String Matching: Tries, Suffix Trees, and Suffix Arrays

CS 7800/4810 Lecture Notes

1 Introduction to String Matching

1.1 Topics Covered

- Tries
- Compressed Tries
- Suffix Trees and Arrays

1.2 The String Matching Problem

Given:

- Text T and Pattern P
- Both are strings over alphabet Σ

Goal: Find some or all occurrences of P in T as substrings.

1.3 Classical Approaches

One-shot algorithms achieve $O(T)$ time:

- Knuth-Morris-Pratt (SICOMP 1977)
- Boyer-Moore (CACM 1977)
- Karp-Rabin (IBM JRD 1987)

Static Data Structure approach:

- Preprocess T, then query with P
- Goal: $O(|P|)$ query time, $O(|T|)$ space

Other data structures consider cases when P has wildcards or when P need not match as an exact substring (Hamming/edit distance). See:

- Cole, Gottlieb, Levenstein (STOC 2004)
- Maaß and Nowak (CPM 2005)

2 Tries

2.1 Warm Up: Pred Among Strings

Given a collection of strings $\{T_1, \ldots, T_k\}$, support predecessor queries (e.g., library search).

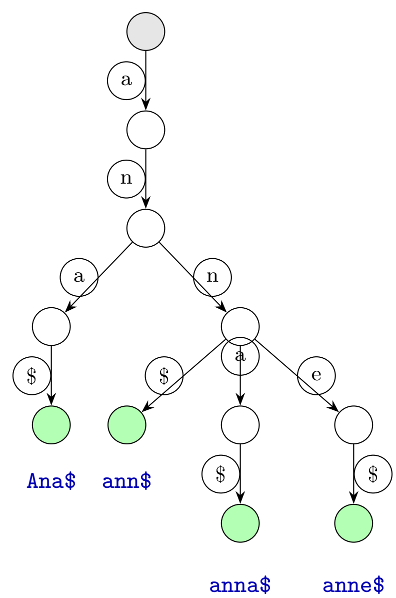
Definition 1 (Trie)**.** *A **trie** is a rooted tree with child branches labeled with letters in* Σ*.*

Key properties:

- Strings are represented as root-to-leaf paths in the trie
- Add a new letter \$ to the end of each string (otherwise cannot distinguish prefixes as absent or present)

2.2 Example

For the set $\{\text{Ana}, \text{ann}, \text{anna}, \text{anne}\}$:



3 Trie Representation

Let $T = \#\text{nodes in trie} \leq \sum_{i=1}^{k} |T_i|$.

Each node stores its children. The following table summarizes different representation options:

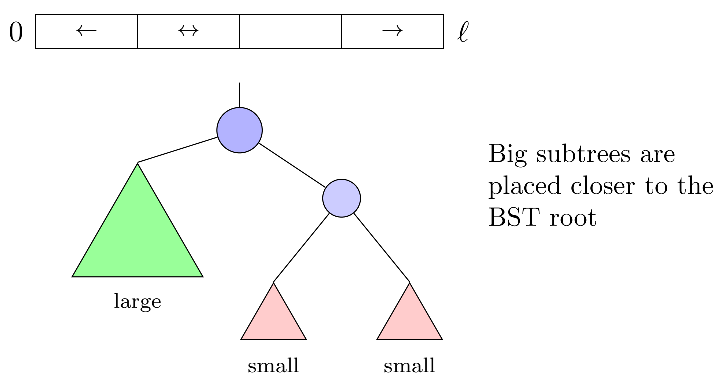
#	Representation	Query Time	Space
(1)	Array	$O(\|P\|)$	$O(T \cdot \|\Sigma\|)$
(2)	Balanced BST	$O(\|P\| \cdot \log \|\Sigma\|)$	$O(T)$
(3)	Hash table (no predecessor)	$O(\|P\|)$	$O(T)$
(3.5)	vEB / Y-fast	$O(\|P\| \cdot \log\log \|\Sigma\|)$	$O(T)$
(3.75)	Trays [Koplowitz & Levenstein 2006]	$O(\|P\| + \log \|\Sigma\|)$	$O(T)$
(4)	Weight-balanced BST	$O(\|P\| + \log k)$	$O(T)$
(5)	Leaf trimming	$O(\|P\| + \log \|\Sigma\|)$	$O(T)$
(*)	vEB only when you fall off	$O(\|P\| + \log\log \|\Sigma\|)$	$O(T)$

4 Weight-Balanced BST Representation

Achieves $(|P| + \log k)$ query time with $O(T)$ space.

Definition 2. *Weight each node by the number of descendant leaves.*

The goal is to get big subtrees closer to the root.



Key insight: Every 2 edges follow either:

- Advance 1 letter in P, or
- Reduce # candidate T_i to $\frac{2}{3}$

5 Leaf Trimming

Cut below maximally deep nodes with $\geq |\Sigma|$ descendant nodes.

- # leaves $\leq \frac{|T|}{|\Sigma|} \Rightarrow$ method (1)
- # branching nodes $< \frac{|T|}{|\Sigma|} \Rightarrow$ method (2)
- # non-branching top nodes: $k < |\Sigma| \Rightarrow$ method (4)

This achieves $O(|P| + \log |\Sigma|)$ query time.

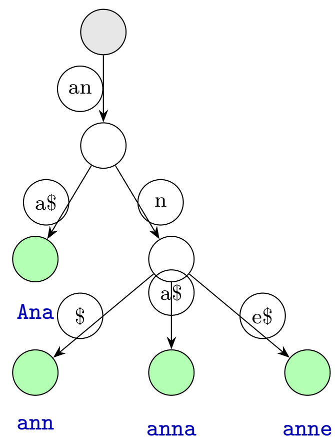
5.1 String Sorting

Sorting strings: $O(T + k \cdot \log|\Sigma|)$ via integer sorting, or $O(T \cdot k \log k)$ via comparison.

6 Compressed Tries

Definition 3 (Compressed Trie)**.** *Contract non-branching paths into a single edge.*

The same representations apply, with $T = \#\text{compressed nodes} = O(k)$ nodes.



7 Suffix Trees

Definition 4 (Suffix Tree)**.** *A **suffix tree** of text T is a compressed trie of all $|T|$ suffixes $T[i:]$ of T (with \$ appended).*

7.1 Example: "banana$"

The suffixes are:

$$\begin{aligned} 0 &: \text{banana\$} \\ 1 &: \text{anana\$} \\ 2 &: \text{nana\$} \\ 3 &: \text{ana\$} \\ 4 &: \text{na\$} \\ 5 &: \text{a\$} \\ 6 &: \text{\$} \end{aligned}$$

Properties:

- $|T| + 1$ leaves
- Edge labels are substrings $T[i:j]$
- Store as two indices (i, j)
- $\Rightarrow O(|T|)$ space

8 Applications of Suffix Trees

- **Pattern search:** Search for P gives subtree whose leaves correspond to all occurrences of P
  - $O(|P|)$ time via hash (+vEB)
  - $O(|P| + \log|\Sigma|)$ via trays $\Rightarrow$ leaves stored in T
- **List first k occurrences:** $O(k)$ additional time
  - Every node points to leftmost descendant leaf
  - Leaves connected via linked list
- **Count occurrences:** $O(1)$ additional time (subtree sizes)
- **Longest repeated substring in T:** $O(|T|)$ time
  - = branching node of maximum "letter depth"
- **Longest common substring:** $T[i:]$ vs $T[j:]$ in $O(1)$ via LCA query

9 Suffix Arrays

Definition 5 (Suffix Array)**.** *The **suffix array** SA of text T stores the indices of suffixes sorted lexicographically.*

9.1 Example: "banana$"

SA	Suffix
6	$
5	a$
3	ana$
1	anana$
0	banana$
4	na$
2	nana$

Note: $\$ < a < b < \dots$
Space: $O(|T|)$
Construction: $O(|T| + \text{sort}(|\Sigma|))$

9.2 LCP Array

Definition 6 (LCP Array)**.** *$LCP[i]$ = length of longest common prefix of i^{th} and $(i+1)^{st}$ suffix in sorted order.*

For "banana$":

$$\text{LCP} = [0, 1, 3, 0, 0, 2]$$

9.3 Searching with Suffix Arrays

Searchable in $O(|P| \cdot \log|T|)$ via binary search.

When binary searching in interval SA$[i : j]$: only need to compare from letter $\text{RMQ}_{\text{LCP}}(i, j-1)$.

This uses **Range Minimum Queries** on the LCP array.

9.4 Cartesian Tree of LCP

The Cartesian tree of the LCP array provides efficient range minimum queries.

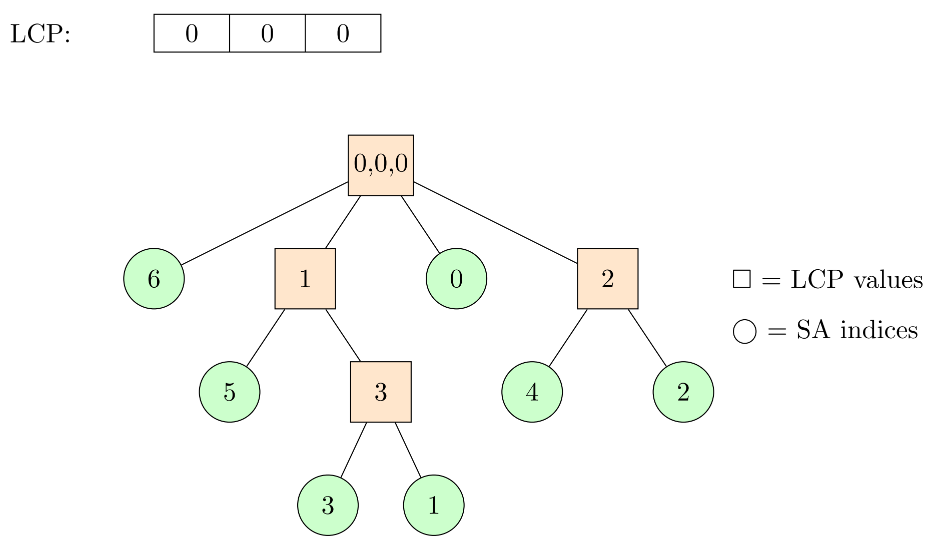
Definition 7 (Cartesian Tree)**.** *Given an array $A[0..n-1]$, the Cartesian tree is a binary tree where:*

- ***Heap property:*** *Each node contains the minimum value in its subtree's range*
- ***BST property:*** *Inorder traversal yields elements in original array order*

Construction: For array $A[i..j]$:

1. Find the minimum element $A[m]$ in the range (leftmost if ties)
2. Make $A[m]$ the root
3. Recursively build left subtree from $A[i..m-1]$
4. Recursively build right subtree from $A[m+1..j]$

Example: For LCP $= [0, 1, 3, 0, 0, 2]$:



Key property: $\text{RMQ}(i, j) =$ LCA of nodes i and j in the Cartesian tree.

10 Summary

Data Structure	Query Time	Space
Suffix Tree (hash)	$O(\|P\|)$	$O(\|T\|)$
Suffix Tree (trays)	$O(\|P\| + \log\|\Sigma\|)$	$O(\|T\|)$
Suffix Array + LCP + RMQ	$O(\|P\| + \log\|T\|)$	$O(\|T\|)$