Algorithms on Strings

An **alphabet** is a set of symbols, a **string** is a finite sequence of symbols in an alphabet, $\Sigma^\star$ built on the alphabet Σ, it contains also the empty string ϵ.

If $x \in \Sigma^\star$ and $y \in \Sigma^\star$ then their **concatenation** $xy \in \Sigma^\star$. Consider $y = xw \in \Sigma^\star$, then x is called *prefix* of y, denoted by $x \sqsubset y$, and w is called *suffix* of y, denoted by $w \sqsupset y$. if $x \in \Sigma^\star$ and $q \in \mathbb{N}$, x_q will be the x's prefix of length q.

Lemma (overlapping suffix lemma): Let x, y, w st. $x \sqsubset w$ and $y \sqsubset w$ then:

- if $|x| > |y|$, then $y \sqsubset x$;
- if $|x| = |y|$, then $y = x$.

Given a finite alphabet Σ, a *text* $T[1, .., n]$ and a *pattern* $P[1, .., m]$ with $m \leq n$, then we say that P *occurs with shift* s *in* T if $T[s+1, ..., s+m] = P$. If P occurs with shift s in T, then s is called **valid shift**„ otherwise it is an invalid shift.

The **string matching problem** requires to find all the valid shifts for P in T. A naive solution would be to try all possible shifts for P in T, the overall complexity of the naive algorithm is $O(|P| \cdot |T|)$.

Knuth-Morris-Pratt Algorithm

We can define the prefix function for P as $\pi[q] = max\{k : k < q \text{ and } P_k \sqsupset P_q\}$, namely $\pi[q]$ is the longest prefix of P that is a proper suffix of P_q (P_q is the q-character prefix of the pattern P).

The prefix function encapsulates knowledge about how the pattern matches against shifts or itself. We can take advantage of this information to avoid testing useless shifts int the naive pattern matching algorithm.

In order to compute the prefix function, consider the following results (which prove the correctness of the Knuth-Morris-Pratt algorithm): let $\pi^\star[q]$ be $\{\pi[q], \pi^2[q], ..., \pi^{(t)}[q]\}$:

Lemma (prefix-function iteration lemma): $\pi^\star[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$;

Lemma: if $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^\star[q-1]$;

Let $E_q = \{k \in \pi^\star[q] : P[k+1] = P[q+1]\}$, then

Theorem: $\pi[q] = 0$ if $E_{q-1} =$, $1 + max\{k \in E_{q-1}\}$ otherwise.

The complexity of computing the prefix function is $\Theta(|P)$.

Use the prefix function to try to fix mismatch, and we keep applying it until we end up in a situation in which we have a match. This is known as **Knuth-Morris-Pratt** algorithm. Its overall asymptotic complexity is $\Theta(|P| + |T|)$.

The Boyer-Moore-Galil Algorithm

Now we try to match the pattern backward.

Consider the **good suffix rule**: if $P[1,..,|P|] = T[i+j,..,|P|+j]$ (match for the suffix of P), and $P[i-1] \neq T[i+j-1]$

- align $T[i+j,...,|P|+j]$ to its rightmost occurrence in P with a preceding character $\neq P[i-1]$;
- it is doesn't exist, align the longest $P_q \sqsubset P$ to $T[|P|+j-q,..,|P|+j]$.

The good suffix is almost like π^{-1} on the reversed pattern P^{-1}, but $P^{-1}[q+1] \neq P^{-1}[\pi[q]+1]$ must be guaranteed. We can guess a complexity of $\Theta(|P|)$ to compute it.

Now consider the **bad-character rule**: if $P[i] \neq T[i+j]$

- align $T[i+j]$ to its rightmost occurrence in P;
- if it doesn't exist, align $P[1]$ to $T[i+j+1]$.

We can compute the bad-character rule as:

1. initialize an array C s.t. $|C| = |\Sigma|$;
2. $C[a] \leftarrow |P| \ \forall a \in \Sigma$;
3. $C[P[i]] \leftarrow |P| - i \ \forall i \in [1,..,|P|]$.

The complexity is $\Theta(|P| + |\Sigma|)$.

The **Galil's rule**: if a valid match has been discovered and P is k-periodic, P is shifted forward by k and $|P| - k$ comparisons are avoided.

The **Boyer-Moore-Galil's algorithm** consists in:

- try to match P on T backward;
- if a mismatch is found, then select the largest shift among those suggested by the good-suffix and bad-character rules;
- if a valid shift is found, apply the Galil's rules or revert to the mismatch case.

The overall asymptotic complexity is $O(|P| + |T|)$ (average-case scenario is sub-linear wrt |T|).

Multiple patterns string matching

We have a text T and a large set of patterns $P = \{P_1, ..., P_l\}$. Our goal is to find a valid shift for each P_i.

A naive solution would be to apply Boyer-Moore-Galil's algorithm to each P_i, but that would cost $O(|T| \cdot l + \sum_{i=1}^{l} |P_i|)$

Suffix tries

We can think about a tree-based solution: searching for P_i in the T's substring costs $\Theta(|P_i|)$„ hence solving the problem costs $\Theta(\sum_{i=1}^{l})|P_i|$.

Formal definition of **suffix tries**: let $\sigma(T)$ be the set of all the substrings of T. *STrie(T)* of T is tuple $(Q \cup \{\bot\}, \bar{\epsilon}, L, g, f)$ where:

- $Q = \{\bar{x} | x \in \sigma(T)\}$;
- $\bot \notin Q$;
- $L: \ Q \to [1, ..., |T|]$ is the *shift label*;
- $g: \ (Q \cup \{\bot\}) \times \Sigma \to Q$ is the *transition function*, such that: $g(\bar{x}, a) = \bar{xa}$ $\forall xa \in \sigma(T)$ and $g(\bot, a) = \bar{\epsilon}$ $\forall a \in \Sigma$;
- $f: \ Q \to Q \cup \{\bot\}$ is the *suffix function*, such that: $f(\bar{ax}) = \bar{x}$ $\forall ax \in \sigma(T)$ and $f(\bar{\epsilon}) = \bot$.

Practically we are growing trees by appending characters.

Let T^i be $T[1, .., i]$, than the **boundary path** of $Strie(T^i)$ os the sequence $\bar{T}^i = s_1, s_2, ..., s_{i+1} = \bot$ where $s_k = f^k(\bar{T}^i)$. The **active point** is the first s_j that is not a leaf, the **end point** is the first $s_{j'}$ having a $T[i+1] - transition$.

In order to build a suffix tree, it is necessary to:

- add a $T[i+1]$-transition from s_h $\forall h \in [1, j'-1]$;
- if $h \in [1, j-1]$, then it extends a branch;
- if $h \in [j, j'-1]$, then it creates a new branch.

In order to compute the complexity of this procedure, we must take into account the fat that each node is visited at most twice, there are constant steps per node and moreover $|Q| = |\sigma(T)|$. Building $STrie(T)$ costs $\Theta(\sigma(T))$ and it holds:

Lemma: $\sigma(T) \in O(|T|^2)$.

Theorem: buildind $STrie(T)$ costs $O(|T|^2)$.

We need to notice that suffix tries are redundant.