

Suffix Trees and Applications

Philip Bille

Exact String Matching

- Input: Two strings over alphabet Σ .
 - A *target* $T[1 \dots n]$.
 - A *pattern* $P[1 \dots m]$.
- Output all occurrences of P in T .

Exact String Indexing

- Given a string $T[1 \dots n]$ build a data structure (or **index**) for T supporting the operation:
- $\text{Occur}(P[1 \dots m])$: Report all occurrences (if any) of P in T .
- For simplicity assume that $\sigma = O(1)$.

Tries

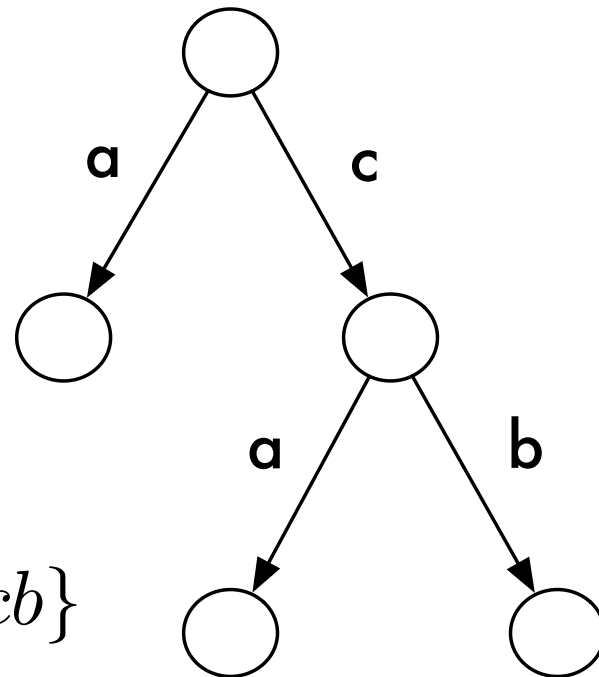
- **Membership** problem: Build a data structure for a set of strings $S = \{S_1, \dots, S_k\}$ supporting the following operation:
- $\text{Member}(P[1 \dots m])$: Decide if $P \in S$.
- A **trie** is useful for **retrieving** this information.

Tries

- A trie for $S = \{S_1, \dots, S_k\}$ is a rooted tree R such that:
- Each edge is labeled by a character from Σ .
- Child edges of a node have *distinct* labels.
- R has k leaves each corresponding to a string in S .

Example

- Trie for $S = \{a, ca, cb\}$
- $\text{Member}(P[1 \dots m])$:
Search from top.
- What about $S = \{a, c, ca, cb\}$

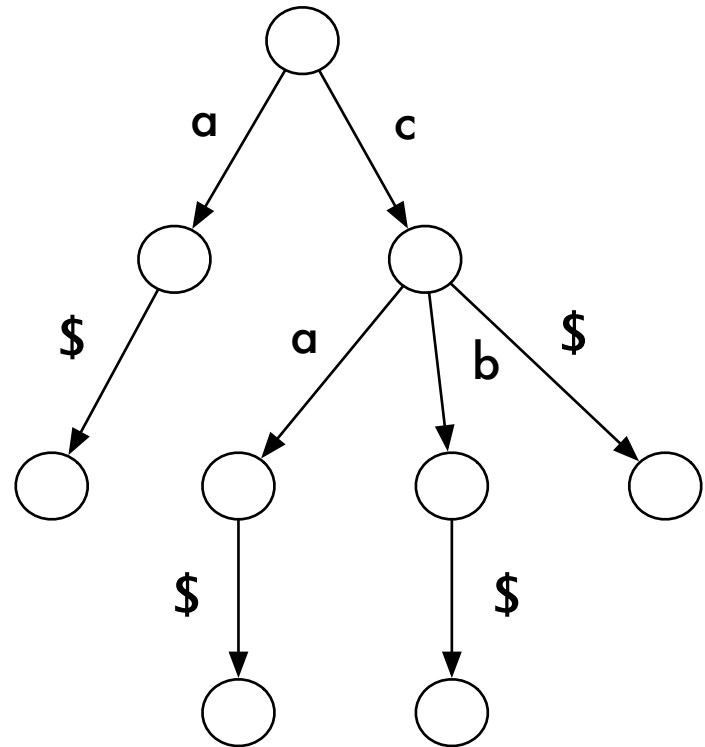


Tries

- Concatenate each string with \$.

$$S = \{a\$, c\$, ca\$, cb\$\}$$

- No string in S is a prefix of another.
- 1-to-1 correspondance between leaves and S .



Constructing tries

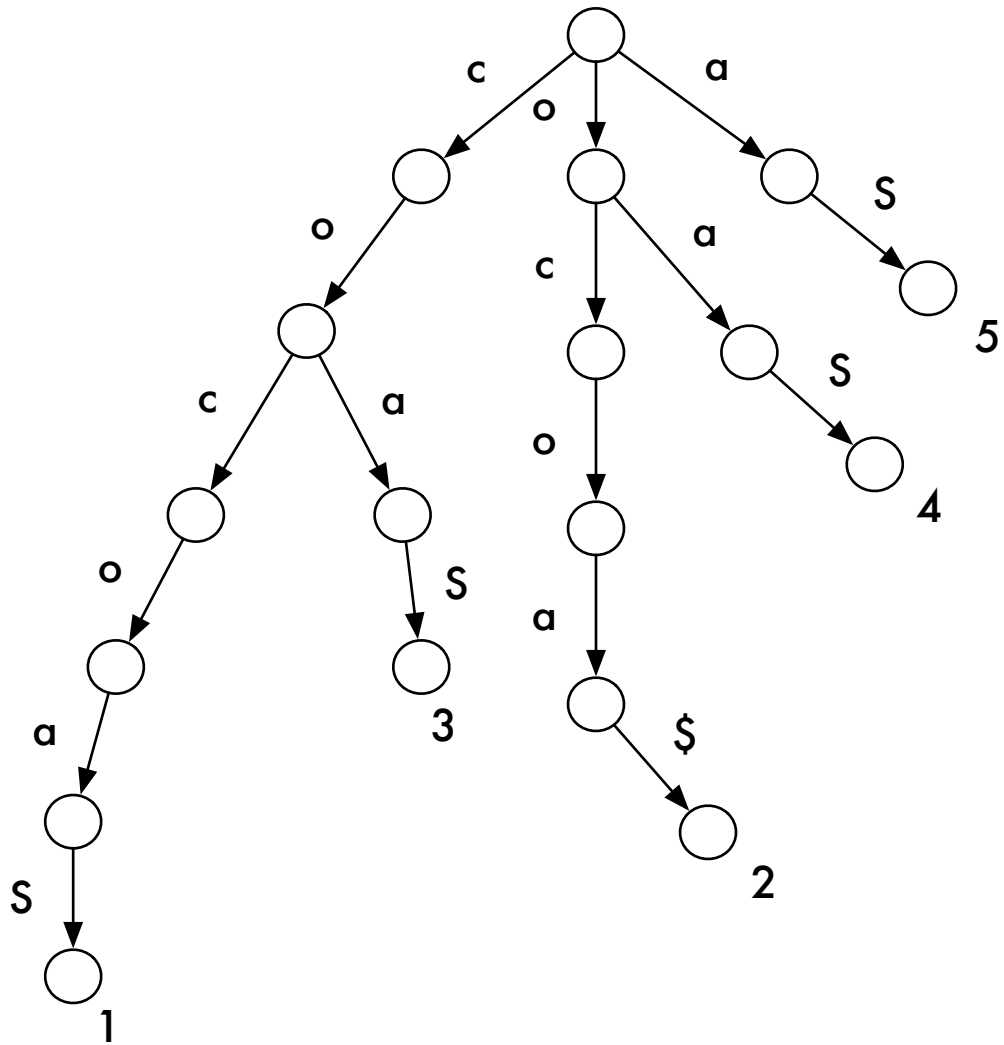
- Let $S = \{S_1, \dots, S_k\}$.
- How do we build trie for S ?
- Add one string at the time.
- To add S_i find the longest prefix of S_i in the current trie consisting of $\{S_1, \dots, S_{i-1}\}$. Create a new branch representing the remaining part of S_i .

Complexity

- Let $|S| = \sum_{i=1}^k S_i$.
- Member($P[1 \dots m]$): $O(m)$.
- Preprocessing: $O(|S|)$.
- Space: $O(|S|)$.

Tries and string indexing

- Recall: Build index for $T[1 \dots n]$ supporting $\text{Occur}(P[1 \dots m])$.
- Preprocessing: Build trie R for $\{T[1..n], T[2..n], \dots, T[n]\}$
- Leaf corresponding to $T[i..n]$ is labeled i .
- $\text{Occur}(P[1 \dots m])$: Find longest match for P in R . Report labels of all descendent leaves.



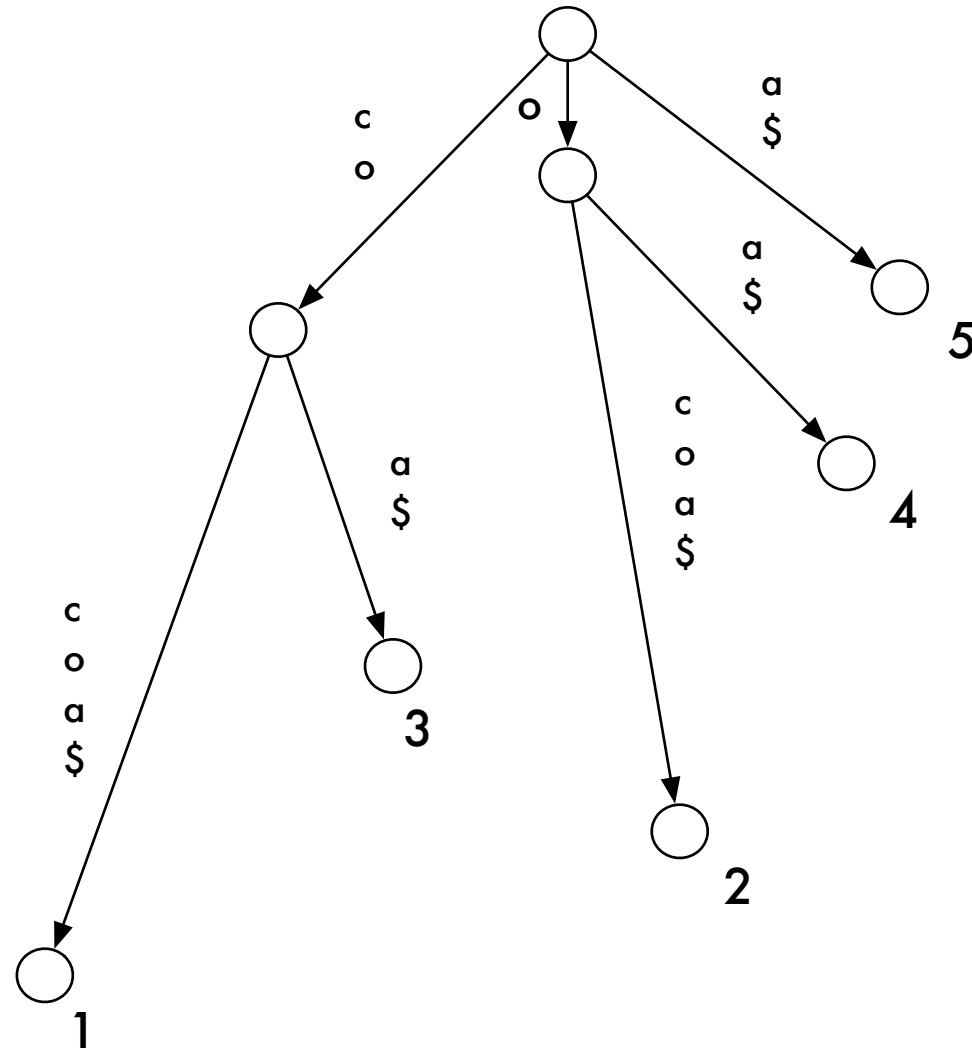
trie of suffixes for “cocoa”

Complexity

- Preprocessing: $O(n^2)$
- Space: $O(n^2)$
- Occur($P[1 \dots m]$): $O(nm)$
- Member is $O(m)$.

Compressed tries

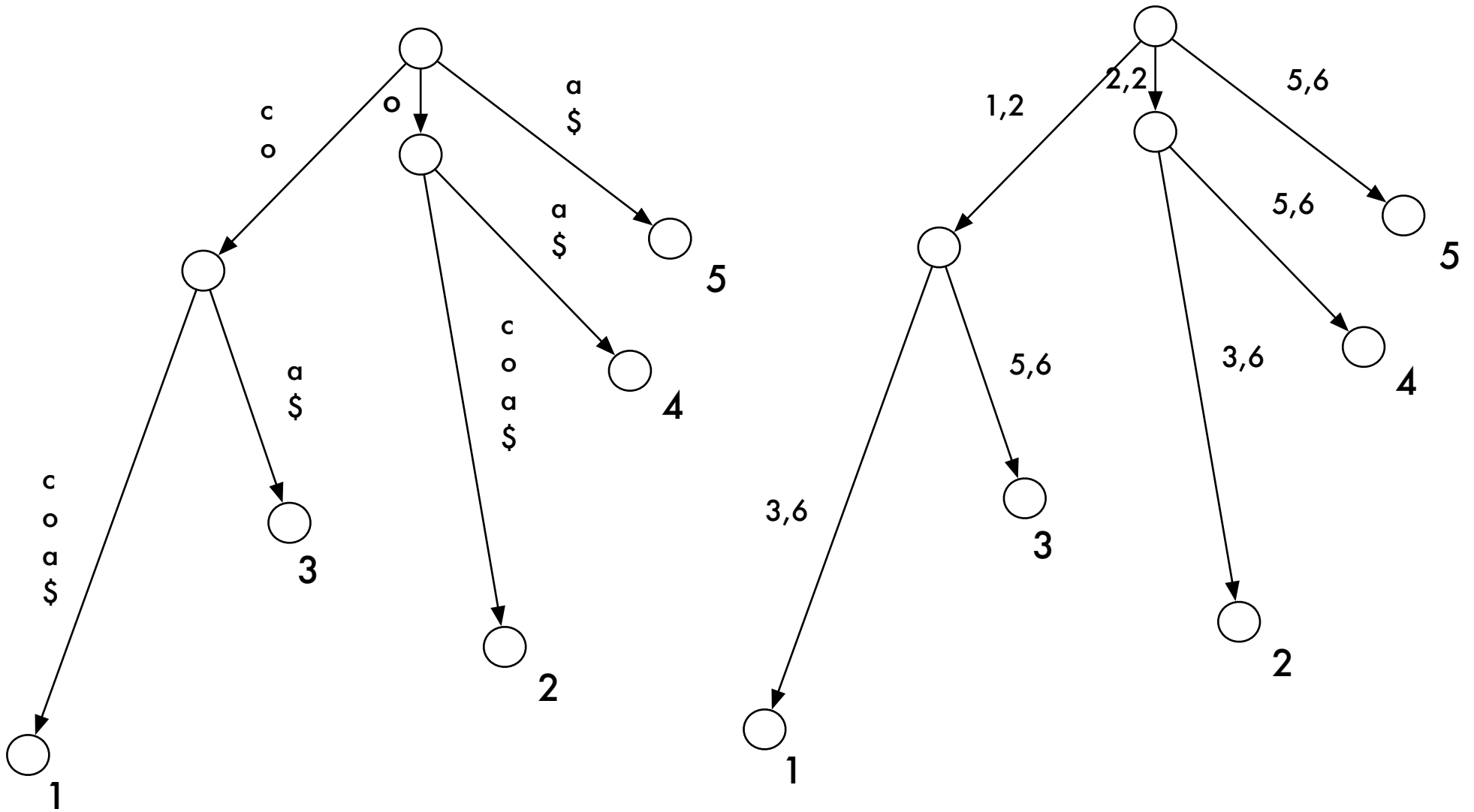
- Idea: Merge paths consisting of nodes with 1 child into a single node.



Compressed trie (or **suffix tree**) for “cocoa”

Suffix tree

- What is the size of the suffix tree:
- How many nodes and edges are in the suffix tree?
- How much space is needed for the edges?



Edge label compression for "cocoa" suffix tree

Complexity

- $\text{Occur}(P[1 \dots m]): O(m + z)$, where z is number of occurrences of P .
- Space: $O(n)$
- Preprocessing: $O(n^2)$ using tries.
- Can be done in $O(n)$ time.

- Let S be a string and R be the corresponding suffix tree.
- What the maximum height of R and when it this achieved?
- What is the minimum height of R and when is this achieved?
- What is the degree of the root of R ?
- How can we use R find the length of the longest substring occurring more than once in S ?

Applications

- String indexing.
- Exact set matching problem.
- Longest common substring.
- Frequent substring

Exact Set Matching

- Given a set of pattern strings $\{P_1, \dots, P_k\}$ and a target $T[1 \dots n]$ find all occurrences of pattern strings in T .

Solution

- Build suffix tree for T and lookup all patterns.
- Total time $O(n + \sum_{i=1}^k |P_i| + z)$, where the total number of occurrences is z .

Indexing multiple strings

- Suffix trees are useful for indexing a single string.
- How do we index multiple strings, e.g, a set of strings $S = \{S_1, \dots, S_k\}$?

“Generalized” suffix trees

- A **Generalized suffix tree** for S is a compressed trie of the all suffixes of each string in S .
- Each leaf is labeled by a pair (i, j) , where i identifies the string and j is the start position in S_i .

Construction

- Build a suffix tree R for string

$$C = S_1\$_1 \cdots S_k\$_k$$

- Each leaf in R is labeled by a start position of C .
- Remove suffixes that “span” multiple strings.
- Convert leaf labels into (i, j) pairs.
- Total time $O(|C|)$.

Longest common substring

- Given two strings S_1 and S_2 compute a longest common substring of S_1 and S_2 .
- Ex: Longest common substring of
 - $S_1 = \text{"superiorcalifornialives"}$
 - $S_2 = \text{"sealiver"}$is `"alive"`.

Solution

1. Build generalized suffix tree for S_1 and S_2 .
2. Mark each internal node v by 1 (2) if the subtree below v contains a leaf for a suffix of S_1 (S_2).
3. Traverse tree to find a node u of maximal **string-depth** marked by 1 and 2.

u correspond to a longest common substring.

Complexity

- Building the generalized suffix tree for S_1 and S_2 takes linear time.
- Step 2 and 3 can be implemented in linear time using a simple tree traversal. (exercise)

Frequent substrings

- How do we find **frequent** substrings in a set of strings? I.e. substrings that are common to many strings in the set.

Frequent substrings

- Let $S = \{S_1, \dots, S_k\}$ be a set of strings
- Define $l(i)$ for each $i = 2, \dots, k$ as the length of a longest substring that is common to at least i strings in S .
- The **frequent substring problem** is to compute $l(i)$ for $i = 2, \dots, k$.

{sandollar, sandlot, handler, grand, pantry}

i $l(i)$ substring

2	4	sand
3	3	and
4	3	and
5	2	an

Solution

- Compute generalized suffix tree R for S . Label each leaf with a number $1, \dots, k$ identifying the string.
- Compute the number of distinct string identifiers, $C(v)$, that occur below each internal node v in R .

Solution

- With $C(v)$'s compute vector $V(k)$ defined as the string depth of the deepest node such that $C(v) = k$.
- $V(k)$ is the length of the longest string that occurs *exactly* k times.
- Finally, compute $l(k)$ values from $V(k)$.

Computing $C(v)$

- Compute for each internal node v a bit-vector $b[1..k]$ where $b[i] = 1$ iff identifier of string i occurs in a leaf below v .
- Vector for v is computed by ORing the vectors of the child nodes v_1, \dots, v_l .
- Time for v is $O(lk)$.
- In total $O(nk)$ time. How do we get $C(v)$?

More Applications

- Show how to find the longest **palindrome** in S .
- Preprocess two strings S_1 and S_2 such that **longest common extension** queries can be solved efficiently. Assume that a nearest common ancestor data structure is available.

Conclusion

- Suffix trees can be built in linear time and space.
- Provides fundamental data structure with a huge number of applications.