String Algorihtms III

Suffix Array

Introduction

Suppose S is a string of length n. A suffix of S is of the form $S[i, n]$, for $1 \leq i \leq n$, and we denote the suffix $S[i, n]$ as S_i. Observe that there are n unique suffixes of S.

We are interested in *sorting* the suffixes in terms of the lexicographic order. In particular, we are focusing on *suffix arrays*.

Suffix array is an integer array SA of length n, such that $\mathrm{SA}[i] = j$ iff S_j is of rank i when sorting S_i's in increasing lexicographic order. For example, $S = aabba$, then the sorted suffixes are a, $aabba$, $abba$, ba, bba, and therefore $\mathrm{SA}[1] = 5$, $\mathrm{SA}[2] = 1$, $\mathrm{SA}[3] = 2$, $\mathrm{SA}[4] = 4$, $\mathrm{SA}[5] = 3$.

Construction

The naive approach that sorts the n suffixes directly will take $O(n^2 \log n)$ time, which is inefficient. Fortunately, there are simple and efficient algorithms for construting suffix arrays.

In paper [MM93], the suffix array was first introduced, and a $O(n \log n)$ algorithm is presented. This time complexity is already sufficient enough for most ACM/ICPC problems. We will briefly introduce the $O(n \log n)$ algorithm.

A simple linear time algorithm is presented in [KS03], and this is one of the most popular algorithm used practically. Please study the paper by yourself. (There is also a ready to use C language implementation in the appendix of the paper).

Height Array

We usually do not use suffix array itself alone. Instead, based on the suffix array, we can construct other structures that are even more powerful.

The *height array* is one of the most important one. For example, with the height array, one can compute the *longest common prefix* (LCP) of any two suffixes efficiently.

Height Array, LCP

The height array H of S is an integer array of length n, such that $H[i] = j$ iff the length of the LCP between $\mathrm{SA}[i]$ and $\mathrm{SA}[i-1]$ is j for $i > 1$, and $H[i] = 0$ for $i = 1$. (We abuse the notation of $\mathrm{SA}[i]$ to denote $S_{\mathrm{SA}[i]}$.)

Suppose $1 \leq i < j \leq n$, and we would like to compute the length of the LCP of $\mathrm{SA}[i]$ and $\mathrm{SA}[j]$. Since $\mathrm{SA}[i], \mathrm{SA}[i+1], \ldots, \mathrm{SA}[j]$ are sorted lexicographically, any common prefix of $\mathrm{SA}[i]$ and $\mathrm{SA}[j]$ is a prefix of $\mathrm{SA}[k]$ for $i \leq k \leq j$. Therefore,
$\mathrm{LCP}(\mathrm{SA}[i], \mathrm{SA}[j]) = \min_{i+1 \leq k \leq j}\{H[k]\}$.

LCP Using Height Array

The computation of $\min_{i+1 \le k \le j}\{H[k]\}$ can be seen as finding the minimum in a continuous range of H. This can be efficiently done by $O(n \log n)$ preprocessing, and then compute $\min_{i+1 \le k \le j}\{H[k]\}$ in $O(1)$ time (using ST algorithm).

Therefore, to compute the LCP of S_i and S_j, we first compute $i' = \mathrm{SA}^{-1}[i]$ and $j' = \mathrm{SA}^{-1}[j]$. Note that given SA, we can compute SA^{-1} in linear time. Then the LCP is $\min_{i'+1 \le k \le j'}\{H[k]\}$, which can be computed efficiently by using RMQ algorithms.

Longest Repeating Substring

We look at another problem that height array can be applied. A substring S' of S is reapeating, if S' appears at least twice in S. The two occurences can partially overlap, but cannot totally. Our goal is to calculate the length of the longest repeating substring.

With the help of the height array, this problem turns to be easy. We just find i such that $H[i]$ is maximized, and such $H[i]$ is the answer.

Longest Common Substring

We generalize the longest repeating substring problem to two strings. That is, we would like to calculate the longest common substring between two strings S and T.

Suppose $|S| = n$ and $|T| = m$. To apply suffix array, we build a new string $P = S0T$, where 0 is a character that is less than any character in the alphabet. We calculate the suffix array and the height array for P. Note that the suffixes of P are of the form $S[i, n]T$, and $T[j, m]$.

We explain why we add an 0 between S and T. Recall that the desired property of height array is for $i \in [1, n]$, $j \in [n+1, m+n]$, we have $\mathrm{LCP}(S_i, T_j) = \min_{i'+1 \le k \le j'}\{H[k]\}$ where $i' = \mathrm{SA}^{-1}[i], j' = \mathrm{SA}^{-1}[j]$. It is easy to check that adding an 0 can guarantee this property[1].

The algorithm is simple. Define $\max = 0$. We scan for $i = 1$ to $m + n - 1$. If $\mathrm{SA}^{-1}[i] \in [1, n]$ and $\mathrm{SA}^{-1}[i+1] \in [n+1, m+n]$, or if $\mathrm{SA}^{-1}[i] \in [n+1, m+n]$ and $\mathrm{SA}^{-1}[i+1] \in [1, n]$, and if $H[i+1] > \max$, we update $\max = H[i+1]$.

[1] Without adding 0, the counter example is $S = a$ and $T = aa$, $i = 1$ and $j = 2$

Longest Palindrome Substring

Another related problem is to find the longest palindrome substring of S. We do not cover the whole detail, but sketch the main steps.

The basic idea is to accelarate the following naive approach. We handle the longest odd length and even length palindrome substring separately. Since they are similar, we only look at the longest odd length palindrome. For each i, we find the longest odd length palindrome centered at i.

Fix some i, to find the longest odd length palindrome centered at i, we find the LCP of the suffix S_i, and suffix S'_i, where S'_i is suffx in the reverse of S that starts at the corresponding character of $S[i]$. To calculate LCP efficiently, we build $P = S0S'$, and apply suffix array techniques on P, where S' is the reverse of S.

Non-overlapping Longest Repeating Substring

We consider the generalized longest repeating substring problem, by putting the restriction that the substrings cannot overlap.

Observe that we can binary search the answer. That is, if length l is feasible (by feasible, we mean there exists non-overlapping repeating substrings), then so is $l' < l$.

The problem then reduces to check whether there exists non-overlapping repeating substring of length l. Since $\text{LCP}(\text{SA}[i], \text{SA}[j]) = \min_{i+1 \leq k \leq j}\{H[k]\}$, we consider each of the maximal continuous intervals $[p, q]$ such that $H[k] \geq l$ for $p \leq k \leq q$. If l is feasible, then there exists k_1, k_2 and an interval, such that $|\text{SA}^{-1}[k_1] - \text{SA}^{-1}[k+2]| \geq l$. This can be checked in linear time, and we conclude that the checking process only requires linear time.

Key Substrings

Please find the statement at
`http://acm.timus.ru/problem.aspx?space=1&num=1713`.

Denote the i-th keyword by K_i, and its length by n_i. We start by concatenating the keywrods together and calculate the suffix array as well as the height array.
Our approach is for each keyword K_i, for each $1 \leq j \leq n_i$, we calculate the minimum prefix of $K_i[j, n_i]$ such that it is not a substring of other keywords. To do so, we calculate the maximum prefix of $K_i[j, n_i]$ such that it is a substring of some other keywords.

Suppose $K_i[j, n_i]$ ranks j'. For some t, we define $[p_t, q_t]$ to be the maximal continuous interval such that $p_t \leq j' \leq q_t$ and $H[k] \geq t$ for all $k \in [p_t, q_t]$. We find the smallest t such that at least one suffix of each keyword presents in the interval $[p_t, q_t]$ (the interval is in terms of the index of the sorted suffixes).

Count the Number of (Different) Substrings

Another application is to counter the number of *different* substrings of a given string S.

Consider the suffixes in the order given by the suffix array, i.e., $\mathrm{SA}[1], \mathrm{SA}[2], \ldots, \mathrm{SA}[n]$. Define c_i to be the number of substrings that start at $\mathrm{SA}[i]$, and do not equal to any substrings that start at $\mathrm{SA}[j]$ for $1 \leq j < i$. Observe that $\sum_{i=1}^{n} c_i$ is what we want.

Calculating c_i

It remains to calculate c_i.

Note that there are $|\mathrm{SA}[i]|$ different substrings that starts at $\mathrm{SA}[i]$. Since any prefix of $\mathrm{SA}[i]$ that is of length at most $H[i]$ is also a prefix of $\mathrm{SA}[i-1]$, we cannot count them in c_i. On the other hand, substrings starting at $\mathrm{SA}[i]$ of length larger than $H[i]$ are not counted in $c_1, \ldots, c_{i-1}$. Therefore, $c_i = |\mathrm{SA}[i]| - H[i]$.

Hence, $\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} |\mathrm{SA}[i]| - \sum_{i=1}^{n} H[i] = \binom{n}{2} - \sum_{i=1}^{n} H[i]$.

Construct the Height Array

We turn to construct the height array. The desired result is that given the suffix array, we can construct the height array in linear time. The algorithm is as follows.

Construct the Height Array

Let $h = 0$
for $i = 1 \rightarrow n$ **do**
  if $SA^{-1}[i] = 1$ **then**
    $H[\mathrm{SA}^{-1}[i]] = 0$
  end
  else
    while $S[SA[SA^{-1}[i] - 1] + h] = S[i + h]$ **do**
      $h = h + 1$
    end
    $H[\mathrm{SA}^{-1}[i]] = h$
    if $h > 0$ **then**
      $h = h - 1$
    end
  end
end

Algorithm 1: Construct Height Array

Analysis

The time complexity is linear. To see it, we analyze the number of iterations in the while loop. Observe that this number is equal to the number of $h = h + 1$ operations. Since $h \leq n$, and $h = 0$ initially, and in each for loop, h decreases at most one, we conclude that $h = h + 1$ operation can only perform at most $2n$ times.

We turn to the correctness. Observe that we are constructing the height array in the increasing order of the length of the suffixes. Since $h = 0$ initially, we get the correct answer for the first i that $\mathrm{SA}^{-1}[i] > 1$. Suppose the for loop just finishes $i = k$ and the H value is still correct. We look at $i = k + 1$, and assume $\mathrm{SA}^{-1}[k + 1] \neq 1$.

The key observation is that $H[\mathrm{SA}^{-1}[k]] - 1 \leq H[\mathrm{SA}^{-1}[k+1]]$. To see why it holds, we suppose the LCP between S_k and $S_{\mathrm{SA}[\mathrm{SA}^{-1}[k]-1]}$ is T, where $|T| = H[\mathrm{SA}^{-1}[k]]$. Then, $T[2, H[\mathrm{SA}^{-1}[k]]]$ is a prefix of S_{k+1}. Moreover, $T[2, H[\mathrm{SA}^{-1}[k]] \neq S_{k+1}$. Therefore, $H[k+1] \geq H[k] - 1$, and it is safe to set $h = h - 1$ in the last line of the for loop.

$O(n \log n)$ Suffix Array Construction

We introduce a $O(n \log n)$ suffix array construction algorithm. We use radix sort as a subroutine. For any string S and T, notation $S <_t T$ denotes that $S[1, \min\{t, |S|\}] < T[1, \min\{t, |T|\}]$. Similarly we define $=_t$ and $\leq_t$.

The algorithm is as follows. Initially, we sort the suffixes according to the first character of the suffixes. That is, we sort them wrt $<_1$. It is possible that there are many suffixes are equal wrt $=_1$, but they will be uniquely sorted wrt $<_n$.

After the initialization, we do the following. For $t = 0 \rightarrow \lfloor \log n \rfloor$, we sort the suffixes wrt $<_{2^{t+1}}$, by using the information of the ordering wrt $<_{2^t}$. We will elaborate more about this step.

Suppose the ordering wrt $<_{2^t}$ is calculated. We group the suffixes under $=_{2^t}$ (two suffixes are in the same group, if they are equal wrt $=_{2^t}$). Note that the grouping actually forms a partition. Also note that the relative order between groups wrt $<_{2^{t+1}}$ is already calculated, it remains to calculate the ordering wrt $<_{2^{t+1}}$ inside each group.

Fix group G. If $|G| = 1$, then no more work should be done. Otherwise, $|G| \geq 2$. In this case, every element in G is of length at least 2^t since otherwise they cannot be equal to each other.

Since elements in G are suffixes of S, the suffixes of the elements are also the suffixes of S. Motivated by this observation, for each $S' \in G$, we consider its suffix starting at $S'[2^t + 1]$, and define G' to be the set of those suffixes. Observe that the ordering of strings in G' wrt $<_{2^t}$ is already calculated, and we can then use a radix sort to sort strings in G' wrt the ordering defined by $<_{2^t}$.

In the end of the whole algorithm, we get the ordering wrt $<_{2^{\lceil \log n \rceil}}$, which is exactly the ordering defined by the suffix array.

Implementation Notes

When implement the algorithm, we append a 0 to the original string S, where 0 is some character that is smaller than any character in the original alphabet set.

An important property of appending 0 is that no suffix is a prefix of another suffix. This can tackle the special cases in the algorithm. For example, when defining G', string S' may be of length exactly 2^t, and the suffix starting at $S'[2^t + 1]$ may not be well defined. If we have 0 appended, this case can never happen.

References

Juha Kärkkäinen and Peter Sanders.
Simple linear work suffix array construction.
In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

Udi Manber and Gene Myers.
Suffix arrays: a new method for on-line string searches.
siam Journal on Computing, 22(5):935–948, 1993.