13 Text Indexing – Searching entire genomes

2 February 2026

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2025–26)
Philipps-Universität Marburg
version 2026-02-02 16:02

Learning Outcomes

Unit 13: ***Text Indexing***

1. Know and understand methods for text indexing: *inverted indexes*, *suffix trees*, *(enhanced) suffix arrays*
2. Know and understand *generalized suffix trees*
3. Know properties, in particular *performance characteristics*, and limitations of the above data structures.
4. Design (simple) *algorithms based on suffix trees*.
5. Understand *construction algorithms* for suffix arrays.

Outline

13 Text Indexing

13.1 Inverted Indexes and Tries

Text indexing

- *Text indexing* (also: *offline text search*):
  - case of string matching: find $P[0..m)$ in $T[0..n)$
  - 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 indexes

≈ same as "indices"

- original indexes 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 indexes

≈ same as "indices"

- original indexes 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 whitespaces
  - actual implementations typically support *stemming* of words
    `goes` $\rightarrow$ `go`, `cats` $\rightarrow$ `cat`

    language specific!
- store mapping from words to a list of occurrences $\rightsquigarrow$ *how?*

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.

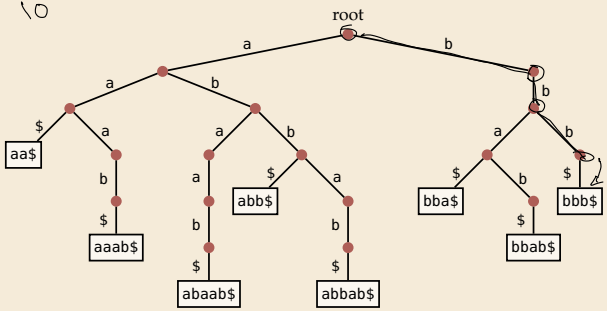
→ sli.do/cs566

Tries

- efficient dictionary data structure for strings
- name from re**trie**val, 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$, bbb$}



Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. Each stored string consists of m characters.
We now search for a query string Q with $|Q| = q$ (with $q \leq m$).
How many **nodes** in the trie are **visited** during this **query**?

A $\Theta(\log n)$

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

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

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

E $\Theta(m)$

F $\Theta(\log m)$

G $\Theta(q)$

H $\Theta(\log q)$

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

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

→ *sli.do/cs566*

Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. Each stored string consists of m characters.
We now search for a query string Q with $|Q| = q$ (with $q \leq m$).
How many **nodes** in the trie are **visited** during this **query**?

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

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

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

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

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

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

G $\Theta(q)$ ✓

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

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

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

→ *sli.do/cs566*

Clicker Question

Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. 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 $\Theta(n \cdot m)$

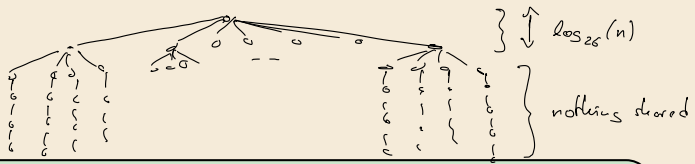
D $\Theta(n \log m)$

E $\Theta(m)$

F $\Theta(m \log n)$

→ *sli.do/cs566*

Clicker Question



Suppose we have a trie that stores n strings over $\Sigma = \{\texttt{A}, \ldots, \texttt{Z}\}$. Each stored string consists of m characters.
How many **nodes** does the trie have **in total** *in the worst case*?

- **A** ~~$\Theta(n)$~~
- **B** ~~$\Theta(n + m)$~~ total # chars
- **C** $\Theta(n \cdot m)$ ✓
- **D** ~~$\Theta(n \log m)$~~
- **E** ~~$\Theta(m)$~~
- **F** ~~$\Theta(m \log n)$~~

$n(m - \log n) + n$

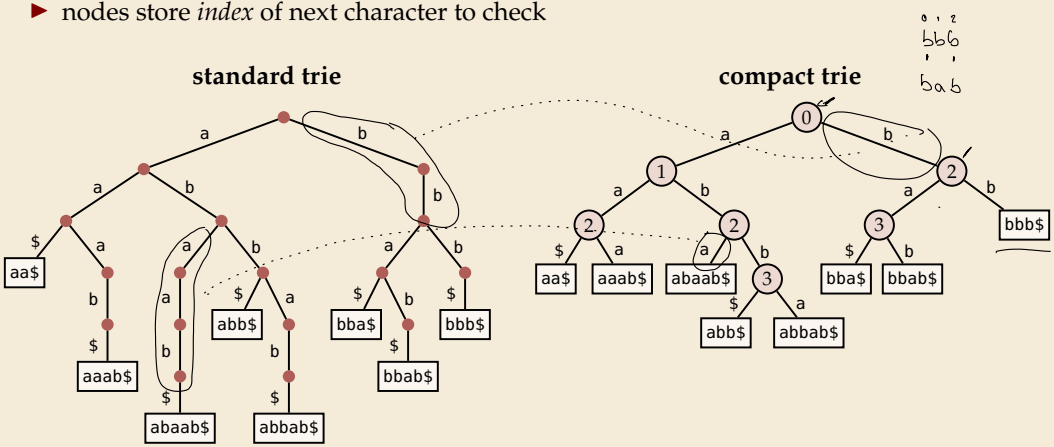
node in trie needs array of child pointers
$\Rightarrow$ $\Theta(\sigma)$ space per node

→ sli.do/cs566

Compact tries

=1 child

- compress paths of unary nodes into single edge
- nodes store *index* of next character to check



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

- all nodes ≥ 2 children ⇝ #nodes ≤ #leaves = #strings ⇝ linear space in # strings

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

```
ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGCCACCGCTGCCCTGCCCCTGGAGGGTGGCCCCACCGGC
CGAGACAGCGAGCATATGCAGGAAGCGGCAGGAATAAGGAAAAGCAGCCTCCTGACTTTCCTCGCTTGGTGGTTTGAGTGGACCTCCCAGGC
CAGTGCCGGGCCCCTCATAGGAGAGGAAGCTCGGGAGGTGGCCAGGCGGCAGGAAGGCGCACCCCCCCAGCAATCCGCGCGCCGGGACAGAA
TGCCCTGCAGGAACTTCTTCTGGAAGACCTTCTCCTCCTGCAAATAAAACCTCACCCATGAATGCTCACGCAAGTTTAATTACAGACCTGAA
```

- binary streams

```
0000001010100111101011100000111110001111101111100110110100001110001001101111000001000110101
0110110000110101101000000010000000011101011000001000011110101110110010001100101101110111111
1100010100010110010100000011101010100110000000011011000011001111110000101 0101011101111000011
1010111001001010101010000011111010011000000111100110101000000010010010000010110001100011011 1
```

need new ideas

13.2 Suffix Trees

Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

- Given: strings $S_1, \ldots, S_k$ **Example:** $S_1 =$ `superiorcalifornialives`, $S_2 =$ `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 $S_1, \ldots, S_k$ **Example:** S_1 = `superiorcalifornialives`, S_2 = `sealiver`
- Goal: find the longest substring that occurs in all k strings $\rightsquigarrow$ `alive`

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

Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

- Given: strings $S_1, \ldots, S_k$ **Example:** S_1 = `superiorcalifornialives`, S_2 = `sealiver`
- Goal: find the longest substring that occurs in all k strings $\rightsquigarrow$ `alive`

Can we do this in time $O(|S_1| + \cdots + |S_k|)$? 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 $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)

Suffix trees – Definition

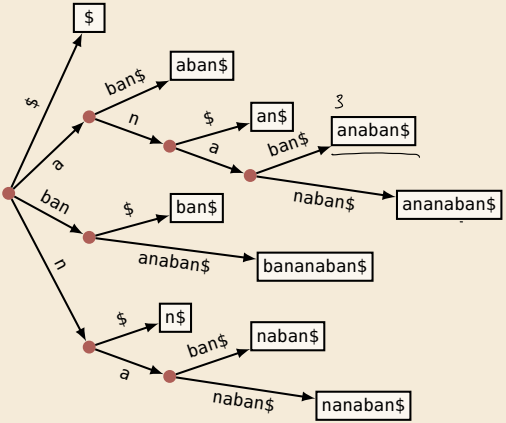
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)

Example:

$T =$ bananaban$

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

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



Suffix trees – Definition

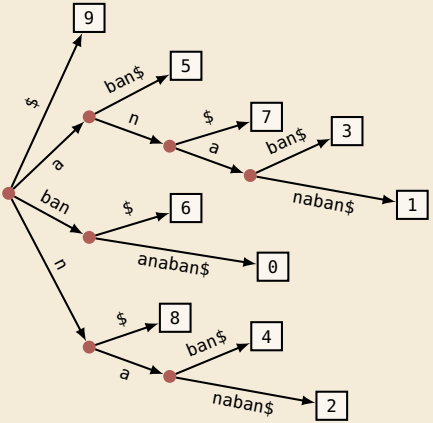
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)
- except: in leaves, store *start index* (instead of copy of actual string)

Example:

T = bananaban$

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

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



Suffix trees – Definition

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

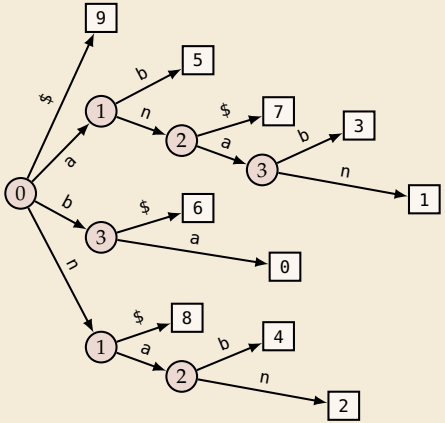
Example:

T = bananaban$

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

	0	1	2	3	4	5	6	7	8	9
T =	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

- $T[0..n]$ 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

- $T[0..n]$ 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 $\Theta(n)$ time!

- algorithms are a bit tricky to understand → EAA Book
- 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?

Clicker Question

Recap: Check all correct statements about suffix tree $\mathcal{T}$ of $T[0..n]$.

- **A** We require T to end with \$.
- **B** The size of $\mathcal{T}$ can be $\Omega(n^2)$ in the worst case.
- **C** $\mathcal{T}$ is a standard trie of all suffixes of $T\$$.
- **D** $\mathcal{T}$ is a compact trie of all suffixes of $T\$$.
- **E** The leaves of $\mathcal{T}$ store (a copy of) a suffix of $T\$$.
- **F** Naive construction of $\mathcal{T}$ takes $\Omega(n^2)$ (worst case).
- **G** $\mathcal{T}$ can be computed in $O(n)$ time (worst case).
- **H** $\mathcal{T}$ has n leaves.

→ *sli.do/cs566*

Clicker Question

Recap: Check all correct statements about suffix tree $\mathcal{T}$ of $T[0..n]$.

- **A** We require T to end with \$. ✓
- **B** ~~The size of $\mathcal{T}$ can be $\Omega(n^2)$ in the worst case.~~
- **C** ~~$\mathcal{T}$ is a standard trie of all suffixes of $T\$$.~~
- **D** $\mathcal{T}$ is a compact trie of all suffixes of $T\$$. ✓
- **E** ~~The leaves of $\mathcal{T}$ store (a copy of) a suffix of $T\$$.~~
- **F** Naive construction of $\mathcal{T}$ takes $\Omega(n^2)$ (worst case). ✓
- **G** $\mathcal{T}$ can be computed in $O(n)$ time (worst case). ✓
- **H** ~~$\mathcal{T}$ has n leaves.~~ $n+1$

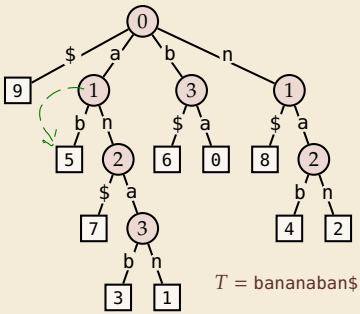
→ *sli.do/cs566*

13.3 Applications

Applications of suffix trees

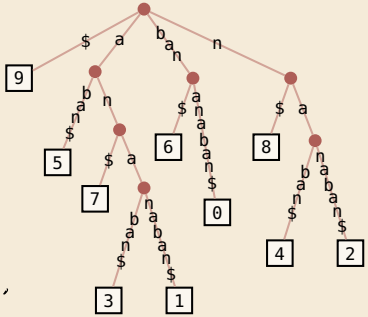
- In this section, always assume suffix tree $\mathcal{T}$ for T given.

Recall: $\mathcal{T}$ stored like this:



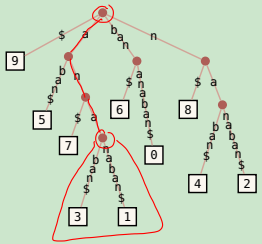
T = bananaban$

but think about this:



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

Clicker Question



What does T's suffix tree (on the left) tell you about the question whether T contains the pattern P = `ana`?

Check all that apply to this example.

A Nothing.

B P occurs in T.

C P does not occur in T.

D P occurs once in T.

E P occurs twice in T.

F P starts at index 0.

G P starts at index 1.

H P starts at index 2.

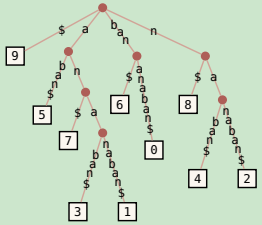
I P starts at index 3.

J P starts at index 4.

K P starts at index 7.

→ *sli.do/cs566*

Clicker Question



What does T's suffix tree (on the left) tell you about the question whether T contains the pattern P = `ana`?

Check all that apply to this example.

- **A** ~~Nothing.~~
- **B** P occurs in T. ✓
- **C** ~~P does not occur in T.~~
- **D** ~~P occurs once in T.~~
- **E** P occurs twice in T. ✓
- **F** ~~P starts at index 0.~~
- **G** P starts at index 1. ✓
- **H** ~~P starts at index 2.~~
- **I** P starts at index 3. ✓
- **J** ~~P starts at index 4.~~
- **K** ~~P starts at index 7.~~

→ *sli.do/cs566*

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

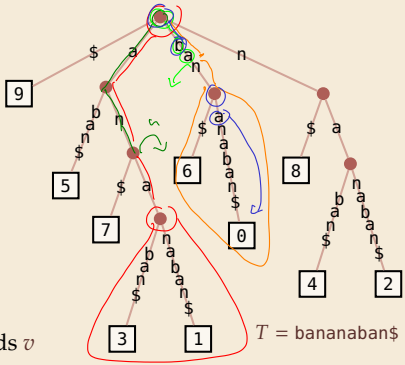
- P occurs in $T \iff P$ is a prefix of a suffix of T
- we have all suffixes in $\mathcal{T}$!

$\rightsquigarrow$ (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)
   $\rightsquigarrow$ 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
   $\rightsquigarrow$ P occurs at all leaves in subtree of v
3. **we run out of tree**
   reach a leaf ℓ with part of P left $\rightsquigarrow$ compare P to ℓ.

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!



Examples:

- P = ann
- P = baa
- P = ana
- P = ba
- P = 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 ⇝ 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 ⇝ Unit 7

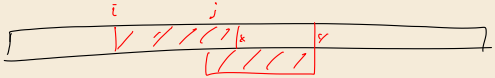
How can we efficiently check *all possible substrings?*

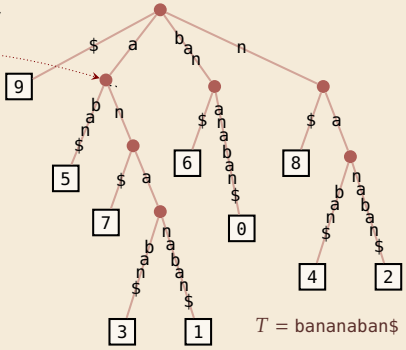
Repeated substrings = shared paths in *suffix tree*

- T_5 = aban$ and T_7 = an$ have *longest common prefix* 'a'

⇝ ∃ 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 ⇝ Unit 7

How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

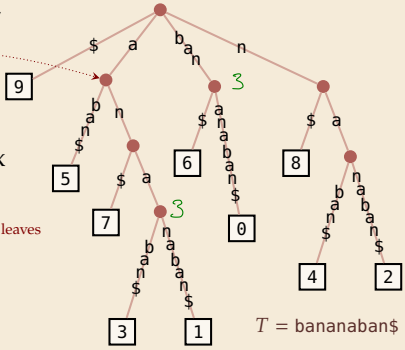
- T_5 = aban$ and T_7 = an$ have *longest common prefix* 'a'

⇝ ∃ 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



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 ⇝ Unit 7

How can we efficiently check *all possible substrings?*

Repeated substrings = shared paths in *suffix tree*

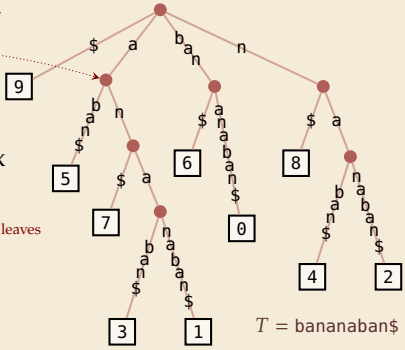
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* '`a`'

⇝ ∃ 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 ⇝ $\Theta(n)$ time

13.4 Generalized Suffix Trees

Generalized suffix trees

- longest *repeated* substring (of one string) feels very similar to
  longest *common* substring of several strings $T^{(1)}, \ldots, T^{(k)}$ 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 $T^{(1)}, \ldots, T^{(k)}$ 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

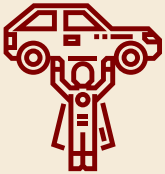
$\rightsquigarrow$ need a *single/joint* suffix tree for *several* texts

Generalized suffix trees

- longest *repeated* substring (of one string) feels very similar to longest *common* substring of several strings $T^{(1)}, \ldots, T^{(k)}$ 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

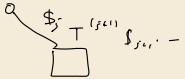
$\rightsquigarrow$ need a *single/joint* suffix tree for *several* texts

Enter: ***generalized suffix tree***



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

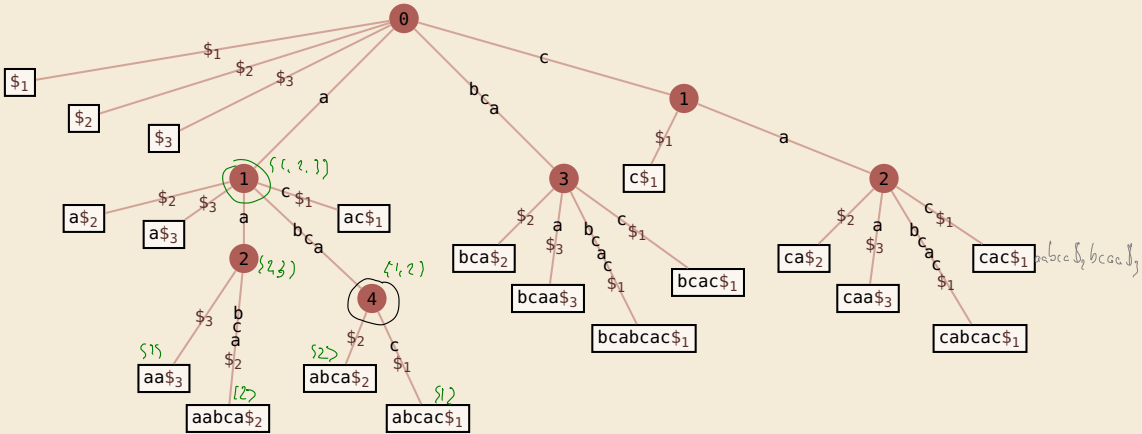
$\rightsquigarrow$ $\$_j$-edges always leads to leaves $\rightsquigarrow$ $\exists$ leaf (j, i) for each suffix $T_i^{(j)} = T^{(j)}[i..n_j]$



Generalized Suffix Tree – Example

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

T = bcabcac$1 aabca$2 bcaa$3



Application 3: Longest common substring

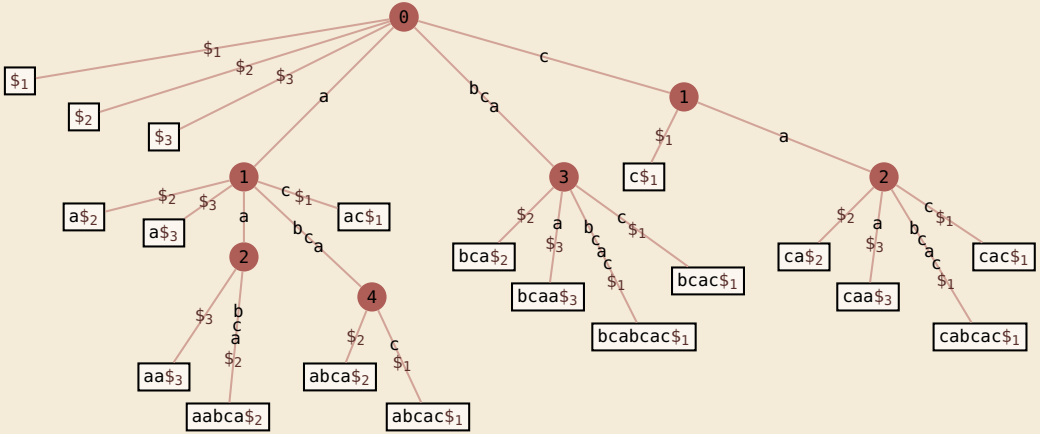
- With that new idea, we can find longest common substrings:
  1. Compute generalized suffix tree $\mathcal{T}$.
  2. Store with each node the *subset of strings* that contain its path label:
     - 2.1. Traverse $\mathcal{T}$ bottom-up.
     - 2.2. For a leaf (j, i), the subset is $\{j\}$.
     - 2.3. 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 $\{1, \ldots, k\}$.

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

"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)]

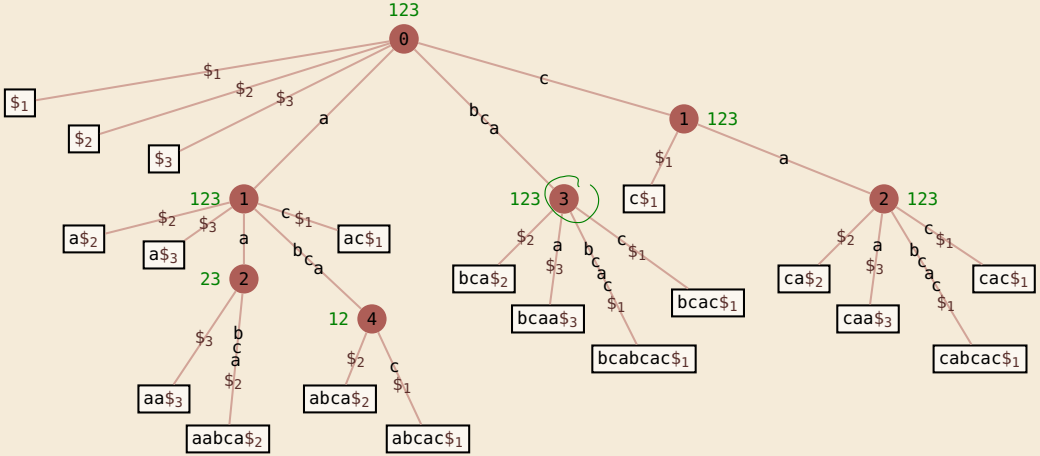
Longest common substring – Example

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



Longest common substring – Example

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



Further applications

Many more applications, in particular for problems on biological sequences

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

- key ingredient: longest common extensions

If you want to see more, come to *Algorithms of Bioinformatics* 😉

Suffix trees – Discussion

- Suffix trees were a threshold invention

- linear time and space
- suddenly many questions efficiently solvable in theory

Suffix trees – Discussion

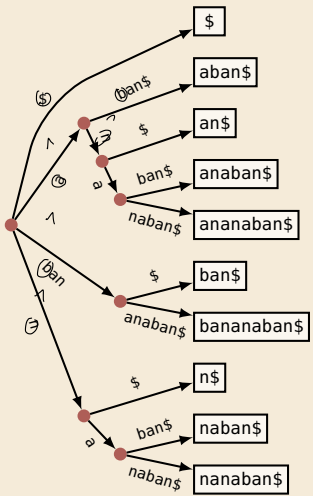
- Suffix trees were a threshold invention

- linear time and space
- 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

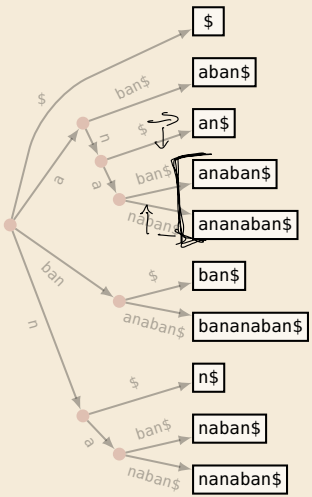
13.5 Suffix Arrays

Putting suffix trees on a diet



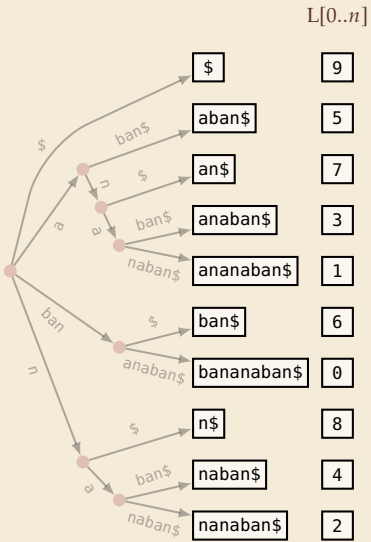
- **Observation:** order of leaves in suffix tree
  = suffixes lexicographically *sorted*

Putting suffix trees on a diet



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

Putting suffix trees on a diet



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

$\rightsquigarrow$ $L[0..n]$ is called ***suffix array***:

$L[r]$ = (start index of) rth suffix in sorted order

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