The background features a stylized illustration of a book spine on the left, with a grid of text filling the rest of the slide. The text in the grid is a repeating pattern of 'ALGORITHMS' and '\$APPLIED', with some letters highlighted in red. The main title is overlaid on this background.

6 Text Indexing – Searching whole genomes

9 March 2020

Sebastian Wild

6 Text Indexing

- 6.1 Motivation
- 6.2 Suffix Trees
- 6.3 Applications
- 6.4 Longest Common Extensions
- 6.5 Suffix Arrays
- 6.6 The LCP Array

6.1 Motivation

Text indexing

- ▶ *Text indexing* (also: *offline text search*):

- ▶ case of string matching: find $P[0..m-1]$ in $T[0..n-1]$

- ▶ but with fixed text \rightsquigarrow preprocess T (instead of P)

- \rightsquigarrow expect many queries P , answer them without looking at all of T

- \rightsquigarrow essentially a data structuring problem: “building an *index* of T ”

Latin: “one who points out”

- ▶ application areas

- ▶ web search engines

- ▶ online dictionaries

- ▶ online encyclopedia

- ▶ DNA/RNA data bases

- ▶ ... searching in any collection of text documents (that grows only moderately)

Inverted indices

same as "indexes"

- ▶ original indices in books: list of (key) words \mapsto page numbers where they occur
- ▶ assumption: searches are only for **whole** (key) **words** \parallel
- \rightsquigarrow often reasonable for natural language text

Inverted indices

same as "indexes"

- ▶ original indices in books: list of (key) words \mapsto page numbers where they occur
- ▶ assumption: searches are only for **whole** (key) **words**
- \rightsquigarrow often reasonable for natural language text

Inverted index:

- ▶ collect all words in T
 - ▶ can be as simple as splitting T at whitespace
 - ▶ actual implementations typically support *stemming* of words
goes \rightarrow go, cats \rightarrow cat
 - ▶ store mapping from words to a list of occurrences \rightsquigarrow how? not here ——— BST
- $go \mapsto \{5, 10, 20\}$
 $cat \mapsto \{4, 21\}$

Clicker Question



Do you know what a *trie* is?

- ☐ A A what? No!
- ☐ B I have heard the term, but don't quite remember.
- ☐ C I remember hearing about it in a module.
- ☐ D Sure.

pingo.upb.de/622222

Tries

$\{aa, a\}$



- ▶ efficient dictionary data structure for strings
- ▶ name from retrieval, but pronounced “try”
- ▶ tree based on symbol comparisons
- ▶ **Assumption:** stored strings are *prefix-free* (no string is a prefix of another)
 - ▶ strings of same length ✓
 - ▶ strings have “end-of-string” marker \$ ✓

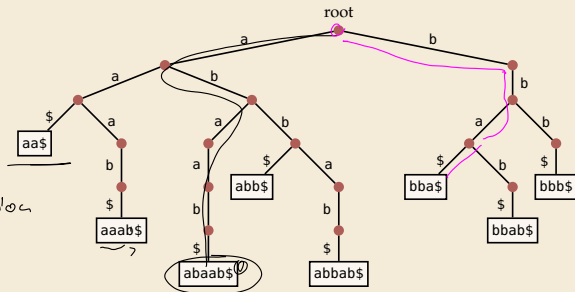
some character $\notin \Sigma$

Example:

$\{aa$, aaab$, abaab$, abb$,
abbab$, bba$, bbab$, bbb$\}$

o construction: top-down
independent of order of insertion

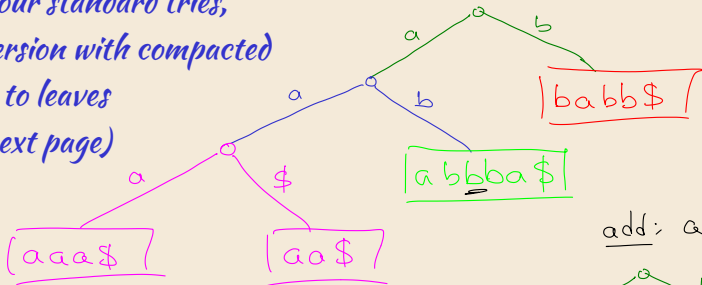
o query: (get) ex: bba\$



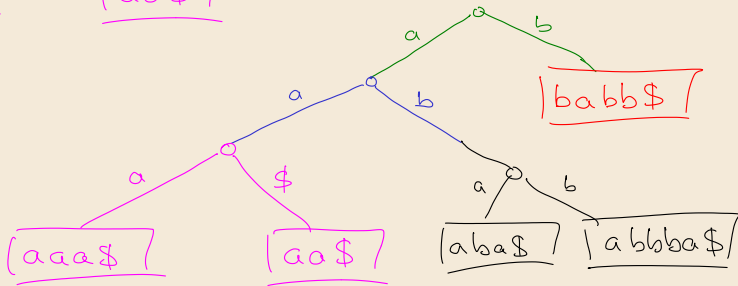
Construction

{aaa\$, babbb\$, aa\$, abbbba\$}

NOT our standard tries,
but version with compacted
paths to leaves
(see next page)



add: a ba\$

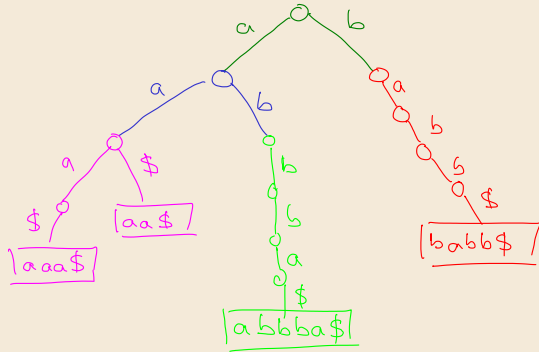
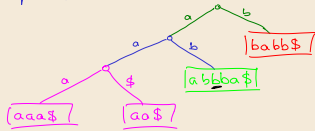


Trie construction (correct version)

{aaa\$, babb\$, aa\$, abbbba\$}

standard trie

trie with compacted
paths to leaves



Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{A, \dots, Z\}$.
Each stored string consists of m characters.

We now search for a query string Q with $|Q| = q$.

How many **nodes** in the trie are **visited** during this **query**?



A $\Theta(\log n)$

F $\Theta(\log m)$

B $\Theta(\log(nm))$

G $\Theta(q)$

C $\Theta(m \cdot \log n)$

H $\Theta(\log q)$

D $\Theta(m + \log n)$

I $\Theta(q \cdot \log n)$

E $\Theta(m)$

J $\Theta(q + \log n)$

pingo.upb.de/622222

Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{A, \dots, Z\}$.
Each stored string consists of m characters.

We now search for a query string Q with $|Q| = q$. ^{successful}
How many nodes in the trie are **visited** during this **query**?



A ~~$\Theta(\log n)$~~

F ~~$\Theta(\log m)$~~

B ~~$\Theta(\log(nm))$~~

G $\Theta(q)$ ✓

C ~~$\Theta(m \log n)$~~

H ~~$\Theta(\log q)$~~

D ~~$\Theta(m + \log n)$~~

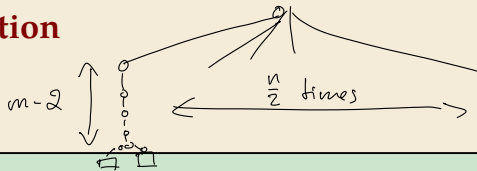
I ~~$\Theta(q \log n)$~~

E ~~$\Theta(m)$~~

J ~~$\Theta(q + \log n)$~~

pingo.upb.de/622222

Clicker Question



$$\begin{aligned} S_1 &= \$_1 \text{ a a a a a o o c c b} \\ S_2 &= \$_1 \text{ c a a o o c c a} \\ S_3 &= \$_2 \\ S_4 &= \$_2 \dots \end{aligned}$$

Suppose we have a trie that stores n strings over $\Sigma = \{A, \dots, Z\}$. Each stored string consists of m characters.

How many **nodes** does the trie have **in total** in the worst case?



A $\Theta(n)$

B $\Theta(n + m)$

C $\Theta(n \cdot m)$

D $\Theta(n \log m)$

E $\Theta(m)$

F $\Theta(m \log n)$

pingo.upb.de/622222

Clicker Question



Suppose we have a trie that stores n strings over $\Sigma = \{A, \dots, Z\}$. Each stored string consists of m characters.

How many **nodes** does the trie have **in total** *in the worst case*?

A ~~$\Theta(n)$~~

B ~~$\Theta(n + m)$~~

C $\Theta(n \cdot m)$ ✓

D ~~$\Theta(n \log m)$~~

E ~~$\Theta(m)$~~

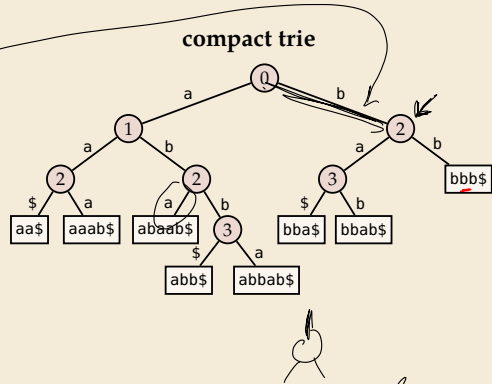
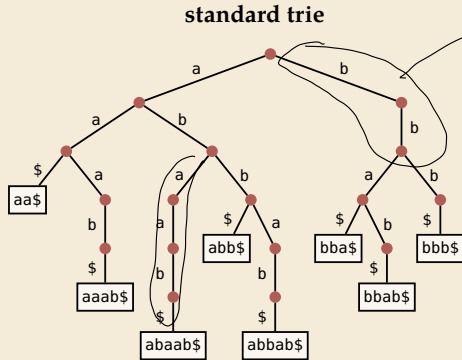
F ~~$\Theta(m \log n)$~~

pingo.upb.de/622222

Compact tries

- ▶ compress paths of unary nodes into single edge
- ▶ nodes store index of next character

o \triangle get needs extra check bab\$





↪ searching slightly trickier, but same time complexity as in trie


- ▶ all nodes ≥ 2 children \rightsquigarrow #nodes \leq #leaves = #strings \rightsquigarrow linear space

$\Theta(n)$ not $\Theta(mn)$

Tries as inverted index

 simple

 fast lookup

 cannot handle more general queries:

- ▶ search part of a word
- ▶ search phrase (sequence of words)

Tries as inverted index

👍 simple

👍 fast lookup

👎 cannot handle more general queries:

- ▶ search part of a word
- ▶ search phrase (sequence of words)

👎 what if the 'text' does not even have words to begin with?!

- ▶ biological sequences

```
ACAAGATGCCATTGTCCCCGGCCTCCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCCCTGGAGGGTGGCCCCACCGGC  
CGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGCCTCCTGACTTTCCTCGCTTGGTGGTTTGAGTGGACCTCCAGGC  
CAGTGCCGGGCCCCCTCATAGGAGAGGAAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGACCCCCCAGCAATCCGCGCGCCGGGACAGAA  
TGCCCTGCAGGAACCTTCTTCTGGAAGACCTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAGTTTAATTACAGACCTGAA
```

- ▶ binary streams

```
00000010101001111010111000001111100011111011111001101101000011100010011011110000010001101010  
0110110000110101101000000010000000011101011000001000011110101110110010001100101101110111111  
110001010001011001010000001110101010011000000001101100001100111110000101 010101110111000011  
10101110010010101010100000111110100110000001111001101010000000100100100000101100011000110111
```

~> need new ideas

6.2 Suffix Trees

Suffix trees – A ‘magic’ data structure

$S[i..j]$

Appetizer: Longest common substring problem

- ▶ Given: strings S_1, \dots, S_k **Example:** $S_1 = \text{superiorcaliforniallives}$, $S_2 = \text{sealiver}$
- ▶ Goal: find the longest substring that occurs in all k strings

Suffix trees – A ‘magic’ data structure

Appetizer: Longest common substring problem

► Given: strings S_1, \dots, S_k

Example: $S_1 = \text{superiorcalifornialives}$, $S_2 = \text{sealiver}$

► Goal: find the longest substring that occurs in all k strings

↪ alive ✓



Can we do this in time $\Theta(|S_1| + \dots + |S_k|)$? How??

Suffix trees – A ‘magic’ data structure

Appetizer: Longest common substring problem

► Given: strings S_1, \dots, S_k

Example: $S_1 = \text{superiorcalifornialives}$, $S_2 = \text{sealiver}$

► Goal: find the longest substring that occurs in all k strings

↪ alive



Can we do this in time $O(|S_1| + \dots + |S_k|)$? How??

Enter: *suffix trees*

- versatile data structure for index with full-text search
- linear time (for construction) and linear space
- allows efficient solutions for many advanced string problems



“Although the longest common substring problem looks trivial now, given our knowledge of suffix trees, it is very interesting to note that in 1970 Don Knuth conjectured that a linear-time algorithm for this problem would be impossible.”

[Gusfield: Algorithms on Strings, Trees, and Sequences (1997)]

Suffix trees – Definition

- ▶ suffix tree \mathcal{T} for text $T = T[0..n-1]$ = compact trie of all suffixes of T \$ (set $T[n] := \$$)

$n+1$ suffixes

Suffix trees – Definition

- suffix tree \mathcal{T} for text $T = T[0..n-1]$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)

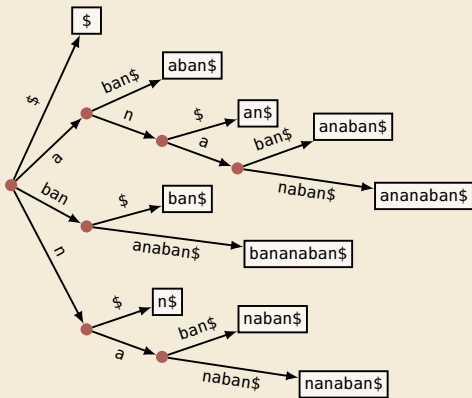
Example:

$T = \text{bananaban\$}$

suffixes: { $\text{bananaban\$}$, $\text{ananaban\$}$, $\text{nanaban\$}$,
 $\text{anaban\$}$, $\text{naban\$}$, $\text{aban\$}$, $\text{ban\$}$, $\text{an\$}$, $\text{n\$}$, $\text{\$}$ }

0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	\$

$T =$



Suffix trees – Definition

- ▶ suffix tree \mathcal{T} for text $T = T[0..n-1]$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)
- ▶ except: in leaves, store *start index* (instead of actual string)

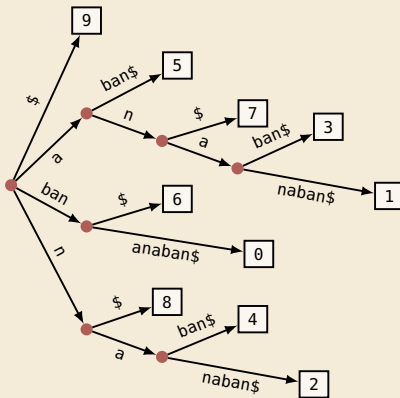
Example:

$T = \text{bananaban\$}$

suffixes: { $\text{bananaban\$}$, $\text{ananaban\$}$, $\text{nanaban\$}$,
 $\text{anaban\$}$, $\text{naban\$}$, $\text{aban\$}$, $\text{ban\$}$, $\text{an\$}$, $\text{n\$}$, $\text{\$}$ }

0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	\$

$T =$



Suffix trees – Definition

- ▶ suffix tree \mathcal{T} for text $T = T[0..n-1]$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)
- ▶ except: in leaves, store *start index* (instead of actual string)

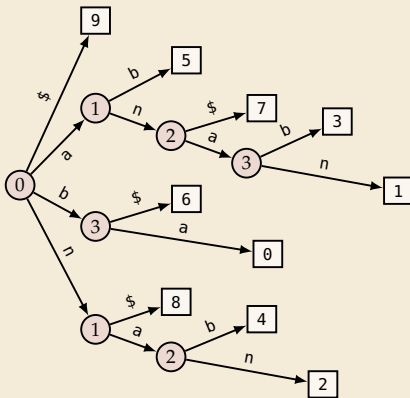
Example:

$T = \text{bananaban\$}$

suffixes: { $\text{bananaban\$}$, $\text{ananaban\$}$, $\text{nanaban\$}$,
 $\text{anaban\$}$, $\text{naban\$}$, $\text{aban\$}$, $\text{ban\$}$, $\text{an\$}$, $\text{n\$}$, $\text{\$}$ }

0	1	2	3	4	5	6	7	8	9
b	a	n	a	n	a	b	a	n	\$

- ▶ also: edge labels like in compact trie
- ▶ (more readable form on slides to explain algorithms)



Suffix trees – Construction

- ▶ $T[0..n-1]$ has $n+1$ suffixes (starting at character $i \in [0..n]$)
- ▶ We can build the suffix tree by inserting each suffix of T into a compressed trie. But that takes time $\Theta(n^2)$. \rightsquigarrow not interesting!



Suffix trees – Construction

- ▶ $T[0..n-1]$ has $n+1$ suffixes (starting at character $i \in [0..n]$)
- ▶ We can build the suffix tree by inserting each suffix of T into a compressed trie. But that takes time $\Theta(n^2)$. \rightsquigarrow not interesting!



same order of growth as reading the text!

Amazing result: Can construct the suffix tree of T in $\underline{\Theta(n)}$ time!

- ▶ algorithms are a bit tricky to understand
- ▶ but were a theoretical breakthrough
- ▶ and they are efficient in practice (and heavily used)!

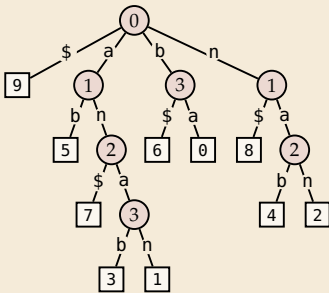
\rightsquigarrow for now, take linear-time construction for granted. What can we do with them?

6.3 Applications

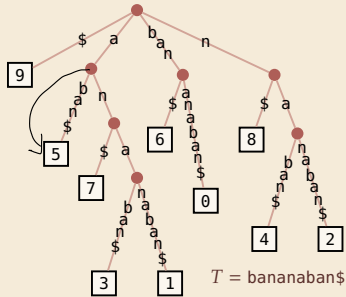
Applications of suffix trees

- In this section, always assume suffix tree \mathcal{T} for T given.

Recall: \mathcal{T} stored like this:



but think about this:



- Moreover: assume internal nodes store pointer to leftmost leaf in subtree.
- Notation: $T_i = T[i..n]$ (including \$)
 \downarrow

Application 1: Text Indexing / String Matching

- ▶ P occurs in $T \iff P$ is a prefix of a suffix of T
- ▶ we have all suffixes in \mathcal{T} !

Application 1: Text Indexing / String Matching

► P occurs in $T \iff P$ is a prefix of a suffix of T

► we have all suffixes in \mathcal{T} !

↪ (try to) follow path with label P , until

1. **we get stuck**

at internal node (no node with next character of P)
or inside edge (mismatch of next characters)

↪ P does not occur in T

2. **we run out of pattern**

reach end of P at internal node v or inside edge towards v

↪ P occurs at all leaves in subtree of v

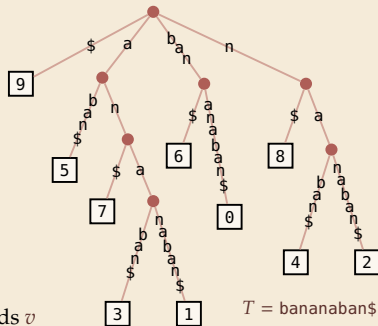
3. **we run out of tree**

we reach a leaf ℓ with part of P left ↪ P does not occur.



This cannot happen when testing edge labels since $\$ \notin \Sigma$, but needs check(s) in compact trie implementation!

► Finding first match (or NO_MATCH) takes $O(|P|)$ time!



Application 1: Text Indexing / String Matching

- P occurs in $T \iff P$ is a prefix of a suffix of T

- we have all suffixes in \mathcal{T} !

→ (try to) follow path with label P , until

1. we get stuck

Q_{NN} — at internal node (no node with next character of P)

- bb — or *inside edge* (mismatch of next characters)

$\rightsquigarrow P$ does not occur in T

- 2. we run out of pattern

reach end of P at internal node v or inside edge towards v

→ P occurs at all leaves in subtree of v

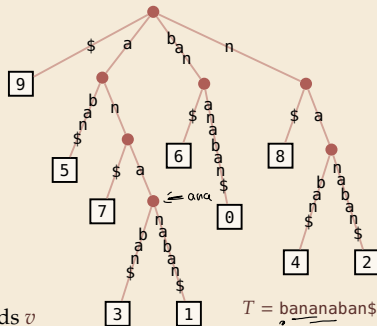
- ### 3. we run out of tree

we reach a leaf ℓ with part of P left \rightsquigarrow ~~P does not occur.~~



This cannot happen when testing edge labels since $\$ \notin \Sigma$, but needs check(s) in compact trie implementation!

- Finding ^{one} first match (or NO_MATCH) takes $O(|P|)$ time!



$T = \text{bananaban\$}$

Examples:

- ▶ $P = \text{ann}$
- ▶ $P = \text{ana}$
- ▶ $P = \text{briar}$

Application 2: Longest repeated substring

- **Goal:** Find longest substring $T[i..i + \ell)$ that occurs also at $j \neq i$: $T[j..j + \ell) = T[i..i + \ell)$.

e.g. for compression \rightsquigarrow Unit 7



How can we efficiently check *all possible substrings*?

Application 2: Longest repeated substring

- **Goal:** Find longest substring $T[i..i + \ell)$ that occurs also at $j \neq i$: $T[j..j + \ell) = T[i..i + \ell)$.

e.g. for compression \rightsquigarrow Unit 7



How can we efficiently check *all possible substrings*?



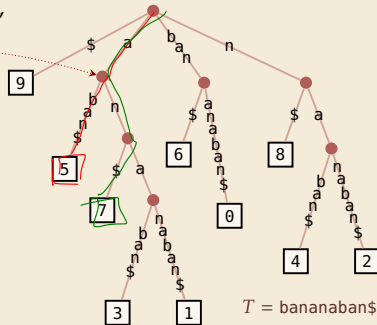
Repeated substrings = shared paths in *suffix tree*



- $T_5 = \text{aban\$}$ and $T_7 = \text{an\$}$ have longest common prefix 'a'

$\rightsquigarrow \exists$ internal node with path label 'a'

here single edge, can be longer path



Application 2: Longest repeated substring

- **Goal:** Find longest substring $T[i..i + \ell)$ that occurs also at $j \neq i$: $T[j..j + \ell) = T[i..i + \ell)$.

e.g. for compression \rightsquigarrow Unit 7



How can we efficiently check *all possible substrings*?



Repeated substrings = shared paths in *suffix tree*



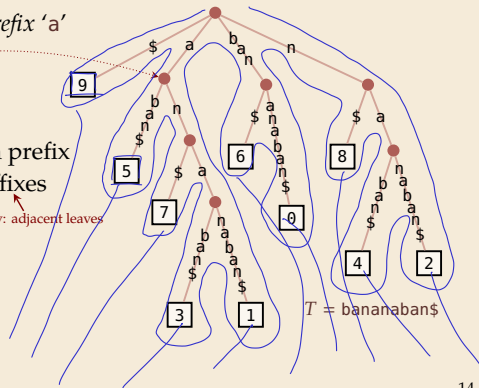
- $T_5 = \text{aban\$}$ and $T_7 = \text{an\$}$ have *longest common prefix* 'a'

$\rightsquigarrow \exists$ internal node with path label 'a'

here single edge, can be longer path

\rightsquigarrow longest repeated substring = longest common prefix (LCP) of two suffixes

actually: adjacent leaves



Application 2: Longest repeated substring

- **Goal:** Find longest substring $T[i..i + \ell)$ that occurs also at $j \neq i$: $T[j..j + \ell) = T[i..i + \ell)$.

e.g. for compression \rightsquigarrow Unit 7



How can we efficiently check *all possible substrings*?



Repeated substrings = shared paths in *suffix tree*



- $T_5 = \text{aban\$}$ and $T_7 = \text{an\$}$ have *longest common prefix* 'a'

$\rightsquigarrow \exists$ internal node with path label 'a'

here single edge, can be longer path

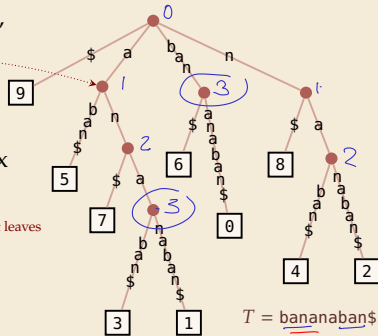
↪ longest repeated substring = longest common prefix (LCP) of two suffixes

actually: adjacent leaves

- Algorithm:

1. Compute string depth (=length of path label) of nodes
2. Find internal nodes with maximal string depth

- Both can be done in depth-first traversal $\rightsquigarrow \Theta(n)$ time



Generalized suffix trees

- ▶ longest *repeated* substring (of one string) feels very similar to longest common substring of several strings $T^{(1)}, \dots, T^{(k)}$ with $T^{(j)} \in \Sigma^{n_j}$
- ▶ can we solve that in the same way?
- ▶ could build the suffix tree for each $T^{(j)}$... but doesn't seem to help

Generalized suffix trees

- ▶ longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings $T^{(1)}, \dots, T^{(k)}$ with $T^{(j)} \in \Sigma^{n_j}$
 - ▶ can we solve that in the same way?
 - ▶ could build the suffix tree for each $T^{(j)}$... but doesn't seem to help
- ~> need a *single/joint* suffix tree for *several* texts

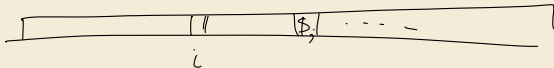
Generalized suffix trees

- ▶ longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings $T^{(1)}, \dots, T^{(k)}$ with $T^{(j)} \in \Sigma^{n_j}$
 - ▶ can we solve that in the same way?
 - ▶ could build the suffix tree for each $T^{(j)}$... but doesn't seem to help
- ↪ need a *single/joint* suffix tree for *several* texts

Enter: *generalized suffix tree*

- ▶ Define $T := \underline{T^{(1)}\$_1 T^{(2)}\$_2 \dots T^{(k)}\$_k}$ for k new end-of-word symbols
- ▶ Construct suffix tree \mathcal{T} for T

↪ $\$_j$ -edges always leads to leaves ↪ \exists leaf (j, i) for each suffix $T_i^{(j)} = T^{(j)}[i..n_j]$



Application 3: Longest common substring

- ▶ With that new idea, we can find longest common superstrings:
 1. Compute generalized suffix tree \mathcal{T} .
 2. Store with each node the *subset of strings* that contain its path label:
 - 2.1. Traverse \mathcal{T} bottom-up.
 - 2.2. For a leaf (j, i) , the subset is $\{j\}$.
 - 2.3. For an internal node, the subset is the union of its children.
 3. In top-down traversal, compute *string depths* of nodes. (as above)
 4. Report deepest node (by string depth) whose subset is $\{1, \dots, k\}$.

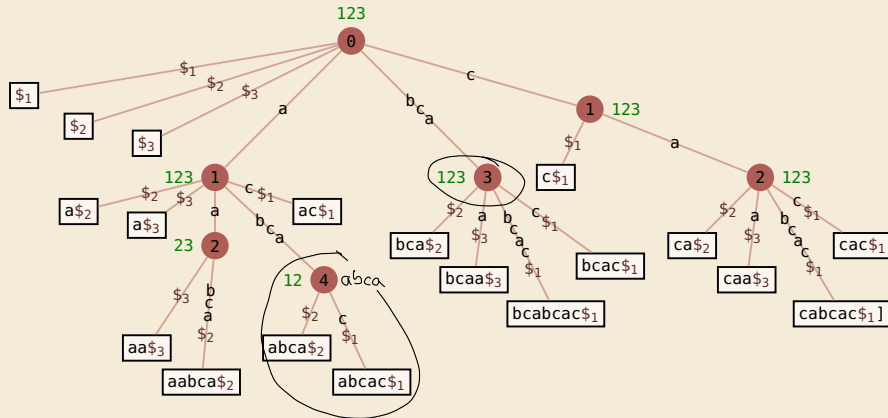
- ▶ Each step takes time $\Theta(n)$ for $n = n_1 + \dots + n_k$ the total length of all texts.

“Although the longest common substring problem looks trivial now, given our knowledge of suffix trees, it is very interesting to note that in 1970 Don Knuth conjectured that a linear-time algorithm for this problem would be impossible.” [Gusfield: Algorithms on Strings, Trees, and Sequences (1997)]

Longest common substring – Example

$T^{(1)} = bcabcac$, $T^{(2)} = aabca$, $T^{(3)} = bcaa$

$T = bcabca c \$, aabca \$_2, bca a \$_3$



6.4 Longest Common Extensions

Application 4: Longest Common Extensions

- ▶ We implicitly used a special case of a more general, versatile idea:

The *longest common extension (LCE)* data structure:

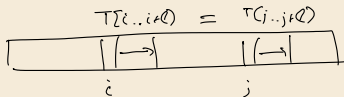
- ▶ **Given:** String $T[0..n-1]$
- ▶ **Goal:** Answer LCE queries, i. e.,
given positions i, j in T ,
how far can we read the same text from there?
formally: $\text{LCE}(i, j) = \max\{\ell : T[i..i+\ell] = T[j..j+\ell]\}$

Application 4: Longest Common Extensions

- We implicitly used a special case of a more general, versatile idea:

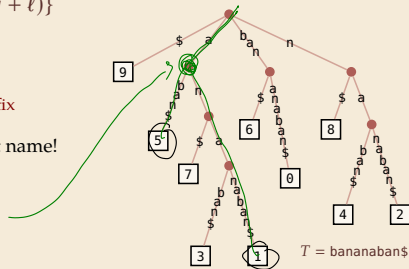
The *longest common extension (LCE)* data structure:

- **Given:** String $T[0..n-1]$
- **Goal:** Answer LCE queries, i. e.,
given positions i, j in T ,
how far can we read the same text from there?
formally: $LCE(i, j) = \max\{\ell : T[i..i+\ell] = T[j..j+\ell]\}$



↪ use suffix tree of T !

- In \mathcal{T} : $LCE(i, j) = LCP(T_i, T_j) \rightsquigarrow$ same thing, different name!
= string depth of
lowest common ancestor (LCA) of
leaves \boxed{i} and \boxed{j}





- in short: $LCE(i, j) = LCP(T_i, T_j) = \text{stringDepth}(\text{LCA}(\boxed{i}, \boxed{j}))$

T_5 aban\$
 T_1 ananaban\$

Efficient LCA

How to find lowest common ancestors?

- ▶ Could walk up the tree to find LCA $\rightsquigarrow \Theta(n)$ worst case 
- ▶ Could store all LCAs in big table $\rightsquigarrow \underline{\Theta(n^2)}$ space and preprocessing 

Efficient LCA

How to find lowest common ancestors?

- ▶ Could walk up the tree to find LCA $\rightsquigarrow \Theta(n)$ worst case 🙄
- ▶ Could store all LCAs in big table $\rightsquigarrow \Theta(n^2)$ space and preprocessing 🙄



Amazing result: Can compute data structure in $\Theta(n)$ time and space that finds any LCA is constant(!) time.

- ▶ a bit tricky to understand
- ▶ but a theoretical breakthrough
- ▶ and useful in practice

and suffix tree construction inside ...



\rightsquigarrow for now, use $O(1)$ LCA as black box.

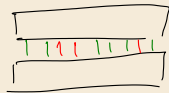
\rightsquigarrow After linear preprocessing (time & space), we can find LCEs in $O(1)$ time.

Application 5: Approximate matching

k -mismatch matching:

- ▶ **Input:** text $T[0..n-1]$, pattern $P[0..m-1]$, $k \in [0..m)$
- ▶ **Output:**
 - ▶ smallest i so that $T[i..i+m)$ and P differ in at most k characters
 - ▶ or NO_MATCH if there is no such i

"Hamming distance $\leq k$ "



Hamming
distance 4

~> searching with typos

- ▶ Assume longest common extensions in $T\$_1P\$_2$ can be found in $O(1)$
 - ~> generalized suffix tree \mathcal{T} has been built
 - ~> string depths of all internal nodes have been computed
 - ~> constant-time LCA data structure for \mathcal{T} has been built

Clicker Question



What is the Hamming distance between heart and beard?

pingo.upb.de/622222

Kangaroo Algorithm for approximate matching



```
1 procedure kMismatch( $T[0..n-1], P[0..m-1]$ )  
2   // build LCE data structure  
3   for  $i := 0, \dots, n-m-1$  do  
4     mismatches  $:= 0$ ;  $t := i$ ;  $p := 0$   
5     while mismatches  $\leq k \wedge p < m$  do  
6        $\ell := \text{LCE}(t, p)$  // jump over matching part  
7        $t := t + \ell + 1$ ;  $p := p + \ell + 1$   
8       mismatches  $:= \text{mismatches} + 1$   
9     if  $p == m$  then  
10      return  $i$ 
```

brute force

$O(n \cdot m)$

► **Analysis:** $\Theta(n + m)$ preprocessing + $O(n \cdot k)$ matching

↪ very efficient for small k

► State of the art

- $O\left(n \frac{k^2 \log k}{m}\right)$ possible with complicated algorithms
- extensions for edit distance $\leq k$ possible

Application 6: Matching with wildcards

- ▶ Allow a wildcard character in pattern

stands for arbitrary (single) character

unit*	P
in_unit5_we_will	T

- ▶ similar algorithm as for k -mismatch $\rightsquigarrow O(n \cdot k + m)$ when P has k wildcards

generalize k -mismatches to k edit distance (+ ins dels)

* * *

Many more applications, in particular for problems on biological sequences

20+ described in Gusfield, *Algorithms on strings, trees, and sequences* (1999)

Suffix trees – Discussion

- ▶ Suffix trees were a threshold invention

👍 linear time and space

👍 suddenly many questions efficiently solvable in theory



Suffix trees – Discussion

- ▶ Suffix trees were a threshold invention

👍 linear time and space $\Theta(n)$

👍 suddenly many questions efficiently solvable in theory



👎 construction of suffix trees:
linear time, but significant overhead

👎 construction methods fairly complicated

👎 many pointers in tree incur large space overhead



storing tries in Java • $\Sigma = \text{ASCII}$ (128 characters)

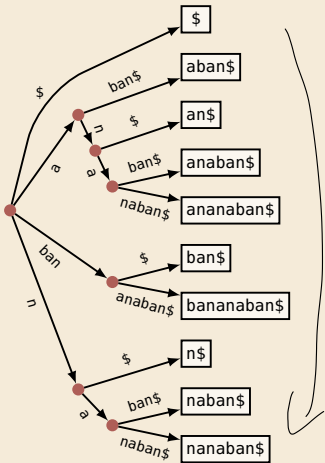
① array[0..127] of children

② dictionary: $\Sigma \mapsto \text{child}$
e.g. BSTs

6.5 Suffix Arrays

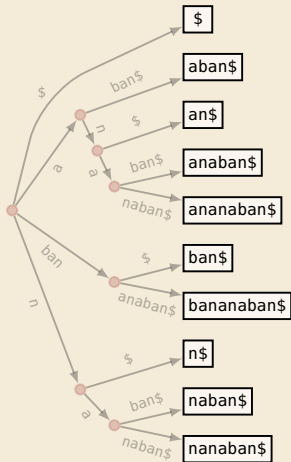
Putting suffix trees on a diet

- **Observation:** order of leaves in suffix tree
= suffixes lexicographically *sorted*



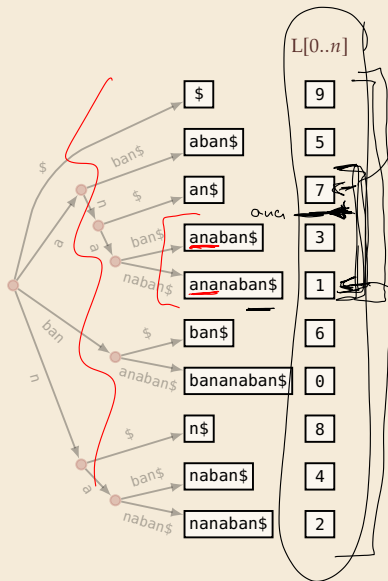
suffixes in leaves
are sorted

Putting suffix trees on a diet



- ▶ **Observation:** order of leaves in suffix tree
= suffixes lexicographically *sorted*
- ▶ Idea: only store list of leaves $L[0..n]$
- ▶ Enough to do efficient string matching!
 1. Use binary search for pattern P
 2. check if P is prefix of suffix after found position
- ▶ **Example:** $P = \text{ana}$

Putting suffix trees on a diet



$L[0..n]$

- **Observation:** order of leaves in suffix tree
= suffixes lexicographically *sorted*

ana > an\$

- Idea: only store list of leaves $L[0..n]$

- Enough to do efficient string matching!

1. Use binary search for pattern P *2 searches*
2. check if P is prefix of suffix after found position

- **Example:** $P = \underline{ana}$ ana > ananab\$

↪ $L[0..n]$ is called *suffix array*:

$L[r] = (\text{start index of } r\text{th suffix in sorted order})$

- using L , can do string matching with
 $\leq (\lg n + 2) \cdot m$ character comparisons