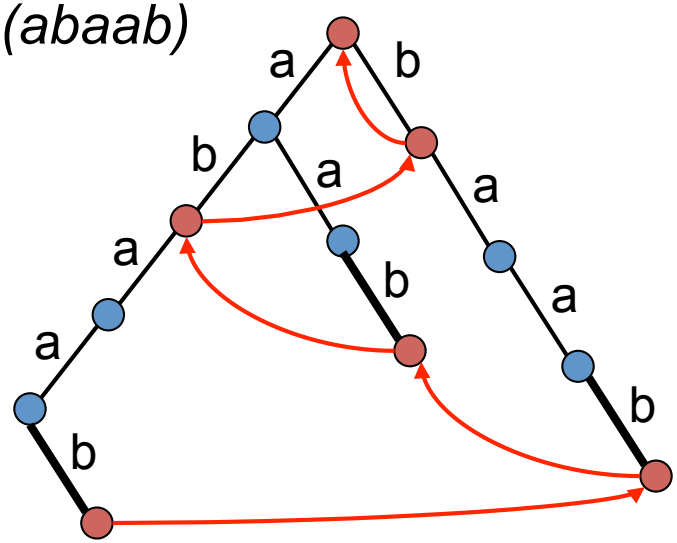
Add next symbol = b

From here on b-arc already exists

Trie(abaab)

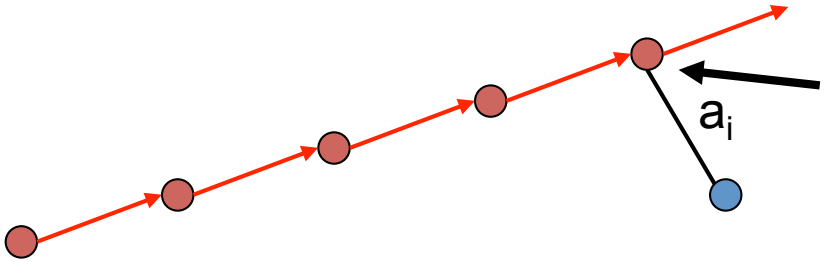


Trie(abaab)



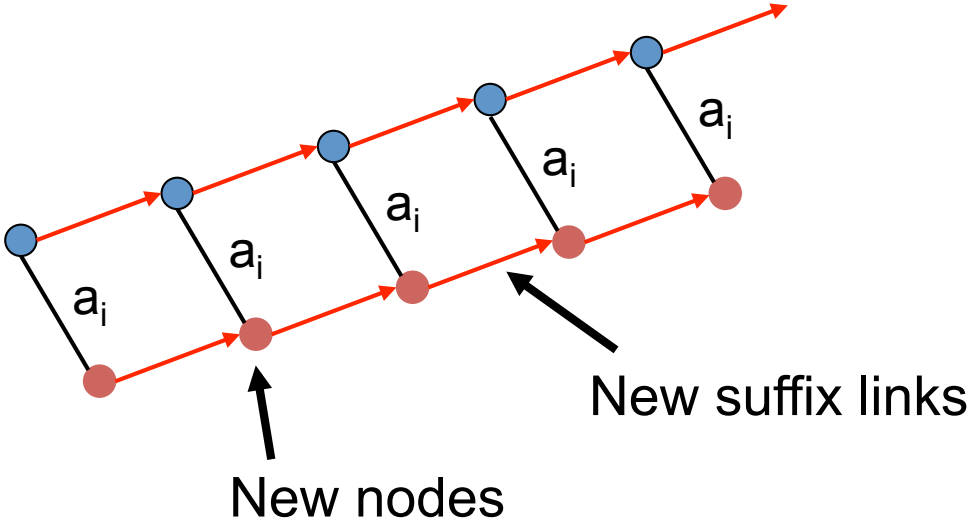
What happens in *Trie(P_i)* => *Trie(P_{i+1})* ?

Before



From here on the a_i-arc exists already => stop updating here

After



What happens in *Trie(P_i)* => *Trie(P_{i+1})* ?

- time: O(size of Trie(T))
- suffix links: slink
  (node(*αα*)) = node(*α)*

On-line procedure for suffix trie

1. Create *Trie(t_1):* nodes *root* and *v*, an arc *son(root, t_1)* = *v,* and suffix links *slink(v) := root* and *slink(root) := root*
2. **for** *i := 2* to *n* **do begin**
3. v_{i-1} := leaf of *Trie($t_1 \ldots t_{i-1}$)* for string $t_1 \ldots t_{i-1}$ (i.e., the deepest leaf)
4. *v := v_{i-1}; v´ := 0*
5. **while** node *v* has no outgoing arc for t_i **do begin**
6. Create a new node *v´´* and an arc *son(v,t_i) = v´´*
7. **if** *v´ ≠ 0* **then** *slink(v) := v´´*
8. *v := slink(v); v´ := v´´* **end**
9. **for** the node *v´´* such that *v´´= son(v,t_i)* **do**
   if *v´´ = v´* **then** *slink(v') := root* **else** *slink(v´) := v´´*

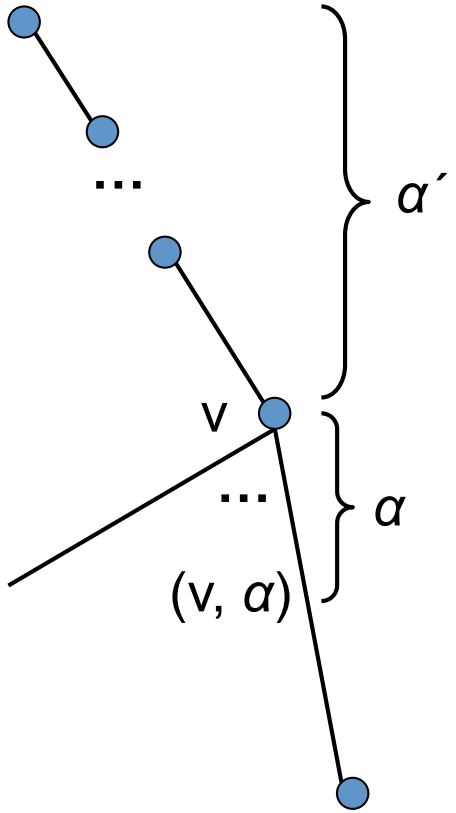
Suffix trees on-line

- ’compacted version’ of the on-line trie construction: simulate the construction on the linear size tree instead of the trie => time O(|T|)
- all trie nodes are conceptually still needed => *implicit* and *real* nodes

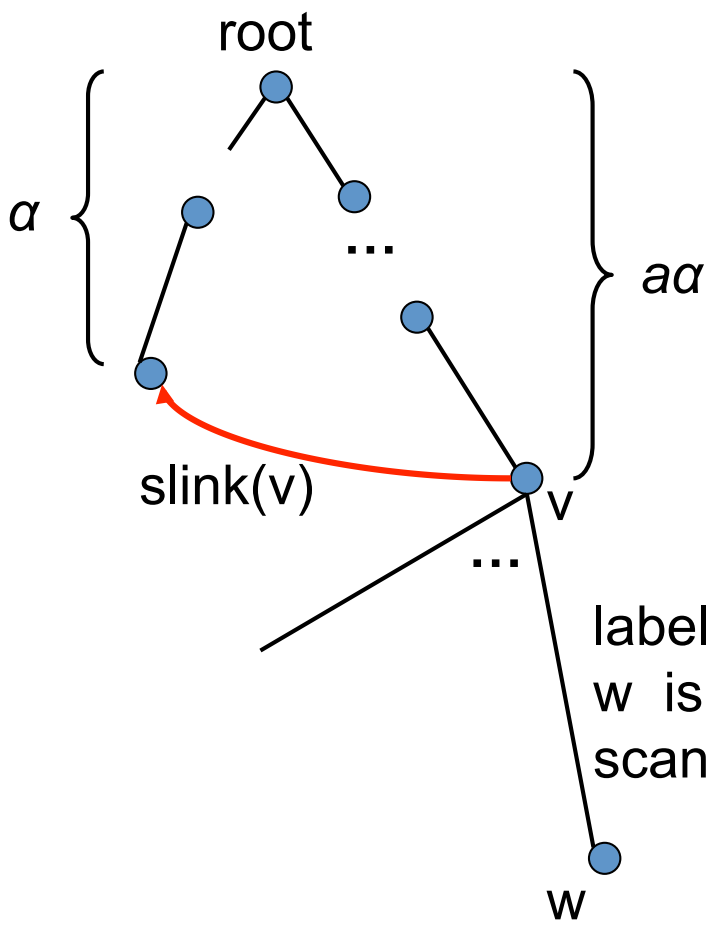
Implicit and real nodes

- Pair (v,α) is an *implicit node* in Tree(T) if v is a node of Tree and α is a (proper) prefix of the label of some arc from *v*. If α is the empty string then *(v, α)* is a *'real'* node (= *v*).
- Let *v = node(α´)* in Tree(T). Then implicit node *(v, α)* represents *node(α´α)* of Trie(T)

Implicit node



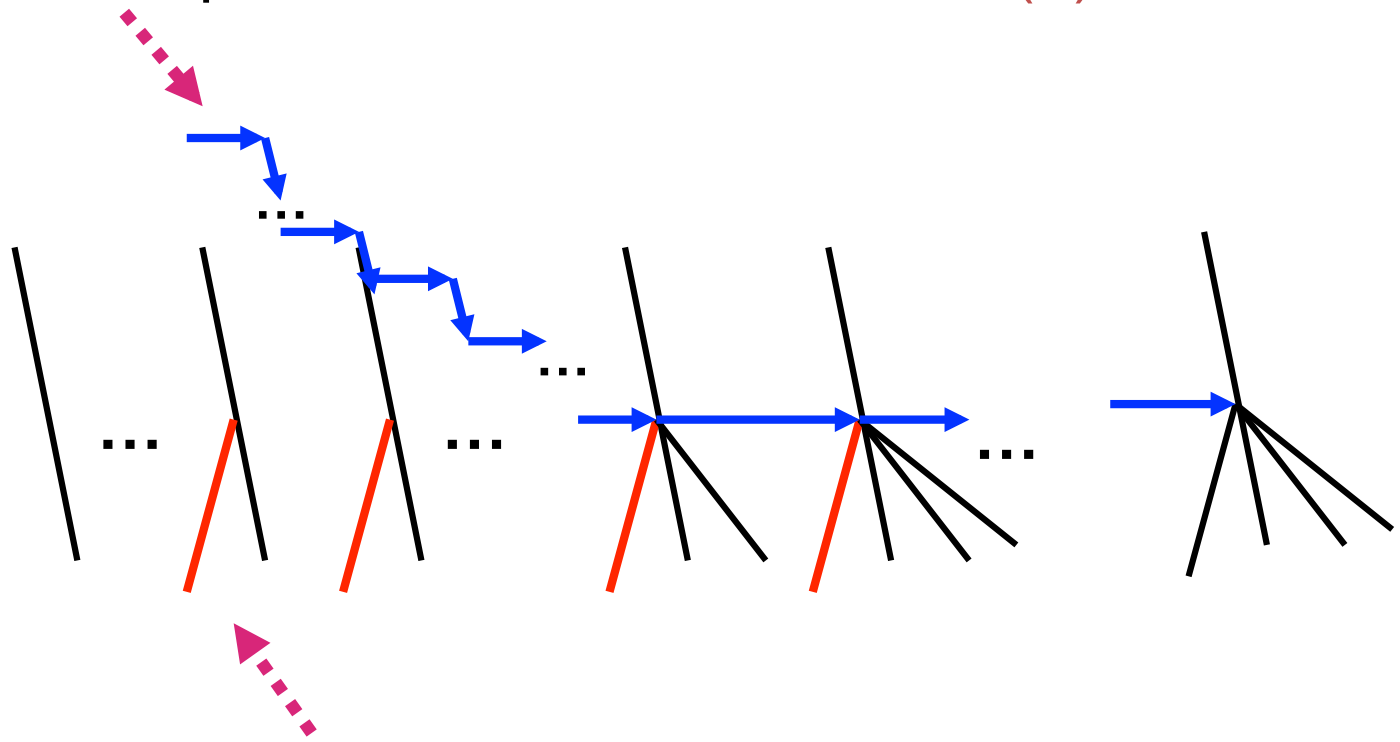
Suffix links and open arcs



label [i,*] instead of [i,j] if w is a leaf and j is the scanned position of T

Big picture

suffix link path traversed: total work O(n)



new arcs and nodes created:
total work O(size(Tree(T))

On-line procedure for suffix tree

Input: string $T = t_1 t_2 \ldots t_n\$$

Output: *Tree(T)*

Notation: *son(v,α) = w* iff there is an arc from *v* to *w* with label *α*

son(v,ε) = v

Function *Canonize(v, α):*

while *son(v, α´) ≠ 0* where $\alpha = \alpha' \alpha'', |\alpha'| > 0$ **do**

v := son(v, α´); α := α´´

return *(v, α)*

Suffix-tree on-line: main procedure

Create *Tree(t_1); slink(root) := root*

(v, *α) := (root, ε)* /* *(v, α)* is the start node */

for *i := 2* **to** *n+1* **do**

v´ := 0

while there is no arc from *v* with label prefix *αt_i* **do**

if *α ≠ ε* **then** /* divide the arc *w = son(v, αη)* into two */

son(v, α) := v´´; son(v´´,t_i) := v´´´; son(v´´,η) := w

else

son(v,t_i) := v´´´; v´´ := v

if *v´ ≠ 0* **then** *slink(v´) := v´´*

v´ := v´´; v := slink(v); (v, α) := Canonize(v, α)

if *v´ ≠ 0* **then** *slink(v´) := v*

(v, α) := Canonize(v, αt_i) /* *(v, α)* = start node of the next round */

The actual time and space

- |Tree(T)| is about 20|T| in practice
- brute-force construction is O(|T|log|T|) for random strings as the average depth of internal nodes is O (log|T|)
- difference between linear and brute-force constructions not necessarily large (Giegerich & Kurtz)
- truncated suffix trees: k symbols long prefix of each suffix represented (Na et al. 2003)
- alphabet independent linear time (Farach 1997)

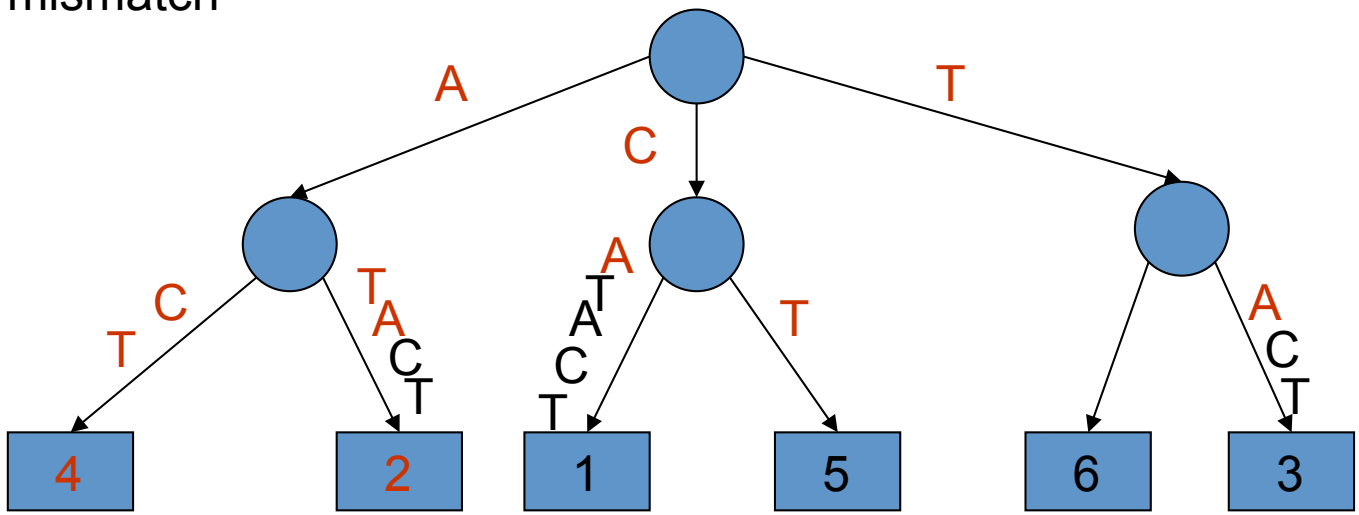
Applications of Suffix Trees

- **Dan Gusfield:** Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Hardcover - 534 pages 1st edition (January 15, 1997). Cambridge Univ Pr (Short); ISBN: 0521585198. - book

- **APL1:** Exact String Matching Search for P from text S. Solution 1: build STree(S) - one achieves the same O(n+m) as Knuth-Morris-Pratt, for example!
- Search from the suffix tree is O(|P|)
- **APL2:** Exact set matching Search for a set of patterns P

Back to backtracking

ACA, 1 mismatch



Same idea can be used to many other forms of approximate search, like Smith-Waterman, position-restricted scoring matrices, regular expression search, etc.

Applications of Suffix Trees

- **APL3:** substring problem for a database of patterns Given a set of strings S=S1, ... , Sn --- a database Find all Si that have P as a substring
- Generalized suffix tree contains all suffixes of all Si
- Query in time O(|P|), and can identify the LONGEST common prefix of P in all Si

Applications of Suffix Trees

- **APL4:** Longest common substring of two strings
- Find the longest common substring of S and T.
- Overall there are potentially O(n^2) such substrings, if n is the length of a shorter of S and T
- Donald Knuth once (1970) **conjectured** that linear-time algorithm is **impossible**.
- Solution: construct the STree(S+T) and find the node deepest in the tree that has suffixes from both S and T in subtree leaves.
- Ex: S= *superiorcalifornialives* T= *sealiver* have both a substring *xxxxxx.*

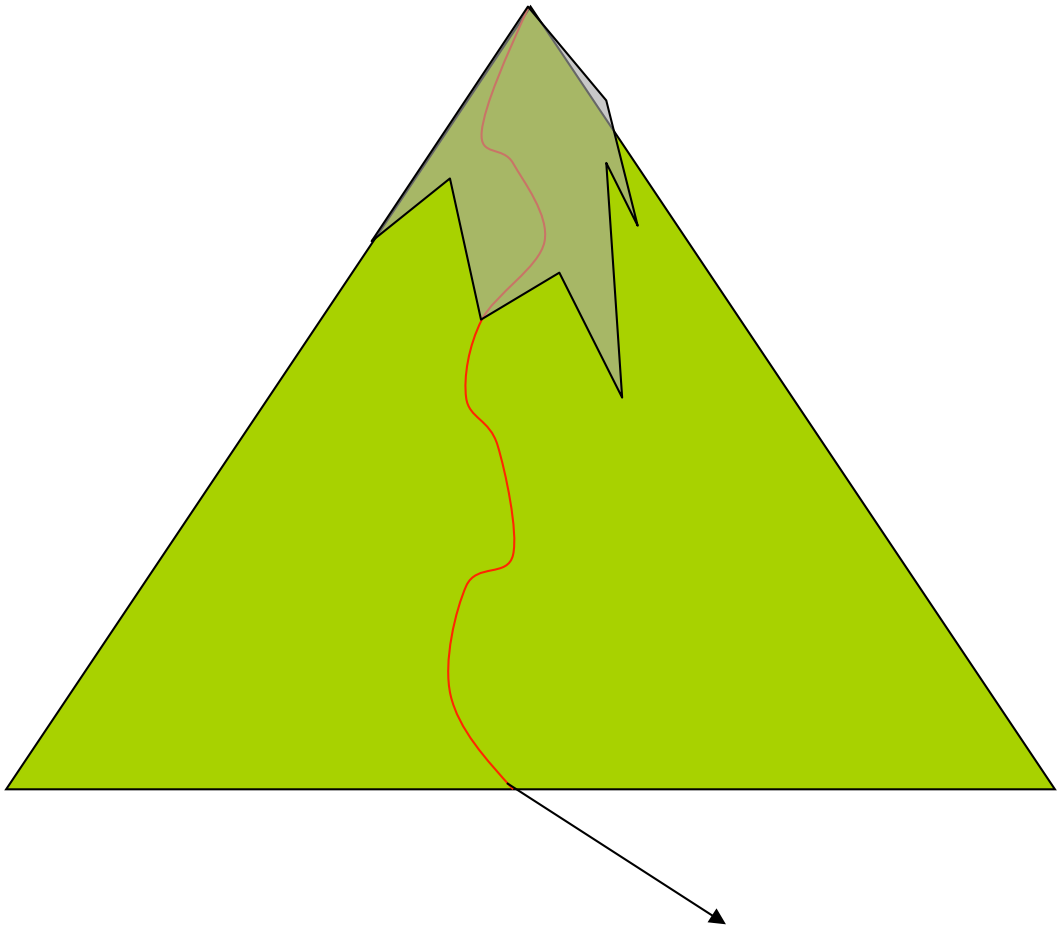
Simple analysis task: LCSS

- Let LCSSA(A,B) denote the longest common substring two sequences A and B. E.g.:
  - LCSS(AGATCTATCT,CGCCTCTATG)=TCTAT.
- A good solution is to build suffix tree for the shorter sequence and make a *descending suffix walk* with the other sequence.

High-throughput genome-scale sequence analysis and mapping using compressed data structures

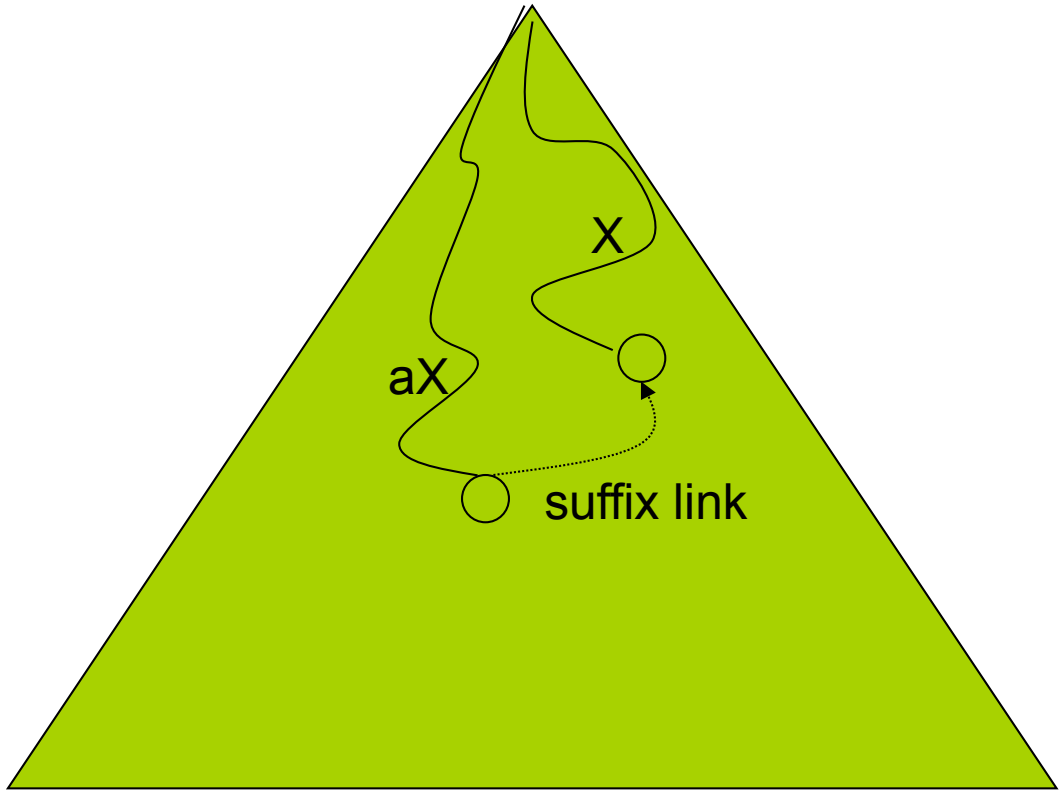
Veli Mäkinen
Department of Computer Science
University of Helsinki

Solution: backtracking with suffix tree

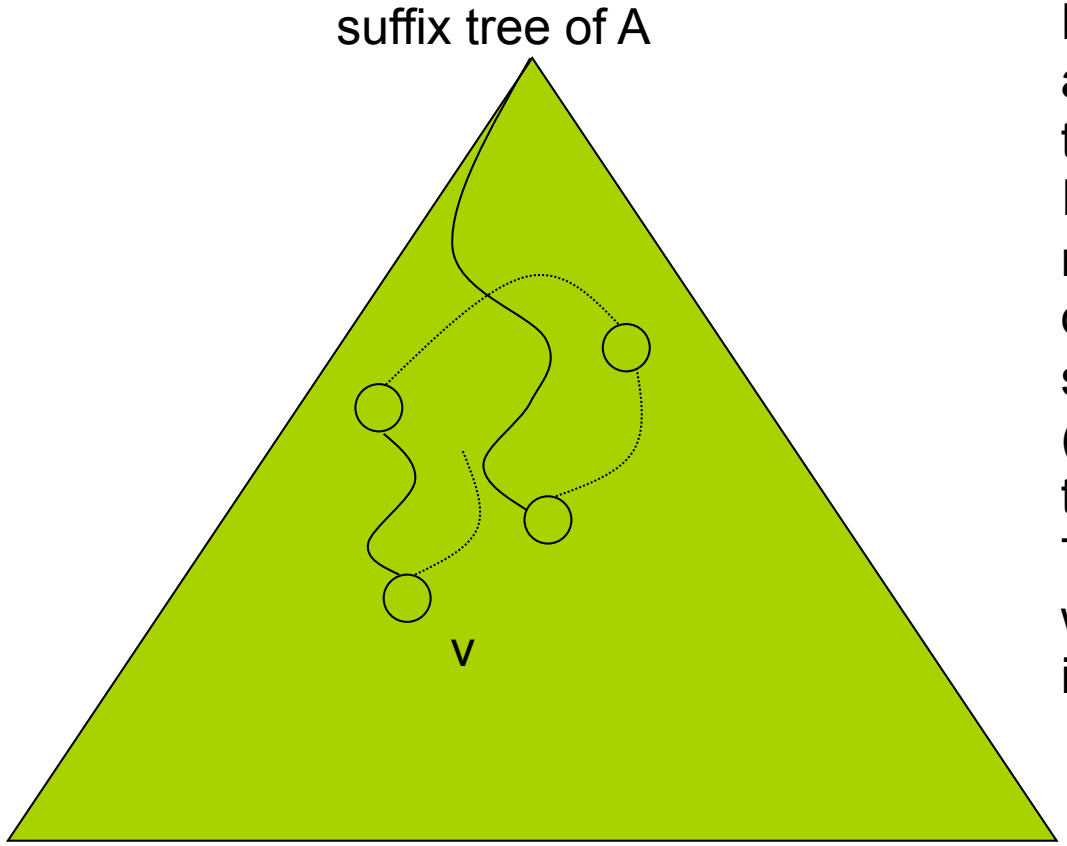


...ACACATTATCACAGGCATCGGCATTAGCGATCGAGTCG.....

Suffix link

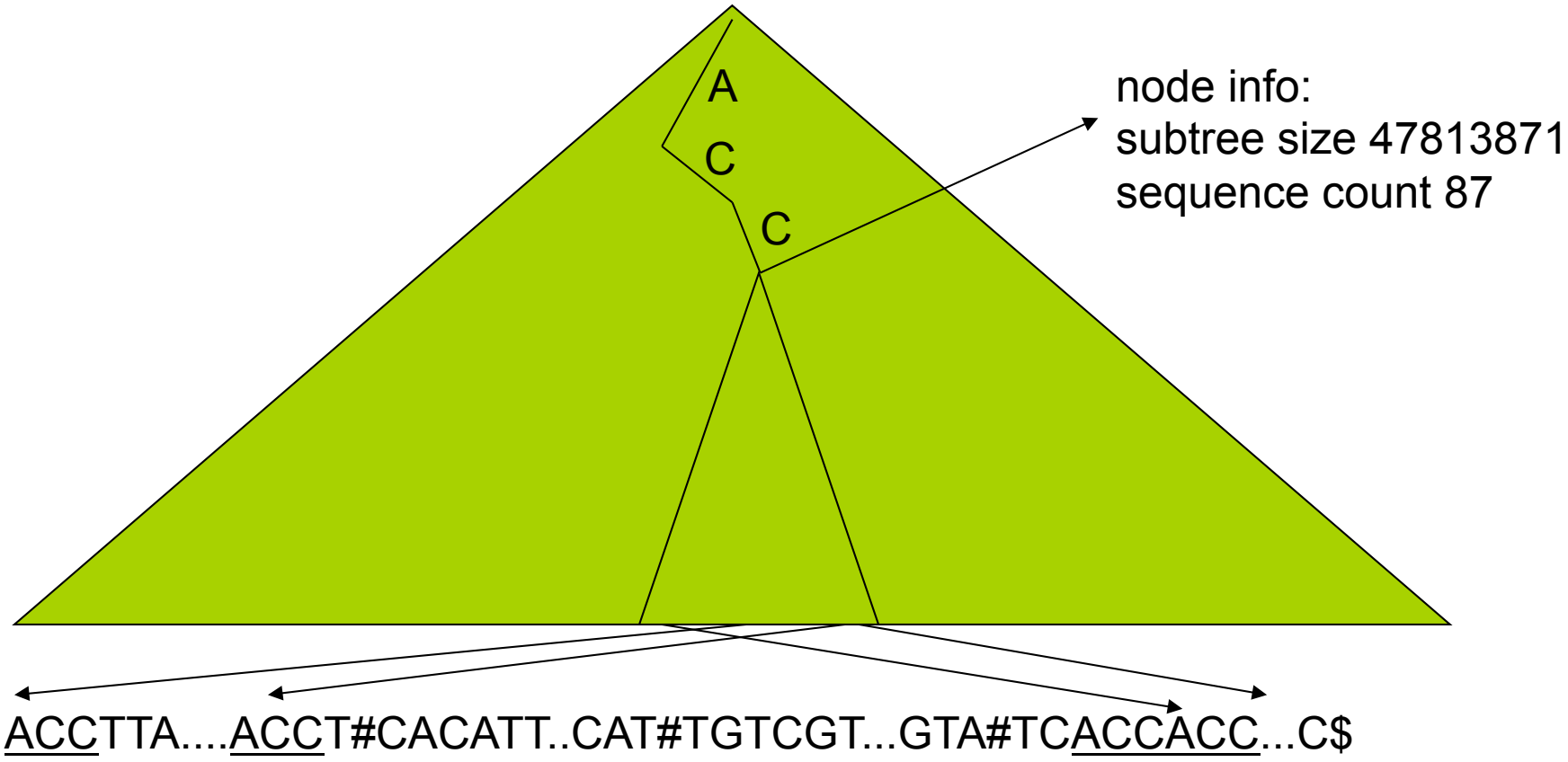


Descending suffix walk

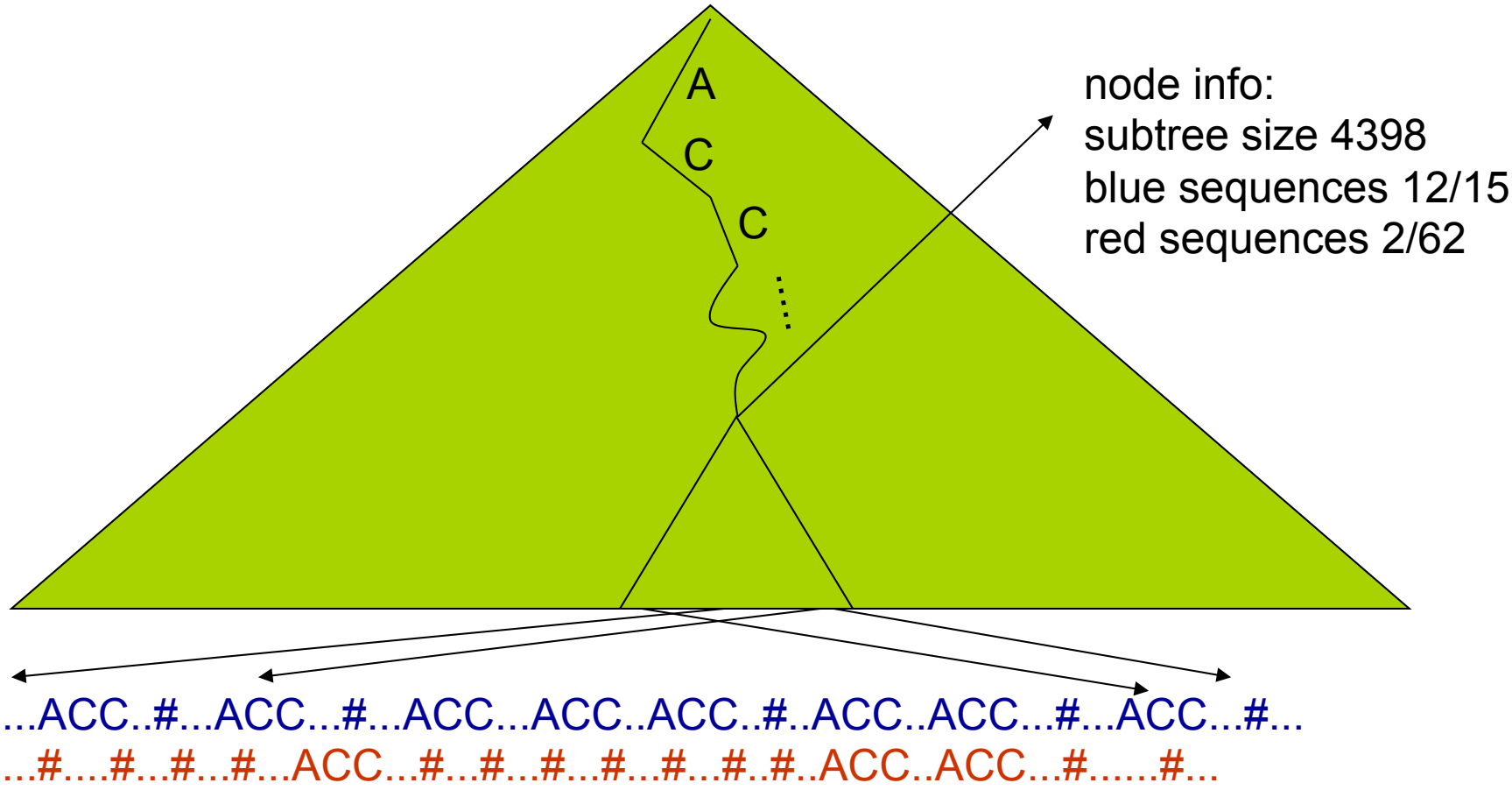


Read B left-to-right, always going down in the tree when possible.
If the next symbol of B does not match any edge label on current position, take suffix link, and try again.
(Suffix link in the root to itself emits a symbol).
The node v encountered with largest string depth is the solution.

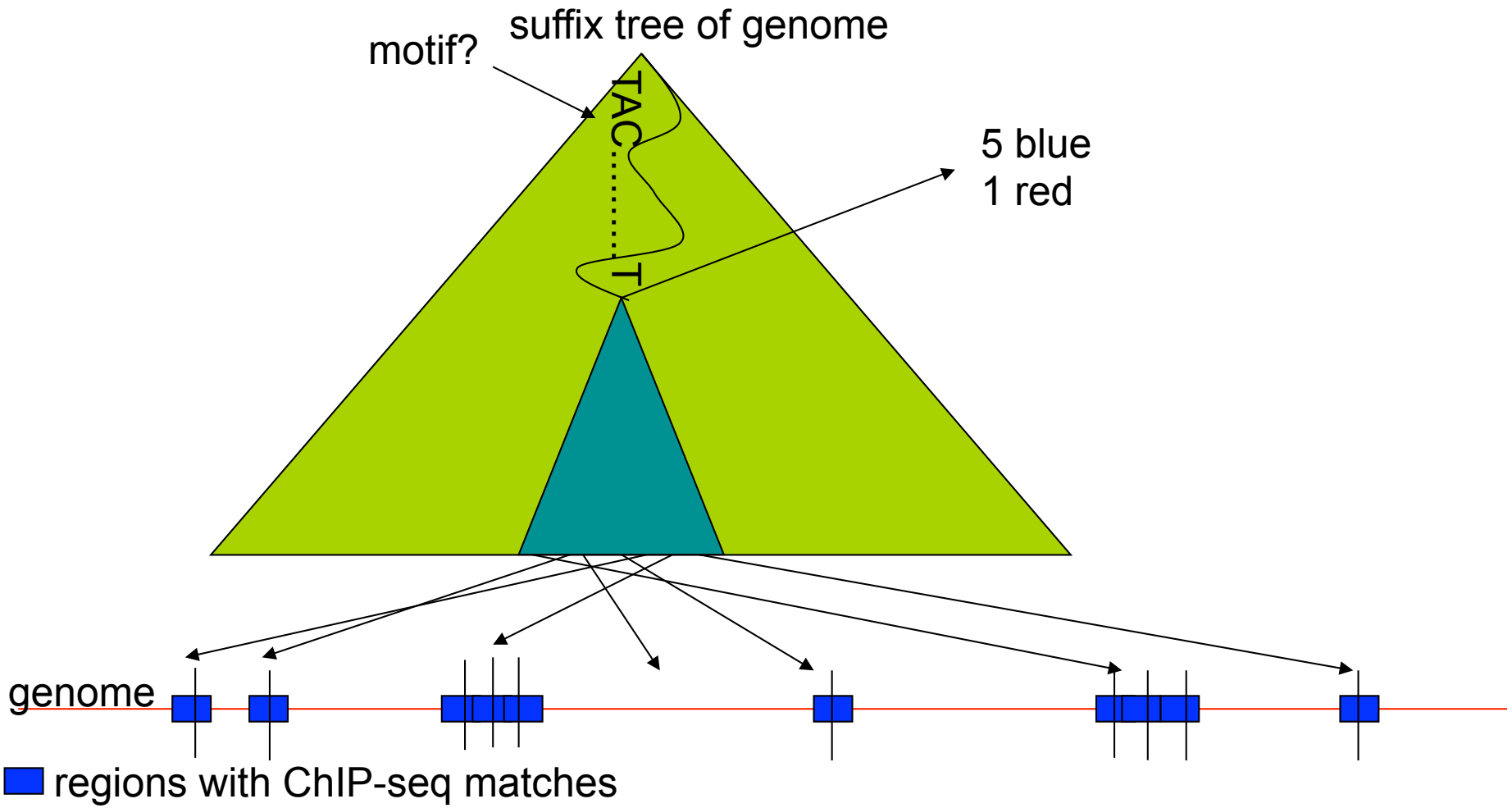
Another common tool: Generalized suffix tree



Generalized suffix tree application



Case study continued



Properties of suffix tree

- Suffix tree has n leaves and at most n-1 internal nodes, where n is the total length of all sequences indexed.
- Each node requires constant number of integers (pointers to first child, sibling, parent, text range of incoming edge, statistics counters, etc.).
- Can be constructed in linear time.

Properties of suffix tree... in practice

- Huge overhead due to pointer structure:
  - Standard implementation of suffix tree for human genome requires over 200 GB memory!
  - A careful implementation (using log n -bit fields for each value and array layout for the tree) still requires over 40 GB.
  - Human genome itself takes less than 1 GB using 2-bits per bp.
- 2012: huge improvements made over last 5-10 years

Applications of Suffix Trees

- **APL5**: Recognizing DNA contamination Related to DNA sequencing, search for longest strings (longer than threshold) that are present in the DB of sequences of other genomes.
- **APL6**: Common substrings of more than two strings Generalization of APL4, can be done in linear (in total length of all strings) time

Applications of Suffix Trees

- **APL7**: Building a directed graph for exact matching: **Suffix graph** - directed acyclic word graph (DAWG), a **smallest finite state automaton** recognizing all suffixes of a string S. This automaton can recognize membership, but not tell which suffix was matched.
- Construction: merge isomorfic subtrees.
- Isomorfic in Suffix Tree when exists suffix link path, and subtrees have equal nr. of leaves.

Applications of Suffix Trees

- APL8: A reverse role for suffix trees, and major space reduction Index the pattern, not tree...
- Matching statistics.
- APL10: All-pairs suffix-prefix matching For all pairs S_i, S_j, find the longest matching suffix-prefix pair. Used in shortest common superstring generation (e.g. DNA sequence assembly), EST alignmentm etc.

Applications of Suffix Trees

- APL11: Finding all maximal repetitive structures in linear time
- APL12: Circular string linearization e.g. circular chemical molecules in the database, one wants to lienarize them in a canonical way...
- APL13: Suffix arrays - more space reduction will touch that separately

Applications of Suffix Trees

- APL14: Suffix trees in genome-scale projects
- APL15: A Boyer-Moore approach to exact set matching
- APL16: Ziv-Lempel data compression
- APL17: Minimum length encoding of DNA

Applications of Suffix Trees

- Additional applications Mostly exercises...
- Extra feature: **CONSTANT time lowest common ancestor retrieval (LCA)** Andmestruktuur mis võimaldab leida konstantse ajaga alumist ühist vanemat (see vastab pikimale ühisele prefixile!) on võimalik koostada lineaarse ajaga.
- APL: Longest common extension: a bridge to inexact matching
- APL: **Finding all maximal palindromes in linear time** Palindrome reads from central position the same to left and right. E.g.: kirik, saippuakivikauppias.
- Build the suffix tree of S and inverted S (aabcbad => aabcbad#dabcbaa) and using the LCA one can ask for any position pair (i, 2i-1), the longest common prefix in constant time.
- The whole problem can be solved in O(n).

Applications of Suffix Trees

- APL: Exact matching with wild cards
- APL: The k-mismatch problem
- Approximate palindromes and repeats
- Faster methods for tandem repeats
- A linear-time solution to the multiple common substring problem
- And many-many more ...

1. Suffix tree
2. **Suffix array**
3. Some applications
4. Finding motifs

Suffixes - sorted

hattivatti	ε
attivatti	atti
ttivatti	attivatti
tivatti	hattivatti
ivatti	i
vatti	ivatti
atti	ti
tti	tivatti
ti	tti
i	ttivatti
ε	vatti

- Sort all suffixes. Allows to perform binary search!

Suffix array: example

1	hattivatti	ε	11
2	attivatti	atti	7
3	ttivatti	attivatti	2
4	tivatti	hattivatti	1
5	ivatti	i	10
6	vatti	ivatti	5
7	atti	ti	9
8	tti	tivatti	4
9	ti	tti	8
10	i	ttivatti	3
11	ε	vatti	6

- suffix array = lexicographic order of the suffixes

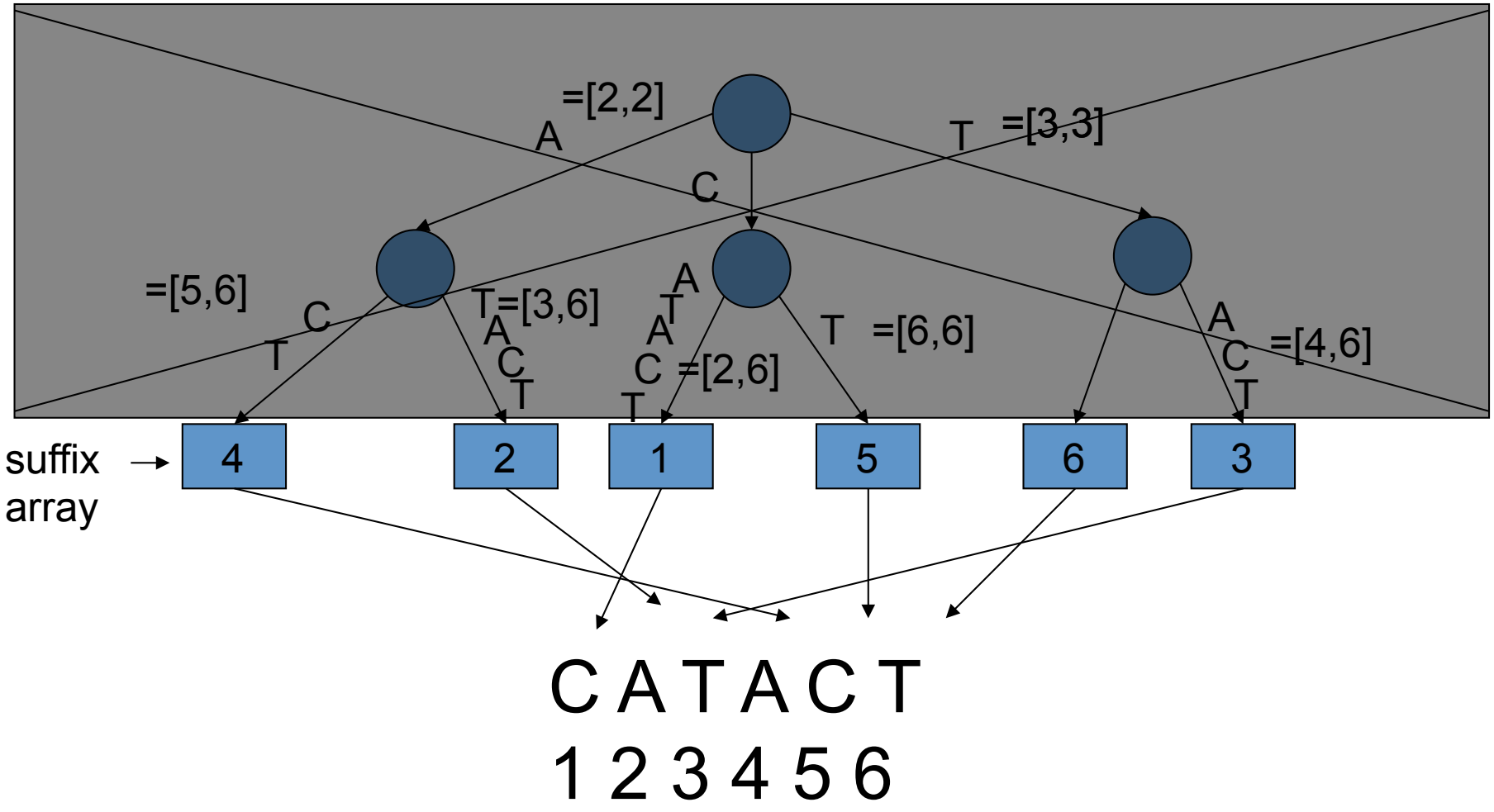
Suffix array construction: sort!

- suffix array = lexicographic order of the suffixes

Suffix array

- suffix array SA(T) = an array giving the lexicographic order of the suffixes of T
- space requirement: 5|T|
- practitioners like suffix arrays (simplicity, space efficiency)
- theoreticians like suffix trees (explicit structure)

Reducing space: suffix array



Suffix array

- Many algorithms on suffix tree can be simulated using *suffix array*...
  - ... and couple of additional arrays...
  - ... forming so-called *enhanced suffix array*...
  - ... leading to the similar space requirement as careful implementation of suffix tree
- Not a satisfactory solution to the space issue.

Pattern search from suffix array

hattivatti	ε	11
attivatti	atti	7
ttivatti	attivatti	2
tivatti	hattivatti	1
ivatti	i	10
vatti	ivatti	5
atti	ti	9
tti	tivatti	4
ti	tti	8
i	ttivatti	3
ε	vatti	6

What we learn today?

- We learn that it is possible to replace suffix trees with *compressed suffix trees* that take 8.8 GB for the human genome.
- We learn that *backtracking* can be done using *compressed suffix arrays* requiring only 2.1 GB for the human genome.
- We learn that *discovering* interesting motif seeds from the human genome takes 40 hours and requires 9.3 GB space.

Recent suffix array constructions

- Manber&Myers (1990): O(|T|log|T|)
- linear time via suffix tree
- January / June 2003: direct linear time construction of suffix array

  - Kim, Sim, Park, Park (CPM03) - Kärkkäinen & Sanders (ICALP03) - Ko & Aluru (CPM03)

Kärkkäinen-Sanders algorithm

1. Construct the suffix array of the suffixes starting at positions i mod 3 ≠ 0. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.
2. Construct the suffix array of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

Notation

- string $T = T[0,n) = t_0 t_1 \ldots t_{n-1}$
- suffix $S_i = T[i,0) = t_i t_{i+1} \ldots t_{n-1}$
- for C \subset [0,n]: $S_C = \{S_i \mid i \text{ in } C\}$

- *suffix array* SA[0,n] of T is a permutation of [0,n] satisfying $S_{SA[0]} < S_{SA[1]} < \ldots < S_{SA[n]}$

Running example

0 1 2 3 4 5 6 7 8 9 10 11

- T[0,n) = y a b b a d a b b a d o 0 0 …
- SA = (12,1,6,4,9,3,8,2,7,5,10,11,0)

Step 0: Construct a sample

- for k = 0,1,2

  $Bk = \{i \in [0,n] \mid i \bmod 3 = k\}$

- C = B1 U B2 *sample positions*
- S_C *sample suffixes*

- Example: B1 = {1,4,7,10}, B2 = {2,5,8,11}, C = {1,4,7,10,2,5,8,11}

Step 1: Sort sample suffixes

- for k = 1,2, construct

  $Rk = [t_k t_{k+1} t_{k+2}] [t_{k+3} t_{k+4} t_{k+5}] \ldots [t_{maxBk} t_{maxBk+1} t_{maxBk+2}]$

  R = R1 ^ R2 concatenation of R1 and R2

  Suffixes of R correspond to S_C: suffix $[t_i t_{i+1} t_{i+2}] \ldots$ corresponds to S_i ; correspondence is order preserving.

  Sort the suffixes of R: radix sort the characters and rename with ranks to obtain R´. If all characters different, their order directly gives the order of suffixes. Otherwise, sort the suffixes of R´ using Kärkkäinen-Sanders. Note: |R´| = 2n/3.

Step 1 (cont.)

- once the sample suffixes are sorted, assign a rank to each: rank(S_i) = the rank of S_i in S_C; rank(S_{n+1}) = rank (S_{n+2}) = 0
- Example:

  R = [abb][ada][bba][do0][bba][dab][bad][o00]

  R´ = (1,2,4,6,4,5,3,7)

  $SA_{R'}$ = (8,0,1,6,4,2,5,3,7) rank

(S_i) - 1 4 - 2 6 - 5 3 – 7 8 – 0 0

Step 2: Sort nonsample suffixes

- for each non-sample $S_i \in S_{B0}$ (note that rank(S_{i+1}) is always defined for i ϵ B0): $S_i \leq S_j \leftrightarrow (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1}))$
- radix sort the pairs $(t_i, \text{rank}(S_{i+1}))$.

- Example: $S_{12} < S_6 < S_9 < S_3 < S_0$ because (0,0) < (a,5) < (a,7) < (b,2) < (y,1)

Step 3: Merge

- merge the two sorted sets of suffixes using a standard comparison-based merging:
- to compare $S_i \in S_C$ with $S_j \in S_{B0}$, distinguish two cases:

- $i \in B1$: $S_i \leq S_j \leftrightarrow (t_i, \text{rank}(S_{i+1})) \leq (t_j, \text{rank}(S_{j+1}))$
- $i \in B2$: $S_i \leq S_j \leftrightarrow (t_i, t_{i+1}, \text{rank}(S_{i+2})) \leq (t_j, t_{j+1}, \text{rank}(S_{j+2}))$

- note that the ranks are defined in all cases!
- $S_1 < S_6$ as $(a,4) < (a,5)$ and $S_3 < S_8$ as $(b,a,6) < (b,a,7)$

Running time O(n)

- excluding the recursive call, everything can be done in linear time
- the recursion is on a string of length 2n/3
- thus the time is given by recurrence

  $T(n) = T(2n/3) + O(n)$

- hence $T(n) = O(n)$

Implementation

- about 50 lines of C++
- code available e.g. via Juha Kärkkäinen's home page

LCP table

- Longest Common Prefix of successive elements of suffix array:
- LCP[i] = length of the longest common prefix of suffixes $S_{SA[i]}$ and $S_{SA[i+1]}$
- build inverse array SA^{-1} from SA in linear time
- then LCP table from SA^{-1} in linear time (Kasai et al, CPM2001)

- Oxford English Disctionary http://www.oed.com/
- Example - Word of the Day , Fourth
  http://biit.cs.ut.ee/~vilo/edu/2005-06/Text_Algorithms/L7_SuffixTrees/wotd_fourth.html
  http://www.oed.com/cgi/display/wotd
- PAT index - by **Gaston Gonnet** (ta on samuti Maple tarkvara üks loojatest ning hiljem molekulaarbioloogia tarkvarapaketi väljatöötajaid)
- PAT index is essentially a suffix array. To save space, indexed only from first character of every word
- XML-tagging (or SGML, at that time!) also indexed
- To mark certain fields of XML, the bit vectors were used.
- Main concern - improve the speed of search on the CD - minimize random accesses.
- For slow medium even 15-20 accesses is too slow...
- **G. H. Gonnet, R. A. Baeza-Yates**, and T. Snider, Lexicographical indices for text: Inverted files vs. PAT trees, Technical Report OED-91-01, Centre for the New OED, University of Waterloo, 1991.

Suffix tree vs suffix array

- **suffix tree ⇔ suffix array + LCP table**

1. Suffix tree
2. Suffix array
3. **Some applications**
4. Finding motifs

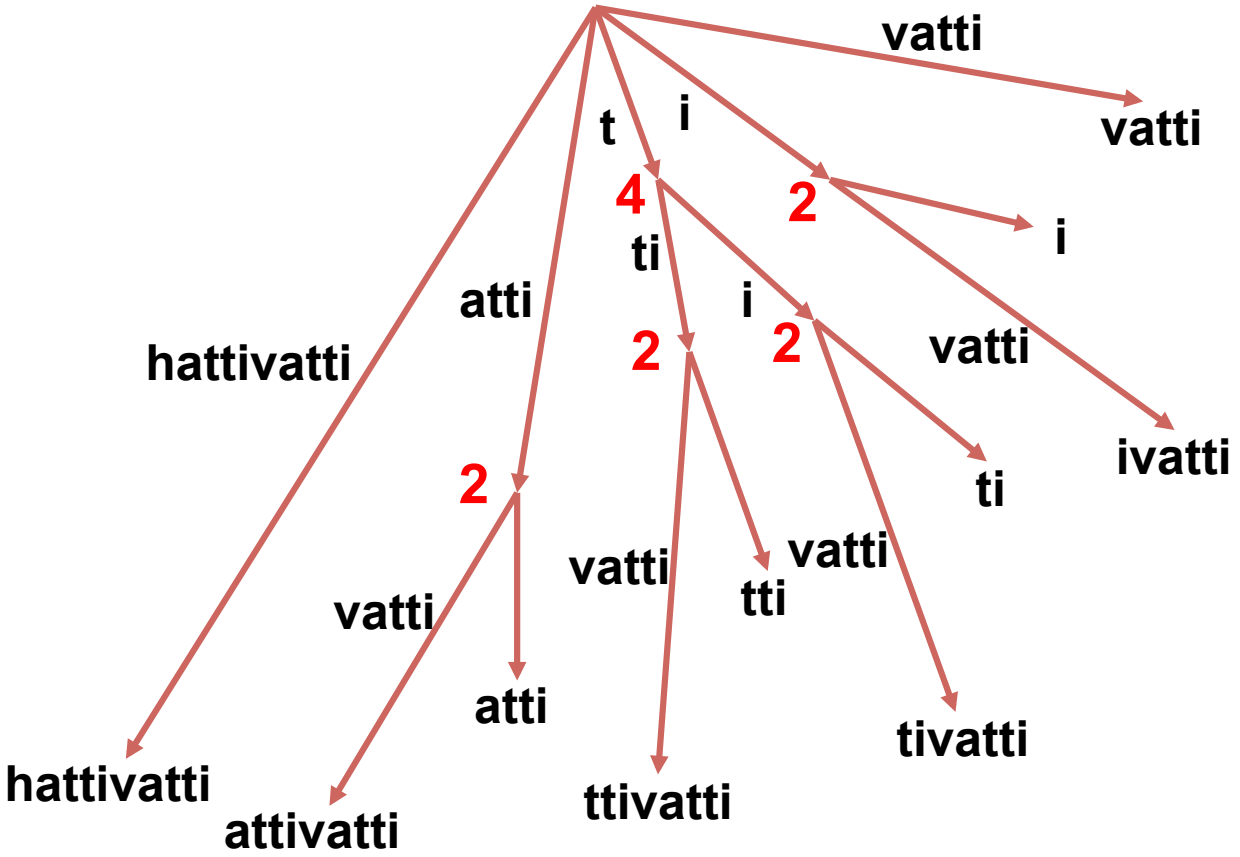
Substring motifs of string T

- string T = $t_1 \ldots t_n$ in alphabet A.
- Problem: what are the frequently occurring (ungapped) substrings of T? Longest substring that occurs at least q times?
- Thm: Suffix tree *Tree(T)* gives complete occurrence counts of all substring motifs of T in $O(n)$ time (although T may have $O(n^2)$ substrings!)

Counting the substring motifs

- internal nodes of Tree(T) $\leftrightarrow$ repeating substrings of T
- number of leaves of the subtree of a node for string P = number of occurrences of P in T

Substring motifs of hattivatti



Counts for the *O(n)* maximal motifs shown

Finding repeats in DNA

- human chromosome 3
- the first 48 999 930 bases
- 31 min cpu time (8 processors, 4 GB)

- Human genome: $3x10^9$ bases
- Tree(HumanGenome) feasible

Longest repeat?

Occurrences at: 28395980, 28401554r Length: 2559

ttagggtacatgtgcacaacgtgcaggtttgttacatatgtatacacgtgccatgatggtgtgctgcacccattaactcgtcatttagcgttaggtatatctccgaat
gctatccctccccctcccccacccacaacagtccccggtgtgtgatgttccccttcctgtgtccatgtgttctcattgttcaattcccacctatgagtgagaa
catgcggtgtttggtttttgtccttgcgaaagtttgctgagaatgatggtttccagcttcatccatatccctacaaaggacatgaactcatcatttttttatggctgcat
agtattccatggtgtatatgtgccacattttcttaacccagtctacccttgttggacatctgggttggttccaagtctttgctattgtgaatagtgccgcaataaacat
acgtgtgcatgtgtctttatagcagcatgatttataatcctttgggtatatacccagtaatgggatggctgggtcaaatggtatttctagttctagatccctgaggaat
caccacactgacttccacaatggttgaactagtttacagtcccagcaacagttcctatttctccacatcctctccagcacctgttgtttcctgacttttaatgatcgc
cattctaactggtgtgagatggtatctcattgtggttttgatttgcatttctctgatggccagtgatgatgagcatttttttcatgtgttttttggctgcataaatgtcttctttt
gagaagtgtctgttcatatccttcgcccacttttgatggggttgtttgtttttttcttgtaaatttgttggagttcattgtagattctgggtattagccctttgtcagatgagt
aggttgcaaaaattttctcccattctgtaggttgcctgttcactctgatggtggtttcttctgctgtgcagaagctctttagtttaattagatcccatttgtcaattttggct
tttgttgccatagcttttggtgttttagacatgaagtccttgcccatgcctatgtcctgaatggtattgcctaggttttcttctagggtttttatggttttaggtctaacatg
taagtctttaatccatcttgaattaattataaggtgtatattataaggtgtaattataaggtgtataattatatattaattataaggtgtatattaattataaggtgtaagga
agggatccagtttcagctttctacatatggctagccagttttccctgcaccatttattaaatagggaatcctttccccattgcttgtttttgtcaggtttgtcaaagatc
agatagttgtagatatgcggcattatttctgagggctctgttctgttccattggtctatatctctgttttggtaccagtaccatgctgttttggttactgtagccttgtagt
atagtttgaagtcaggtagcgtgatggttccagctttgttcttttggcttaggattgacttggcaatgtgggctctttttggttccatatgaactttaaagtagttttttc
caattctgtgaagaaattcattggtagcttgatggggatggcattgaatctataaattaccctgggcagtatggccattttcacaatattgaatcttcctacccatga
gcgtgtactgttcttccatttgtttgtatcctcttttatttcattgagcagtggtttgtagttctccttgaagaggtccttcacatcccttgtaagttggattcctaggtatttt
attctctttgaagcaattgtgaatgggagttcactcatgatttgactctctgtttgtctgttattggtgtataagaatgcttgtgatttttgcacattgattttgtatcctgag
actttgctgaagttgcttatcagcttaaggagattttgggctgagacgatggggttttctagatatacaatcatgtcatctgcaaacagggacaatttgacttcctct
tttcctaattgaataccgttatttccctctcctgcctgattgccctggccagaacttccaacactatgttgaataggagtggtgagagagggcatccctgtcttgt
gccagttttcaaagggaatgcttccagtttttgtccattcagtatgatattggctgtgggtttgtcatagatagctcttattattttgagatacatcccatcaatacctaa
tttattgagagtttttagcatgaagagttcttgaattttgtcaaaggccttttctgcatcttttgagataatcatgtggtttctgtctttggttctgtttatatgctggagtac
gtttattgattttcgtatgttgaaccagccttgcatcccagggatgaagcccacttgatcatggtggataagctttttgatgtgctgctggattcggtttgccagtattt
tattgaggatttctgcatcgatgttcatcaaggatattggtctaaaattctctttttttgttgtgtctctgtcaggctttggtatcaggatgatgctggcctcataaaatga
gttagg

Ten occurrences?

tttttttttttttgagacggagtctcgctctgtcgcccaggctggagtgcagtggcgggat
ctcggctcactgcaagctccgcctcccgggttcacgccattctcctgcctcagcctcc
caagtagctgggactacaggcgcccgccactacgcccggctaatttttgtatttttagt
agagacggggtttcaccgttttagccgggatggtctcgatctcctgacctcgtgatccg
cccgcctcggcctcccaaagtgctgggattacaggcgt

Length: 277

Occurrences at: 10130003, 11421803, 18695837, 26652515, 42971130, 47398125
In the reversed complement at: 17858493, 41463059, 42431718, 42580925

Using suffix trees: plagiarism

- find longest common substring of strings X and Y
- build Tree(X$Y) and find the deepest node which has a leaf pointing to X and another pointing to Y

Using suffix trees: approximate matching

- edit distance: insertions, deletions, changes
- STOCKHOLM vs TUKHOLMA

String distance/similarity functions

STOCKHOLM vs TUKHOLMA

STOCKHOLM_

_TU_

KHOLMA

=> 2 deletions, 1 insertion, 1 change

Approximate string matching

- A: S T O C K H O L M

B: T U K H O L M A

- minimum number of 'mutation' steps:

  a -> b a -> ϵ ϵ -> b …

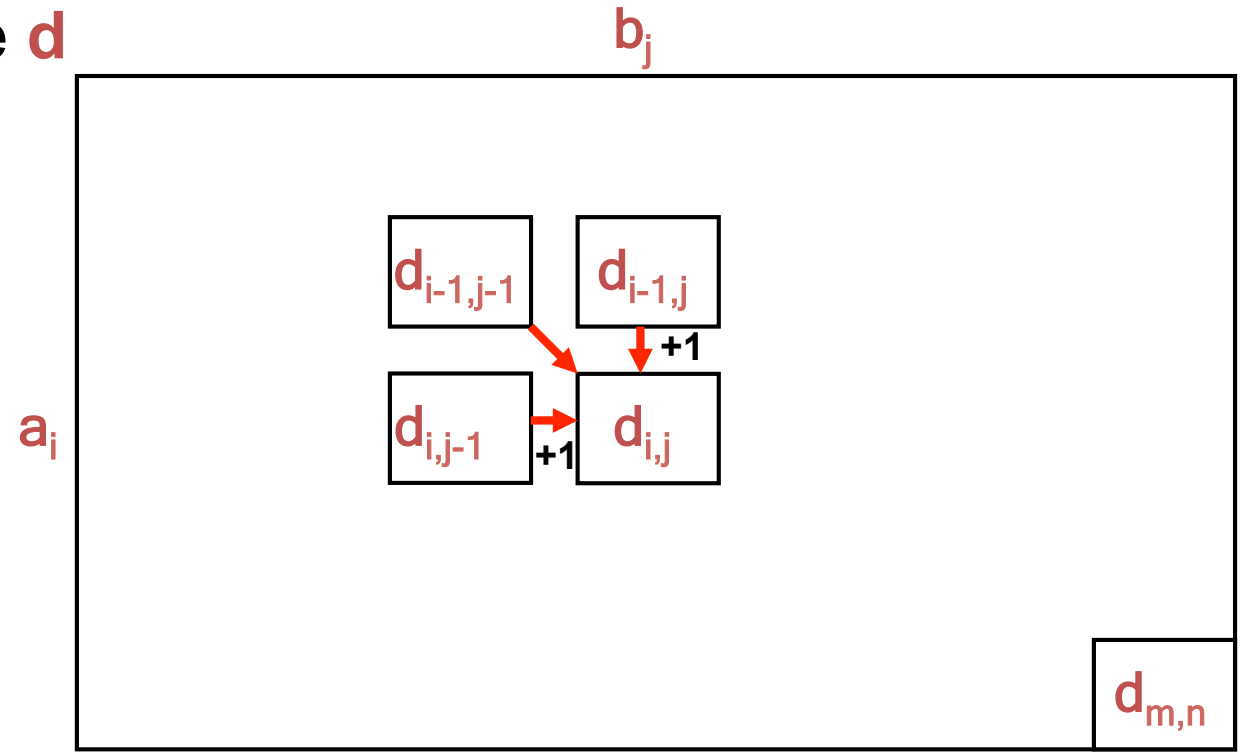
- $d_{ID}(A,B) = 5$ $d_{Levenshtein}(A,B) = 4$
- mutation costs => probabilistic modeling
- evaluation by dynamic programming => alignment

Dynamic programming

$d_{i,j}$ = min(if $a_i = b_j$ then $d_{i-1,j-1}$ else ∞,
$d_{i-1,j} + 1$,
$d_{i,j-1} + 1$)

= distance between i-prefix of A and j-prefix of B
(substitution excluded)

mxn table **d**



$$d_{i,j} = \min(\text{if } a_i = b_j \text{ then } d_{i-1,j-1} \text{ else } \infty,\ d_{i-1,j} + 1,\ d_{i,j-1} + 1)$$

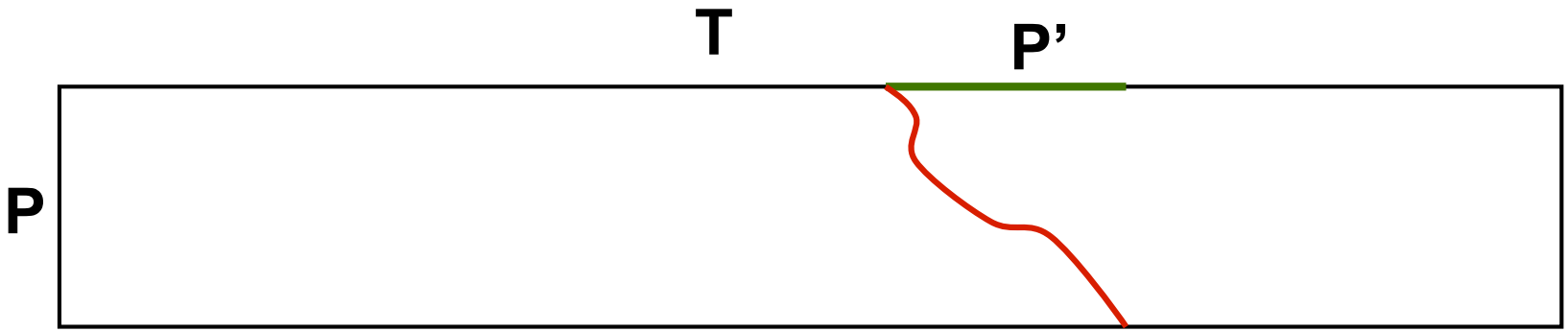
A\B		s	t	o	c	k	h	o	l	m
	0	1	2	3	4	5	6	7	8	9
t	1	2	1	2	3	4	5	6	7	8
u	2	3	2	3	4	5	6	7	8	9
k	3	4	3	4	5	4	5	6	7	8
h	4	5	4	5	6	5	4	5	6	7
o	5	6	5	4	5	6	5	4	5	6
l	6	7	6	5	6	7	6	5	4	5
m	7	8	7	6	7	8	7	6	5	4
a	8	9	8	7	8	9	8	7	6	5

optimal alignment by trace-back

$d_{ID}(A,B)$

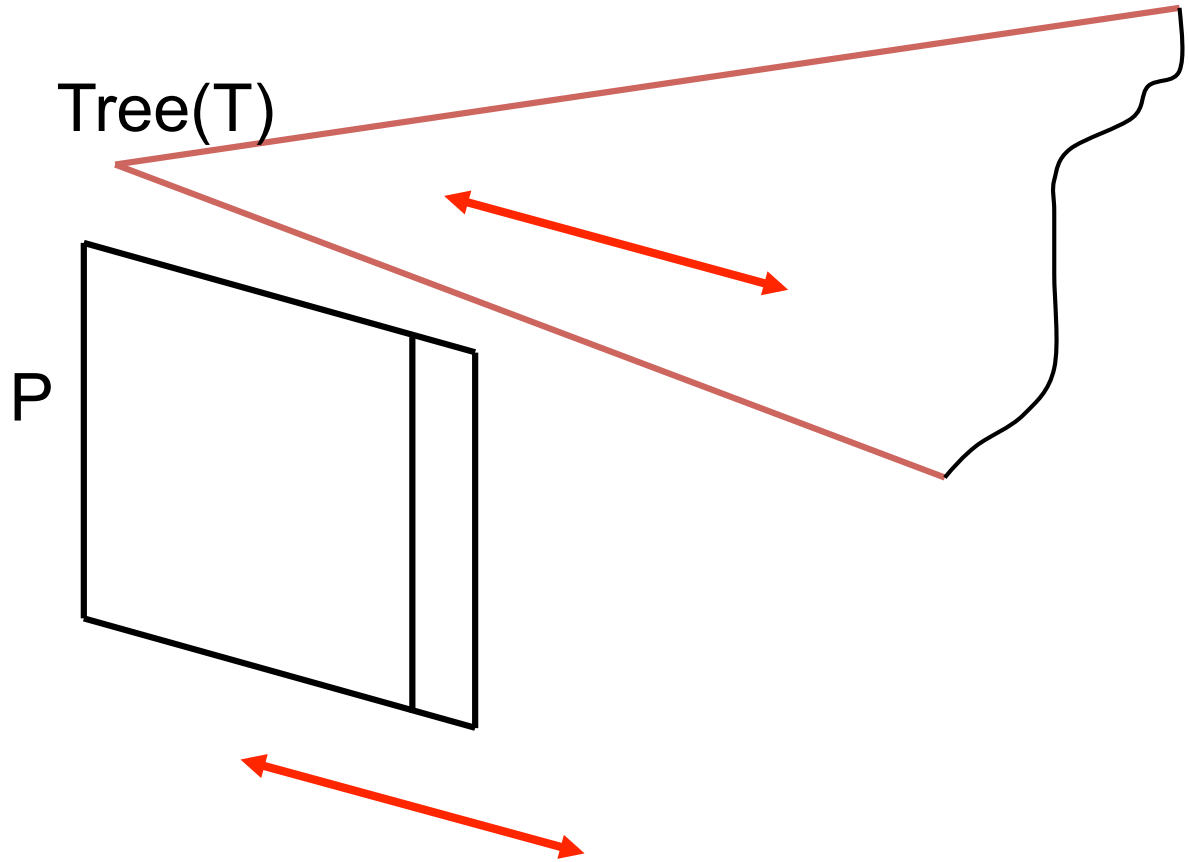
Search problem

- find approximate occurrences of pattern P in text T: substrings P' of T such that d(P,P') small
- dyn progr with small modification: O(mn)
- lots of (practical) improvement tricks



Index for approximate searching?

- dynamic programming: P x Tree(T) with backtracking



1. Suffix tree
2. Suffix array
3. Some applications
4. **Finding motifs**

Gapped motifs

a##bc

Gapped motifs

a##bc a##bc

ababbbccbcaaabca

Gapped motifs of T

- gapped pattern: P in $(A \cup \{\#\})^*$

- gap symbol # matches any symbol in A
- aa##bb#b
- L(P) = occurrence locations of P in T

- P is called a motif of T if $|L(P)| > 1$ and a motif with quorum q if $|L(P)| \geq q$.
- Problem: find occurrence count $|L(P)|$ for all gapped motifs P of T

- $a^n b a^n$ has exponentially many motifs!

Maximal motifs

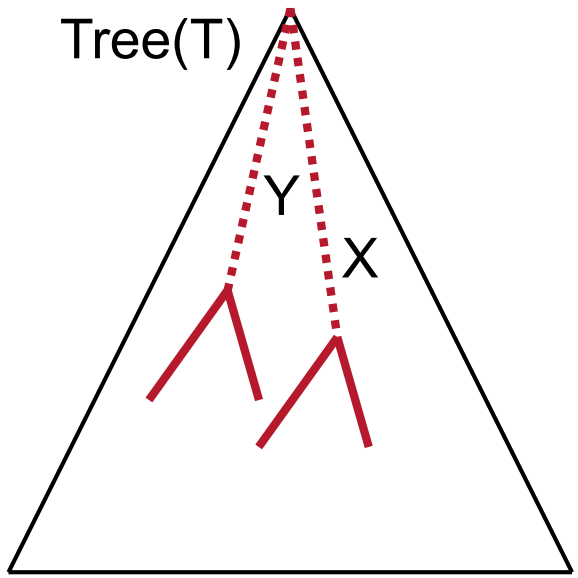
- motif P´ is the maximal version of motif P if P ´ has the largest possible number of non-# symbols among motifs that have the same set of occurrence locations as P

- every motif has unique maximal version

- unfortunately still exponential number of different maximal motifs

Blocks of maximal motifs

- aaa##b##ba has blocks aaa, b, ba
- Lemma: Maximal 1-block motifs ↔ (branching) nodes of Tree(S)
- Thm: Each block of a maximal motif of T is a maximal substring motif of T. Hence there are O(n) different strings that can be used as a block of a maximal motif.
- => There are $O(n^{2k-1})$ different maximal motifs with k blocks [$O(n^{2k})$ unrestricted motifs].



Two-block maximal pattern: X###Y

Counting 2-block maximal motifs

- Thm: The occurrence counts for all maximal motifs with two blocks can be found in (optimal) time $O(n^3)$.
- c.f., Arimura et al (2000), Apostolico et al (2004), …

Algorithm (very simple)

2-block motif *(X,d,Y)*

for each maximal substring motif X

  for each distance d = 1,2, …

    mark the leaves of Tree(T) that correspond to locations L(X) + d

    for each maximal substring motif Y,
      find the number h(Y) of marked leaves in its subtree in Tree(T)

    the occurrence count of motif (X,d,Y) is h(Y)

Algorithm (very simple)

2-block motif *(X,d,Y)*

for each maximal substring motif X — ***O(n)***

  for each distance d = 1,2, … — ***O(n)***

    mark the leaves of Tree(T) that correspond to locations L(X) + d — ***O(n)***

    for each maximal substring motif Y,
      find the number h(Y) of marked leaves in its subtree in Tree(T)

    the occurrence count of motif (X,d,Y) is h(Y)

Counting 2-block maximal motifs (cont)

- Thm: The occurrence counts for all maximal motifs with two blocks can be found in (optimal) time $O(n^3)$.

- flexible gaps:
  x*y * = gap of any length

- Thm: The occurrence counts for all maximal motifs with two blocks and one flexible gap can be found in (optimal) time $O(n^2)$.
- k-block case?

Conclusion

- suffix structures provide very efficient algorithmic tools for finding and learning potentially interesting patterns in strings