Suffix Arrays and SA-IS

Charlie Gunn

December 7 2018

1 Suffix Arrays

A suffix of a string is a substring from a starting index to the end of the string. We can identify a suffix by the starting index, so $suf(S, k)$ is the suffix starting at and including $S[k]$. A suffix array is an alphabetically sorted array of all suffixes of a string. We could use an $O(n \log n)$ sort with $O(n)$ string comparison, which computes the sorted suffix array in $O(n^2 \log n)$. All faster algorithms for suffix arrays use the fact that we are sorting related strings, not a list of n random strings.

There are algorithms that can compute a suffix array in $O(n)$ time; they are more complicated than what you will usually need, but that wont prevent me from lecturing on them!

2 Suffix Array Construction by Induced Sorting (SA-IS)

SA-IS is one of the simplest algorithms to construct a suffix array in $O(n)$ time, and relies heavily on induced sorting, hence the name.

2.1 Reducing the Problem

Let S be a string of n characters, represented by an array indexed by $[0..n-1]$. For presentation simplicity, S is supposed to be terminated by a sentinel \$, which is the unique lexicographically smallest character in S. Let $suf(S, i)$ be the suffix in S starting at $S[i]$ and running to the sentinel. A suffix $suf(S, i)$ is said to be S- or L-type if $suf(S, i) < suf(S, i+1)$ or $suf(S, i) > suf(S, i+1)$, respectively. The last suffix $suf(S, n-1)$ consisting of only the single character \$(the sentinel) is defined as S-type. Correspondingly, we can classify a character $S[i]$ to be S- or L-type if $suf(S, i)$ is S-or L-type, respectively. From the above definitions, we observe the following properties: (i) $S[i]$ is S-type if (i.1) $S[i] < S[i+1]$ or (i.2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is S-type; and (ii) $S[i]$ is L-type if (ii.1) $S[i] > S[i+1]$ or (ii.2) $S[i] = S[i+1]$ and $suf(S, i+1)$ is L-type. These properties suggest that by scanning S once from right to left, we can determine the type of each character in $O(1)$ time and fill out the type array t in $O(n)$ time. Further, we introduce two new concepts called LMS character and LMS-substring as following.

Definition 2.1 (LMS character): A character $S[i]$, $i \in [1, n-1]$, is called leftmost S-type (LMS) if $S[i]$ is S-type and $S[i-1]$ is L-type.

Definition 2.2 (LMS-substring): A LMS-substring is (i) a substring $S[i..j]$ with both $S[i]$ and $S[j]$ being LMS characters, and there is no other LMS character in the substring, for $i = j$; or (ii) the sentinel itself.

Intuitively, if we treat the LMS-substrings as basic blocks of the string, and if we can efficiently sort all the LMS-substrings, then we can use the order index of each LMS-substring as its name, and replace all the LMS-substrings in S by their names. As a result, S can be represented by a shorter string, denoted by S_1, thus the problem size can be reduced to facilitate solving the problem in a manner of divide-and-conquer. Now, we define the order for any two LMS-substrings.

Definition 2.3 (Substring Order): To determine the order of any two LMS-substrings, we compare their corresponding characters from left to right: for each pair of characters, we compare their lexicographical values first, and next their types if the two characters are of the same lexicographical value, where the S-type is of higher priority than the L-type.

From this order definition for LMS-substring, we see that two LMS-substrings can be of the same order index, i.e. the same name, iif they are equal in terms of lengths, characters and types. When $S[i] = S[j]$, we assign a higher priority to S-type because $suf(S, i) > suf(S, j)$ if $suf(S, i)$ is S-type and $suf(S, j)$ is L-type.

To sort all the LMS-substrings, we don't need extra physical space; instead, we simply maintain a pointer array P_1, which contains the pointers for all the LMS-substrings in S. This can be done by scanning S (or t) once from right to left in $O(n)$ time.

Definition 2.3 (Sample Pointer Array): P_1 is an array containing the pointers for all the LMS-substrings in S preserving their original positional order.

Suppose we have all the LMS-substrings sorted into the buckets in their lexicographical orders where all the LMS-substrings in a bucket are identical. Then we name each item of P_1 by the index of its bucket to produce a new string S_1. Here, we say two equal-size substrings $S[i..j]$ and $S[i..j]$ are identical iif $S[i + k] = S[i + k]$ and $t[i + k] = t[i + k]$, for $k \in [0, j - i]$

2.2 Inducing SA from SA_1

As defined, SA maintains the indexes of all the suffixes of S according to their lexicographical order. Trivially, we can see that in SA, the pointers for all the suffixes starting with a same character must span consecutively. Let's call a sub-array in SA for all the suffixes with a same first character as a bucket. Further, there must be no tie between any two suffixes sharing the identical character but of different types.

Therefore, in the same bucket, all the suffixes of the same type are clustered together, and the S-type suffixes are behind, i.e. to the right of the L-type suffixes. Hence, each bucket can be further split into two sub-buckets for the L- and S-type buckets respectively.

Further, when we say to put an item $SA_1[i]$ to its bucket in SA, it means that we put $P_1[SA_1[i]]$ to the bucket in SA for the suffix $suf(S, P_1[SA_1[i]])$ in S. With these notations, we describe our algorithm for inducing SA from SA_1 in linear time/space as below:

1. Find the end of each S-type bucket; put all the items of SA_1 into their corresponding S-type buckets in SA, with their relative orders unchanged as that in SA_1;

2. Find the head of each L-type bucket in SA; scan SA from head to end, for each item $SA[i]$, if $S[SA[i] - 1]$ is L-type, put $SA[i] - 1$ to the current head of the L-type bucket for $S[SA[i] - 1]$ and forward the current head one item to the right.

3. Find the end of each S-type bucket in SA; scan SA from end to head, for each item $SA[i]$, if $S[SA[i] - 1]$ is S-type, put $SA[i] - 1$ to the current end of the S-type bucket for $S[SA[i] - 1]$ and forward the current end one item to the left.

2.3 Induced Sorting Substrings

In the above discussion, we have safely assumed that how to sort substrings is not an issue. We discovered that this seemingly difficult problem can be easily solved by using induced sorting too, i.e. the induced-sorting idea originally used for inducing the order of suffixes SA from SA_1 can be extended to induce the order of substrings. Specifically, we only need to make a single change to the algorithm in the section 2.3 in order to efficiently sort all the variable-length LMS-substrings. This single change is to revise step 1 to:

1. Find the end of each S-type bucket; put all the LMS suffixes in S into their S-type buckets in SA according to their first characters.