Exact Matching: Data Structures

In the next lectures we will look at Data Structures that make searching for exact string matches more efficient:

① Keyword Trees — Preprocess patterns
② Suffix Tries
③ Suffix Trees
④ Suffix Arrays
⑤ Burrows-Wheeler Transform
(②–⑤: Preprocess target)

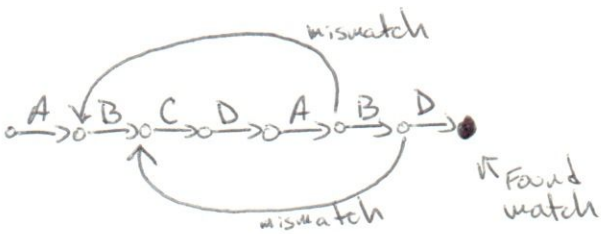
—o—

Keyword Tree (Aho-Corasick)

→ First, let's understand the KMP algorithm as automata.

Consider the following pattern:

```
    A B C D A B D
spm: - 0 0 0 1 2 0
```



Examine T one character at a time, if match character in edge, follow edge, if mismatch, follow mismatch edge. (in this picture, go to root if no mismatch edge).

Note: Naive matching algorithm corresponds to starting at root after mismatch.

Note: We'll refer to mismatch edges as "failure links"

Note: We can reconstruct failure links from spm_i values.

Keyword Tree: Generalization of "KMP as automata" idea to a set of patterns.

Ex. {potato, tattoo, theater, other}

Def by construction:

0) Create root
1) For each pattern P in set
   - set current node v to root
   - for $i = 1, \ldots, |P|$
     - if edge (v, w) labeled $P[i]$ exits current node
       - set current to w
     - else
       - create new node, label edge $P[i]$

How can we use failure links? Edge (v, w) s.t. $L(w)$ is label of node v, w is node with label $L(w)$ and α is longest proper suffix of $L(v)$ matching a prefix of some pattern P in set.

How can we use keyword tree to find all occurrences of patterns in set in target T?

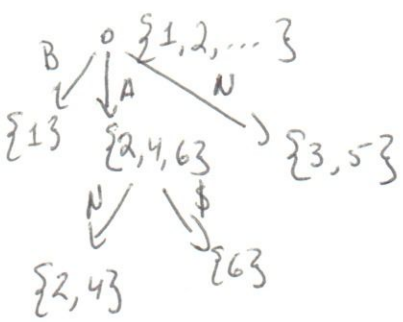
In project 2 you will implement the Aho-Corasick algorithm that constructs the keyword tree in linear time (including failure links).

Suffix Tries

Consider the case where we match many small strings against a very long target string (e.g., second generation short read sequencing). The algorithms we've seen so far are $O(|P| + |T|)$ for each pattern. We now approach the problem by preprocessing the target, in linear time, so that matching each pattern is only $O(|P|)$.

Conceptually, we need a data structure that guides the search for a given pattern along the target. Something like a search tree:

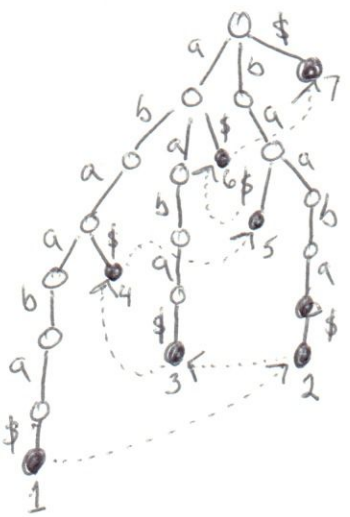
T: BANANA$



Key Idea: Use a keyword tree-like structure on <u>suffixes</u> of target.

Suffix Trie:

S = abaaba$



Like keyword tree, edges are labeled with characters in alphabet. Edges exiting a node are labeled with distinct characters.

Every path from root to leaf represents a <u>suffix</u> of S.

Every suffix is represented by some path from root to leaf.

Q: How many leaves will there be?

Key Idea: Every substring of S is a <u>prefix</u> of some <u>suffix</u> of S.

Substring search: Follow path given by query string. Searching suffix trie is $O(|P|)$ regardless of $|T|$.

Applications:

1. Is P a substring of T?
2. Is P a suffix of T?
3. How many times does P occur in T?
4. What is the first, lexicographically, suffix of T?
5. Find the longest repeat in T?

Suffix Links

Suffix links connect node "$x\alpha$" to node labeled "α". For construction algorithm, the most important links are those connecting suffixes of full string (see example).

Note: Every node has a suffix link

Q: How do we know there exists a node labeled "α"?

A: Every substring is the prefix of some suffix. Since suffix trie contains all suffixes S, it contains a path representing S, and therefore contains a node for every prefix of S.

Suffix Trie construction

Obs: Naive algorithm is $O(m^3)$ but I'll give you an $O(m^3)$ algorithm that can be improved to $O(m)$, but we'll stop at $O(m^2)$... $(|S| = m)$.

High-level algorithm

Construct tree T_1

For $i = 1, \ldots, m-1$ (phase $i+1$)

  For $j = 1, \ldots, i+1$ (extension j)

    Find node labeled $S[j \cdots i]$ in current tree, (*)

    extend path adding character $S[i+1]$ if needed

Step (*) searches string $S[j, \ldots, i]$ in tree, an $O(i-j+1)$ operation. So this is an $O(m^3)$ algorithm.

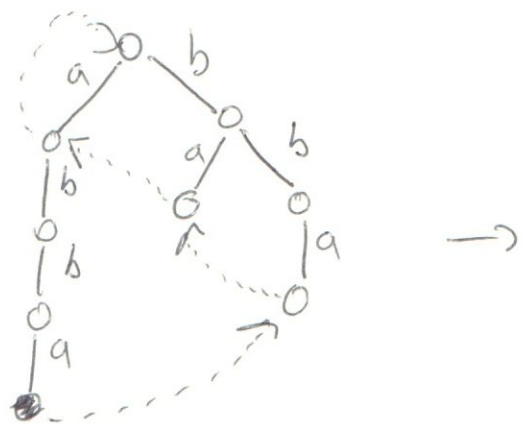
Ex: abbacabaa

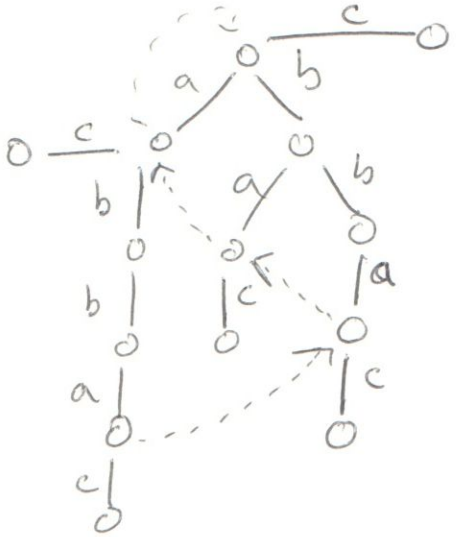
→ $i = 4$

Obs: The current tree contains suffixes

abba	we are extending → to	abbac
bba		bbac
ba		bac
a		ac
		c

Can we do this without using search for step (*)? Yes, using suffix links, extension takes $O(1)$ time.





abba

Q) How do we update the suffix links?

We extended node labeled $x\alpha$ to node $x\alpha y$,
Since there was suffix link $x\alpha \dots > \alpha$, there must now be link $x\alpha y \dots > \alpha y$, which we just created!
(if necessary)

New extension phase (build tree $i+1$):

Set Current suffix = loggest suffix
While current != root or current has no edge labeled S[i+1]:
    add child of current with edge labeled S[i+1]
    follow suffix link of current to next shortest suffix
Add suffix links connecting new nodes in the order they were added.

Note: You can design implementation that adds suffix links as you go along

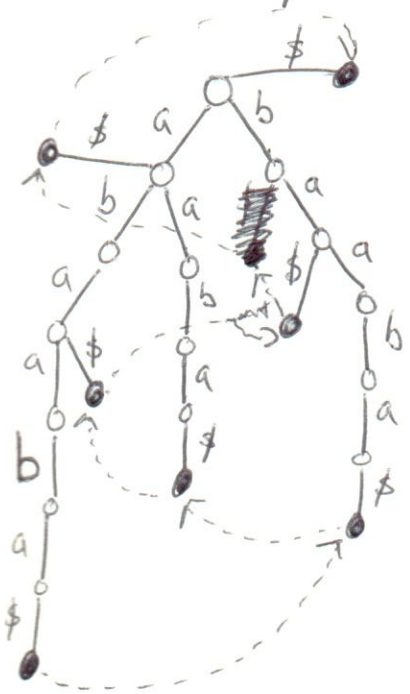
With new extension phase we have $O(m^2)$ algorithm.

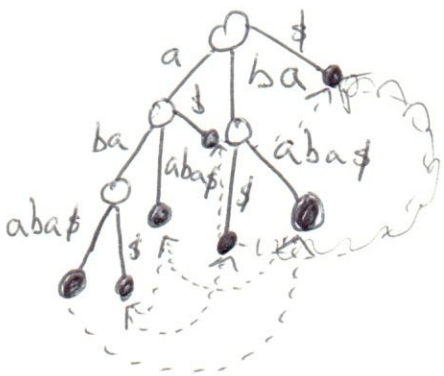
Suffix Tree:

The suffix trie above can be of size $O(n^2)$ in worst case (e.g.) $S = a^n b^n$. Can we find $O(n)$ space data structure?

1) Collapse paths with no choices

$S = abaaba\$$



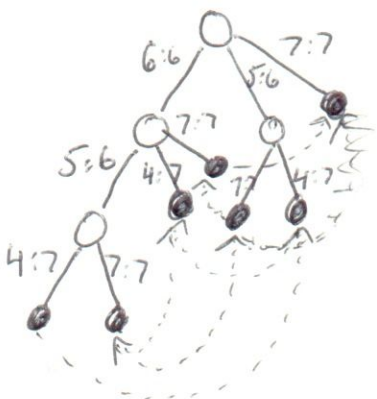


nodes ≈ # leaves

but space to represent edge labels varies

2) Label edges using index ranges: $O(1)$

$S = abaaba\$$
$1\ 2\ 3\ 4\ 5\ 6\ 7$



nodes ≈ # leaves = m

space for edges is constant $\Rightarrow$

$O(m)$ to store suffix tree

Constructing Suffix Tree (Ukkonen's Algorithm)

- Same idea as suffix trie construction, but some nodes of suffix trie may not be represented explicitly in suffix tree.
- Represent trie nodes as pair (v, α) where v is a node in suffix tree and α is part of string labeling exiting edge.

$S = abab$

suffix_link(u) = (v, ab)

- Additional tricks get $O(m)$ construction (see reading)