Lecture 8

Two approaches for string search (given P and T)

- (search func)
  1. Preprocess P in $O(m)$ time
  2. Search T in $O(n)$ time
- (index data structure)
  1. Preprocess T in $O(n)$ time
  2. Search for P in $O(m)$ time

The data structure constructed by the Aho-Corasick algorithm is a tree, but with edges labeled by symbols (characters), and the children of a node have distinct labels. It is called a TRIE (derived from information reTRIEval).

Suffix Trees

A suffix tree T_S is defined on a string S. We put a special symbol # (which is not in the alphabet) at the end of S so that a suffix of S may not be a prefix of another suffix. Let $n = |S\#|$.

Conceptionally easy definition (but this is not the way we construct the suffix tree).

1. Build the trie with all the suffixes of S. (the number of nodes is $O(n^2)$)
2. Remove every node which has a single child, and concatenate the labels. This is called a *compacted trie*.

Example: ababa#

- Since # is not in the alphabet, all the suffixes of S are distinct and each of them is associated with a leaf of T_S.
- The number of leaves is n.
- Each internal node has degree at least two.
- The number of internal nodes $< n$.
- A label is a nonempty substring of S, and it is represented by the start and end positions of AN occurrence (usually leftmost) of the substring.

Let $L(v)$ for a node v be the string obtained by concatenating the labels on the path from the root to v.

Linear Time Construction

As in the AC algorithm, put suffixes into the suffix tree from longest to shortest. But we are dealing with a compacted trie, not a trie.

The suffix tree defined by McCreight [Mc76] has one more piece of information: Each internal node u such that $L(u) = a\alpha$, a a character and α a string, has a *suffix link* $SL(u)$ pointing to the node w such that $L(w) = \alpha$.

The *locus* of a string α in the suffix tree T_S is the node v, if any, such that $L(v) = \alpha$.

Define $head_i$ to be the longest prefix of $S[i..n]$ which is also a prefix of $S[j..n]$ for some $j < i$. The locus of $head_i$ always exists.

Lemma 1 *If $head_{i-1} = a\alpha$ for some character a and some (possibly empty) string α, then α is a prefix of $head_i$.*

At the beginning of stage i, each suffix $S[j..n]$, $j < i$, is in the tree and we insert $S[i..n]$ and return the locus of $head_i$ at stage i.

Invariant: After stage i, the locus of $head_i$ is the only node that could fail to have a suffix link.

General stage: Let v be the locus of $head_{i-1}$.

B. $x \leftarrow SL(parent(v))$. Let β be the label of edge $(parent(v), v)$.

C. (Construct the suffix link of $head_{i-1}$ if it does not exist already.) By Lemma 1, starting from node x, there is a path that has β as prefix. That path is traversed as follows. Set $\hat{\beta} \leftarrow \beta$. Let α be the label of the edge from x to its child f such that the first characters of α and $\hat{\beta}$ are equal. If $|\alpha| < |\hat{\beta}|$, set $\hat{\beta} \leftarrow \hat{\beta} - \alpha$ and $x \leftarrow f$ and repeat the label selection with the new values of $\hat{\beta}$ and x until $|\alpha| \geq |\hat{\beta}|$.

   1. If $|\alpha| > |\hat{\beta}|$, create an internal node d such that $L(d) = head_{i-1} - S[i-1]$. Set $SL(v) \leftarrow d$. Create a leaf w

such that $L(w) = S[i..n]$, as a child of d. Stop and return d as the locus of $head_i$.

   2. If $|\alpha| = |\hat{\beta}|$, f is the locus of $head_{i-1} - S[i-1]$. Set $SL(v) \leftarrow f$; $y \leftarrow f$. Go to Step D.

D. (Construct the locus of $head_i$.) By Lemma 1, $head_i = L(y) \cdot \gamma$, for some possibly empty string γ. Therefore, we can start the search from y. The search is guided by the characters of $S[i..n] - L(y)$ which are scanned one by one from left to right. When the search falls out of the tree, create an internal node v such that $L(v) = head_i$, if one does not exist. Create a leaf w such that $L(w) = S[i..n]$, as a child of v. Return v as the locus of $head_i$.

Theorem 1 *Given a string $S[1..n] = a_1 a_2 \cdots a_{n-1}\#$, the suffix tree for S can be correctly built in $O(n)$ time.*

Proof. correctness: invariant

Time: each stage takes constant time except for Steps C and D.

Step D: The number of characters that must be scanned during stage i to locate $head_i$ is given by $|head_i| - |head_{i-1}| + 1$. The sum of such terms, taken over all stages is bounded by n, since $head_1 = head_n$ is empty.

Step C: Let res_i be $S[i..n] - L(x)$, the suffix of $S[i..n]$ starting from node x. Notice that for every node f encountered during Step C, there is a nonempty string α which is contained in res_i but not in res_{i+1}. Therefore, the number of nodes visited during Step C of stage i is at most the number of nodes from x to the parent of $head_i$ which is $\leq |res_i| - |res_{i+1}| + 1$. The total time over all steps is bounded by $2n$, since $res_1 = n$ and $res_n = 1$. $\square$

Applications of Suffix Tree

String matching (index approach)

1. Compute the suffix tree of T.
2. Search down the suffix tree with P.
3. P is a prefix of $L(v)$ for some node v iff P is a substring of T.
   - Existence test (leftmost occurrence)
   - All occurrences: Find all leaves of the subtree rooted at v.

Lowest Common Ancestor

The LCA problem: A tree is given.

1. Preprocess the tree in linear time
2. Query: for any two nodes, find their LCA. The query can be done in constant time [Harel and Tarjan, Schieber and Vishkin].

With the LCA preprocessing on a suffix tree, the LCA of two suffixes (leaves) can be found in constant time.

Back to approximate string matching

Problem: for a pattern position i and text position j, find in constant time how many matches there are from $P[i]$ and $T[j]$.

Solution:

1. Construct the suffix tree for $P\#T\$$.
2. Find the LCA of the suffixes starting at $P[i]$ and $T[j]$.

Therefore, in the k-mismatches and k-differences problems (recall $m \leq n$), we have $O(kn + |\Sigma|n)$ time.

Suffix Arrays

Suffix array of T: sorted list of all suffixes of T

Suffix array of $ababa\#$

	Pos	Suffix
1	6	#
2	5	a#
3	3	aba#
4	1	ababa#
5	4	ba#
6	2	baba#

More space efficient than suffix trees

```
suffix tree  <----> suffix array
                O(n)
```

Direct construction of suffix arrays

- $O(n \log n)$: Manber-Myers, Gusfield
- $O(n)$: Kim-Sim-Park-Park, Ko-Aluru, Karkkainen-Sanders

Pattern search with suffix arrays

- $O(m + \log n)$: Manber-Myers
- $O(m \cdot |\Sigma|)$: Abouelhoda-Kurtz-Ohlebusch
- $O(m \log |\Sigma|)$: Kim-Jeon-Park