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 $y \sqsubset w$ (we have two suffixes for w) then:

- if $|x| > |y|$, then $y \sqsubset x$ (second string is the final part of the first one);
- 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 $O(|P| \cdot |T|)$.

A better idea could be to find the shift which matches as much as we already matched in the previous one, up to the last match we did. We are searching for the largest prefix of the already matched string which is also a suffix for it. In this way we avoid to waste the time we already spent in matching the first part, we already know that it is done correctly.

Knuth-Morris-Pratt Algorithm

We can define the prefix function for P as $\pi[q] = max\{k : k < q \ and \ P_k \sqsupset P_q\}$, 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 $\{\pi[q], \pi^2[q], ..., \pi^{(t)}[q]\}$ (set of subsequent iteration of π, up to the empty string).

Lemma (prefix-function iteration lemma): $\pi^\star[q] = \{k : k < q \ and \ P_k \sqsupset P_q\}$ (all the proper prefixes which are also suffixes);

Lemma: if $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^\star[q-1]$ (need to investigate all elements in $\pi^\star$, but this is the idea);

Let $E_q = \{k \in \pi^\star[q] : P[k+1] = P[q+1]\}$ (largest prefix that is extendable), 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, from right to the left.

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 $T[|P| + j - q, .., |P| + j]$ (completely shift the pattern outside the mismatch).

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** (this is the reason why we move from left to right): if $P[i] \neq T[i + j]$

- align $T[i + j]$ to its rightmost occurrence in P (try to match the mismatch with the rightmost occurrence of that character in the pattern);
- if it doesn't exist, align $P[1]$ to $T[i + j + 1]$ (we move forward and skip that character, we skip a lot of character on the text, this brings to sub-linear execution time).

We can compute the bad-character rule as:

1. initialize an array C s.t. $|C| = |\Sigma|$ this takes $\Theta(|\Sigma|)$;
2. $C[a] \leftarrow |P|$ $\forall a \in \Sigma$ this takes $\Theta(|\Sigma|)$;
3. $C[P[i]] \leftarrow |P| - i$ $\forall i \in [1, .., |P|]$ (update the shift once we discovered a mismatch) this takes $\Theta(|P|)$.

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

The **Galil's rule** (how to behave when we discover a valid shift): if a valid match has been discovered and P is k-periodic (repeated pattern inside the pattern itself), P is shifted forward by k (period of the pattern) 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 (these two rules suggest two different shifts, we take the largest among the two);
- 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 $\Theta(|P_i|)$ (complexity of reading the pattern), hence solving the problem costs $\Theta(\sum_{i=1}^{l})|P_i|$ (way better than the naive solution, the part of the complexity dealing with the length of the tree is discarded by the tree we build). We need to build the tree ,but this is built once and for all, not once for every pattern. If we cannot find the pattern, then the pattern is not present in the text.

Our goal is to understand the cost of building the tree.

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

- $Q = \{\bar{x} | x \in \sigma(T)\}$ (each substring in T has a node corresponding to it);
- $\perp \notin Q$;
- labelling function $L: \ Q \to [1, ..., |T|]$ is the *shift label* (this function tells for each substring what is the shift of that substring in the text, assume for now the first occurrence);
- transition function $g: \ (Q \cup \{\perp\}) \times \Sigma \to Q$ is the *transition function* (basically it is the set of the edges and the corresponding labels), such that: $g(\bar{x}, a) = \bar{xa} \ \forall xa \in \sigma(T)$ and $g(\perp, a) = \bar{\epsilon} \ \forall a \in \Sigma$ (this node can reach the empty string by "consuming" each character);
- $f: \ Q \to Q \cup \{\perp\}$ is the *suffix function* (move from text to the suffix), such that: $f(\bar{ax}) = \bar{x} \ \forall ax \in \sigma(T)$ and $f(\bar{\epsilon}) = \perp$ (bottom). By this function we can move to the largest suffix of the substring (edges going from the bottom up of the tree).

Practically we are growing tries by appending characters (the root corresponds to the ϵ substring). Labels in the nodes basically represent the shift.

The idea is to start from the node representing the largest sequence in the tree, add anew node and keep moving along the suffix function until we end up in a situation in which we already have the node we need.

Let T^i be $T[1,..,i]$, than the **boundary path** of $Strie(T^i)$ os the sequence $\bar{T}^i = s_1, s_2, ..., s_{i+1} = \perp$ where $s_k = f^k(\bar{T}^i)$ (keep applying the prefix function from the longest path in the tree). The **active point** is the first s_j (in the boundary path) that is a leaf, the **end point** is the first $s_{j'}$ having a $T[i+1] - transition$ (from the character we are trying to insert in our text).

In order to build a suffix tree, it is necessary to (start from the active point):

- 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: building $STrie(T)$ costs $O(|T|^2)$.

This complexity is not due to the procedure (which is linear). It is not physible if we consider string with billions of characters (like genomes).

We need to notice that suffix tries are redundant. For example when we have only a way to read a substring, some nodes are useless. We can remove them removing also the suffix and transition function passing through these nodes. So we need to update also the definition of these functions.

So we end up having:

- Q' containing branching nodes Q_b (useful to browse nodes along the tree) and Q_l (splitting set of all the nodes);
- $g' : ((Q_b \cup \{\perp\}) \times \Sigma^\star) \leftarrow Q'$ (deals exclusively with branching nodes and bottom, mpve along the transition function depending on a substring of the text);
- $f' : Q_b \leftarrow Q_b$ (not dealing with leaves).

We want to remove all internal nodes which are not branching. We want to build the tree with an algorithm which is linear wrt the length of the text.

All the leaves are suffixes for T, we have at most $|T|$ leaves. All the internal nodes are branching, since the ariety of the branch is 2, we have at most $|T| - 1$ internal nodes. So we are guaranteed that these trees have $‰(|T|)$ nodes. These are called *suffix trees*.

The problem is now that labels are substrings, so we are wasting up to quadratic time in reading these strings. And we need a quadratic space wrt to the original tree in representing all the labels of the transition function.

We can deal with this problem by representing strings using intervals: meaning that we have two indexes, the first representing the first character in the text for that substring, and the second representing the index in the text for the last character of the substring. This reduce the space required to represent the labels of the transition function from the length of the string to a pair of numbers (and it is fixed).

We can also avoid the labelling function L, we can deduce the shift by looking at the last matching label.

A pair of nodes plus a substring allow us to pin point to a non branching node which is not present in the Suffix Tree but was present in the suffix Trie. So we can represent *implicit nodes* by *reference pairs* (explicit node, substring). The same thing can be done for explicit nodes by using the empty substring.

It may be the case that we want to address implicit nodes that are not closest ancestors of the nodes we are in (there are several ways to address an implicit node). If the node we use to address an implicit node is the closest ancestor, the representation is called *canonical* (x, w) where x is the closest ancestor.

Now we want to see how we extend branches in the Suffix Tree. We want to avoid to update at each step the last index of the transition, we may simply relabel the second index with ∞.
We avoid to update all nodes in the boundary path which are leaves. So branch extension is avoided by labelling transition to leaf as $[h, \infty]$.

If we need to branch an internal node (we are adding a branching node), it could also be implicit. That node should become explicit (we are breaking a transition into two sub-transitions), we can do it if we have its canonical ancestor. Consider that s_j has a canonical reference pair $(s, [k, , i])$, if the node is implicit it should become explicit. If s_j is explicit, add a new branch labeled $[i + 1, \infty]$. The closest ancestor of the newly added node, for sure wasn't a leaf at the previous step, so it has the suffix link, and we can compute its predecessor using the suffix function. Now we see why it is useful to have the link from to the bottom node labeled with every character in the alphabet. In this way we can find the successors and ancestors.

We also need to update the suffix node for the new internal branching node.

We can have that the following the boundary path we have a non-canonical address. To be able to perform all the operations, we need to be able to deal with canonical representations, otherwise we don't know how to break the transition.