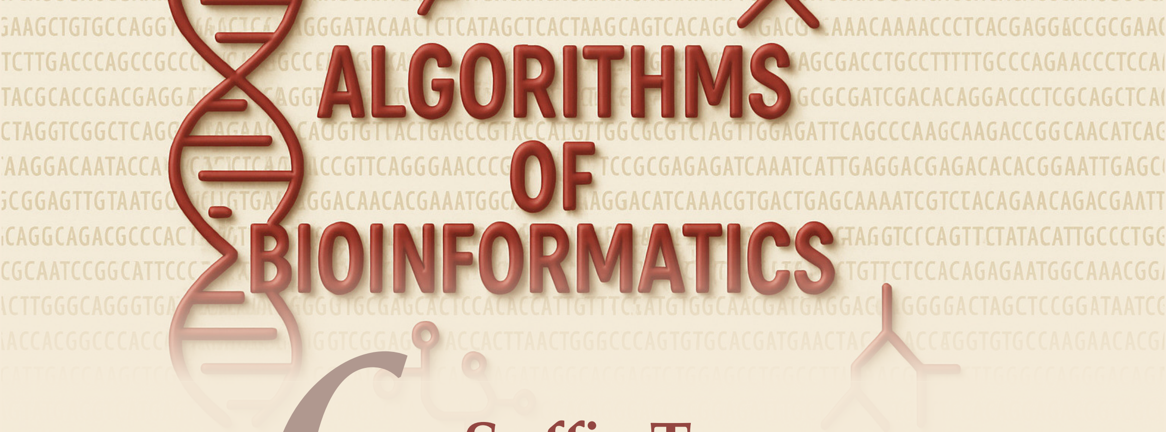


6 Suffix Trees

11 December 2025

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version `2025-12-11 14:04`

Outline

6 Suffix Trees

Context

We're still working towards practical solutions for the read mapping problem.

So far, our preprocessing was mostly getting smart on the ***reads/patterns****.*

$\rightsquigarrow$ Now preprocess the genome/text.

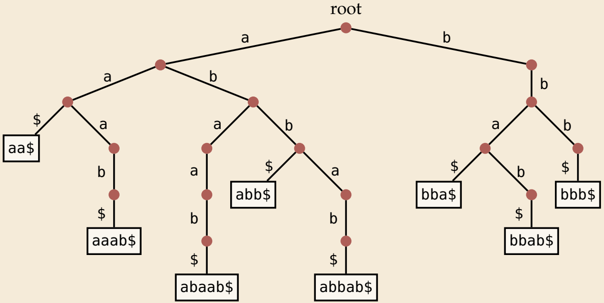
6.1 Suffix Trees

Recap: Tries

- efficient dictionary data structure for strings (or for Aho-Corasick automata 😉)
- name from re**trie**val, but pronounced "try"
- tree based on symbol comparisons

- **Assumption here:** stored strings are *prefix-free* (no string is a prefix of another)
  - strings of same length ✓
  - strings have "end-of-string" marker \$ ✓ some character $\notin \Sigma$

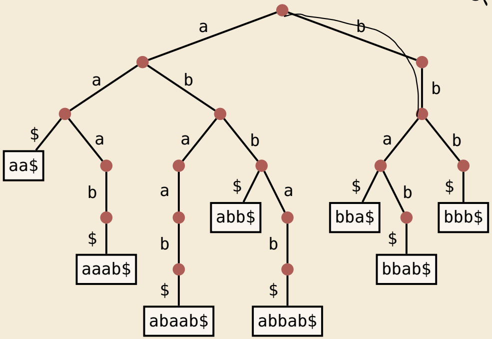
- **Example**:
  {aa\$, aaab\$, abaab\$, abb\$, abbab\$, bba\$, bbab\$, bbb\$}



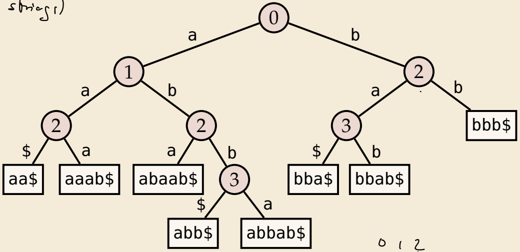
Compact tries

- compress paths of unary nodes into single edge =1 child
- nodes store *index* of next character to check

standard trie size = O(total length of all strings)



compact trie size = O(#strings)



0 1 2
bab$

- search gives first character of edge only ⇝ must check for match against stored string
- all nodes ≥ 2 children ⇝ #nodes ≤ #leaves = #strings ⇝ linear space

Suffix trees – A 'magic' data structure

Appetizer: Longest common substring problem

- Given: strings $S_1, \ldots, S_k$

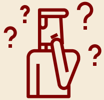
Example: $S_1 =$ `superiorcalifornialives`, $S_2 =$ `sealiver`

- 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`

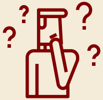


Can we do this in time $O(|S_1| + \cdots + |S_k|)$? How??

Suffix trees – A 'magic' data structure

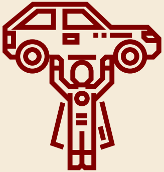
Appetizer: Longest common substring problem

- 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 $T[n] := \$$)

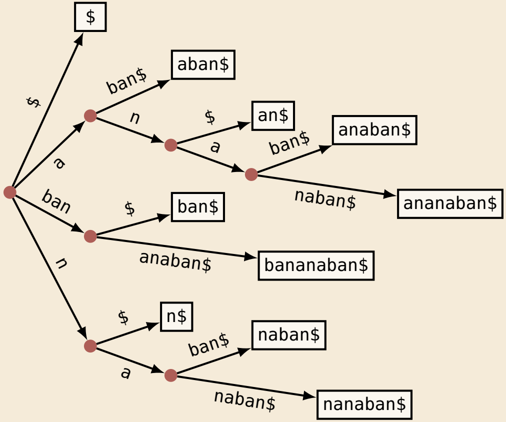
Example:

$T =$ bananaban\$

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

$T =$

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



Suffix trees – Definition

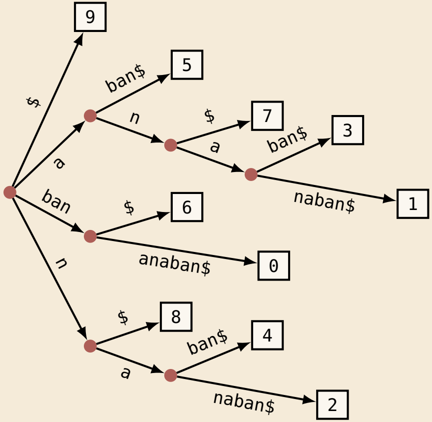
- suffix tree $\mathcal{T}$ for text $T = T[0..n)$ = compact trie of all suffixes of $T\$$ (set $T[n] := \$$)
- except: in leaves, store *start index* (instead of copy of actual string)

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)

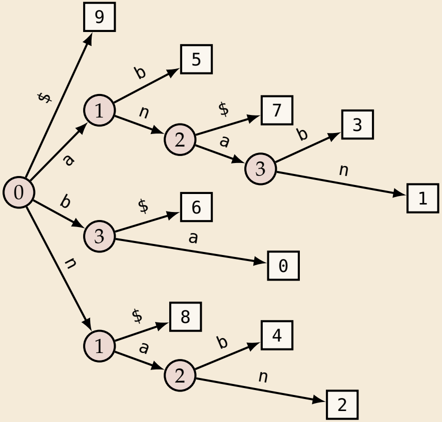
Example:

T = bananaban$

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

T =

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

- $T[0..n]$ 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!

- several fundamentally different methods known
- started as theoretical breakthrough
- now routinely used in bioinformatics practice

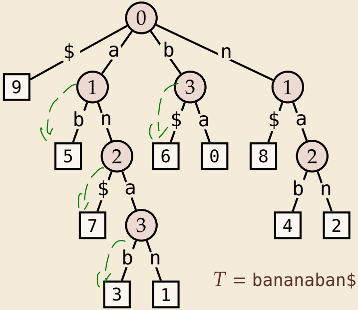
$\rightsquigarrow$ for now, take linear-time construction for granted. What can we do with them?

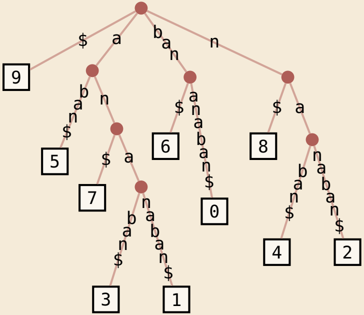
6.2 Applications

Applications of suffix trees

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

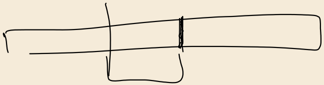
Recall: $\mathcal{T}$ stored like this: but think about this:





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

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

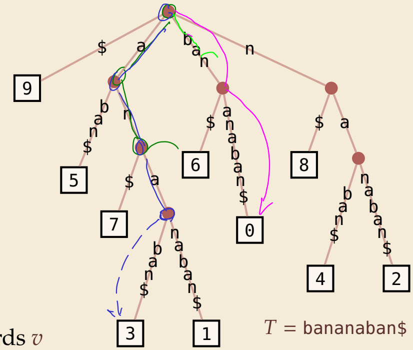
- $\boxed{P \text{ occurs in } T \iff P \text{ is a prefix of a suffix of } T}$
- we have all suffixes in $\mathcal{T}$!

⇝ (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**
   reach a leaf ℓ with part of P left ⇝ 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!



T = bananaban$

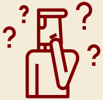
Examples:

- 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 $\rightsquigarrow$ 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

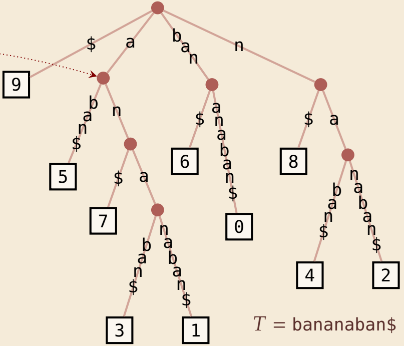
How can we efficiently check *all possible substrings?*

Repeated substrings $=$ shared paths in *suffix tree*

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

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

here single edge, can be longer path



T = `bananaban$`

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*

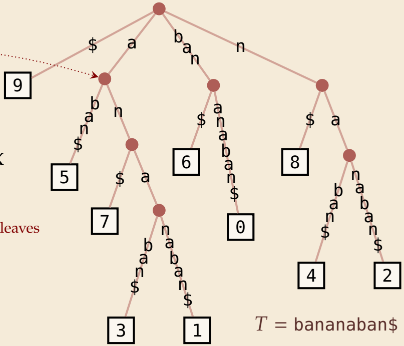
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* ‘`a`’

⇝ ∃ 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

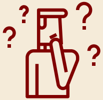


T = `bananaban$`

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*

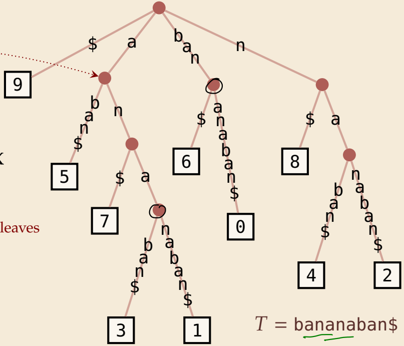
- T_5 = `aban$` and T_7 = `an$` have *longest common prefix* '`a`'

$\rightsquigarrow$ $\exists$ 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



- 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