SIMPLE ALGORITHM TO MAINTAIN DYNAMIC SUFFIX ARRAY FOR TEXT INDEXES

Dmitry Urbanovich, Pavel Ajtkulov
Udmurt State University
e-mail: hun10@yandex.ru, ajtkulov@gmail.com

Abstract

Dynamic suffix array is a suffix data structure that reflects various patterns in a mutable string. Dynamic suffix array is rather convenient for performing substring search queries over database indexes that are frequently modified. We are to introduce an $O(nlog^2n)$ algorithm that builds suffix array for any string and to show how to implement dynamic suffix array using this algorithm under certain constraints. We propose that this algorithm could be useful in real-life database applications.

Keywords: *dynamic extended suffix array, string matching, text index.*

1. INTRODUCTION

Suffix array is a permutation of all suffixes of a given string that sorts them lexicographically. Suffix array has numerous applications, especially in string matching [1], bioinformatics [10] and text compression [11].

Here is an example of suffix array built for the string “banana”:

Table 1. Suffixes’ numeration and suffix array

(0) a	(0) a
(1) na	(2) ana
(2) ana	(4) anana
(3) nana	(5) banana
(4) anana	(1) na
(5) banana	(3) nana

Left column shows how suffixes are numbered[*]. Right column shows how (0, 2, 4, 5, 1, 3) suffix array sorts the suffixes.

There are a lot of suffix array construction algorithms for immutable string [1, 6, 7]. None of them allow you to reflect changes in text efficiently. However, efficient algorithms for maintaining dynamic suffix structures exist [2, 4, 5].

In this paper we introduce an algorithm that maintains suffix array for string that can grow to the left only. One can perform any suffix array aware algorithm on this data structure, because it maintains LCP information as

---

[*] We number suffixes in reverse order. We do so because our approach builds suffix array by successive insertions of characters at the beginning of the string. It’s convenient to preserve old numbers as it clarifies what happen after insertion.

well. For example, you can use classical algorithm to find all occurrences of pattern in text [1]. However, there will be additional $O(\log n)$ overhead due to data structures used (see section 3.1 for details).

Also the algorithm can maintain suffix array for a string that consists of records (record is a substring which ends with special character that appears nowhere else in the record). Every record can be removed and new record can be inserted at the beginning of the string. This results in data structure that allows searching for any substring in all records. It's similar to suffix data structures on words (suffix arrays on words [8], suffix trees on words [9]) in sense that you can search inside records only, but, as opposed to structures on the words, our structure can find any substring, not only word-aligned one.

In next section, we describe basic ideas which background the algorithm. In "Details" section, we describe the algorithm itself.

2. BASIC IDEA

Basic idea is to restrict operations on string in such a way that only minimal changes in the suffix array will happen.

Table 2. Insertion of "s" letter at beginning of the "issippi" string

(0) i	(0) i
(3) ippi	(3) ippi
(6) issippi	(6) issippi
(1) pi	(1) pi
(2) ppi	(2) ppi
(4) sippi	(4) sippi
(5) ssippi	(7) sissippi
	(5) ssippi

One of these operations is insertion of single character at the beginning of the string. For example, consider we already built suffix array for string "issippi", and now we are inserting the character "s" at beginning (our string will become "sissippi").

As you can see, relative order of 0–6 suffixes isn't changed. The only thing happened is a single insertion of new suffix between 4 and 5. The same will happen in general case, which is described in "Single Insertion" section. Clearly, we can insert any number of characters by this way.

The second operation is a removal of a special substring called a "record". As we said before, record is a substring which ends with special character that appears nowhere else in the record. In fact, we can safely remove such a substring (see "Record Removal" section) without disturbing other suffixes' relative position.

Suffix array can be enhanced by LCP array maintenance. LCP array is an array which shows the length of longest common prefix for each pair of suffixes that have adjacent positions in suffix array. LCP array is very useful, especially for improving various search algorithms time complexity [1], and our algorithm can optionally maintain it.

3. DETAILS

3.1. Data Structure

As we need to perform efficient modifications in suffix array, we need a data structure that supports indexed access, arbitrary insertions, removals and range minimum query [3, 5], which is based on balanced tree (order-statistic tree [12]). It is also used to find position of suffix in suffix array, when suffix is given by its length. (This operation implements inverted suffix array.) All operations are performed within $O(log\ n)$.

3.2. Single Insertion

Insertion of single character at the beginning of the string is two-step. First, we find the place where to insert new suffix. Second, we make the insertion itself.

Clearly, inserting a character at the beginning doesn't change existing suffixes. Hence, their relative position will not be changed. And of course, we need to insert new suffix just because the string became one character larger.

In order to find the place where to insert new suffix, we perform a binary search over suffix array comparing some of suffixes with a new one (see "Comparator Implementation"). This step works in $O(log_2 n)$, where n is a length of string. It got additional logarithmic multiplier due to fact that comparator accesses suffix array and its inversion via data structure described above.

Because insertion is a separate step, $O(log^2 n)$ overall complexity isn't affected.

Additionally, if we want to maintain LCP array, we need to update at most two values in that array, because at most two pairs of adjacent suffixes have been changed (they are neighbors of the new suffix). See "LCP Calculation" for details.

3.3. Record Removal

To remove the record we just need to remove all the suffixes that begin in positions corresponding to the record. We don't need to change relative order of other suffixes.

First of all, that's so, because relative order of suffixes, which begin in positions of the same record, doesn't depend on other records: positions of left-most special character in such suffixes never coincide — that is, lexicographical order of such suffixes depends on characters of single record only.

However, suffix array may become inconsistent after record removal operation. Let's look at the example. Given a string "ac|bs|ac|b|", where "|" is a special character. We are to remove "bs|" record.

Table 3. Example of what happens after removal

<table>
<tr><td>(0) |</td><td>(0) |</td><td>(0) |</td></tr>
<tr><td>(5) |ac|b|</td><td></td><td></td></tr>
<tr><td>(2) |b|</td><td>(2) |b|</td><td>(5) |ac|b|</td></tr>
<tr><td>(8) |bs|ac|b|</td><td>(5) |ac|b|</td><td>(2) |b|</td></tr>
<tr><td>(4) ac|b|</td><td>(4) ac|b|</td><td>(7) ac|ac|b|</td></tr>
<tr><td>(10) ac|bs|ac|b|</td><td>(7) ac|ac|b|</td><td>(4) ac|b|</td></tr>
<tr><td>(1) b|</td><td>(1) b|</td><td>(1) b|</td></tr>
<tr><td>(7) bs|ac|b|</td><td></td><td></td></tr>
<tr><td>(3) c|b|</td><td>(3) c|b|</td><td>(6) c|ac|b|</td></tr>
<tr><td>(9) c|bs|ac|b|</td><td>(6) c|ac|b|</td><td>(3) c|b|</td></tr>
<tr><td>(6) s|ac|b|</td><td></td><td></td></tr>
</table>

First column shows the state of suffix array before removal. Second column shows the state after removal. Third column shows correct state of suffix array for modified string ("ac|ac|b|"). As you can see, 2–7 suffixes in second column are placed incorrectly. But it doesn't matter, because search queries never contain special character. (If we truncate all the suffixes after left-most special character, we will see that sorting is correct.)

To perform record removal within $O(mlog\ n)$, where m is a length of the record, we use the same data structure as described in previous section.

If we want to maintain LCP array, we need to update LCP value for each suffix that is lexicographically previous to the suffix to be removed. See "LCP Calculation" for details.

3.4. Comparator Implementation

Comparator lexicographically compares new suffix to be added against other suffixes. At first, it compares suffixes by their first letters. If letters are equal, then comparison reduces to comparison of suffixes that obtained by removing first letter of each of the suffixes to be compared. The order of reduced suffixes is obtained from the structure that maintains the suffix array: it can give us position of any given suffix in the suffix array and this position gives us relative order of suffixes.

Here's sample implementation of comparator:

```
bool Compare(int pos) {
    int idx = array[pos];
    char fstCh = Str[Str.Length - 1 - idx];
    if (fstCh == ch) {
        if (idx == 0) {
            return true;
        }
        int inverse = array.GetInverse(idx - 1);
        return posLongestSuffix > inverse;
    }
    return fstCh < ch;
}
```

Variable *pos* shows the index of suffix to be compared with new suffix, variable *ch* is a new character to be added. *Str* is a string, for which suffix array array is already built. *posLongestSuffix* is a position of the longest suffix in already built suffix array (it equals *array.GetInverse*(*Str.Length*-1)), *inverse* is a position of suffix reduced from suffix with index *pos* in the suffix array.

We need to find where suffix given by its position in original string is located in a suffix array. The structure that answers those queries is an inversion of suffix array. It's essentially based on the same data structure that supports efficient random access by index, insertions and deletions. This inversion, of course, should be updated to reflect move of suffixes in the dynamic suffix array.

3.5. LCP Calculation

When we insert new suffix, LCP value is calculated for the longest suffix and its neighbors. This value is calculated with the same as comparator's logic. If first letters of suffixes are different, then their LCP is empty. In case they are the same, their LCP will be greater than LCP of corresponding reduced suffixes by one (reduced suffix is a suffix obtained by removing its first letter).

Reduced suffixes may be non-adjacent in suffix array. In this case, their LCP equals $\min_{i \le k<j} LCP_k$, where i and j are positions of those suffixes [1]. Data structure, that implements our suffix array, can evaluate this minimum within $O(\log n)$ operations (dynamic range minimum query, DRMQ) [3, 5].

When we remove i suffix (where "i" is a position of suffix in a suffix array) then $LCP_{i-1}=\min(LCP_{i-1}, LCP_i)$ [1].

4. CONCLUSION

We presented an algorithm of suffix array construction by successive insertions at the beginning of the string. Also we presented how to remove special type of substrings. Construction of suffix array for n-character string requires $O(n\ log^2 n)$ operations. Removal of k-character substring requires $O(klog\ n)$ operations. We presented an algorithm that maintains auxiliary LCP array. The algorithm can be used to maintain text database index which supports insertion of new record, removal and replacing of existing ones.

The designed algorithm is simpler than other existing dynamic suffix array construction algorithm [2, 5].

REFERENCES

1. **Manber U., Mayers G.** Suffix arrays: a new method for on-line string searches // SIAM Journal on Computing. -1993. -No 22. -P. 953-948.
2. **Salson M., Lecroq T., Leonard M., Mouchard L.** Dynamic extended suffix arrays // Journal of Discrete Algorithms. -2010. -Vol. 8. -P.241-257.
3. **Shibuya T., Kurochkin I.** Match chaining algorithm for cDNA Mapping // Algorithms in Bioinformatics: Third International Workshop, Budapest, WABI, 2003.
4. **Russo L., Navarro G., Oliveira A.** Dynamic Fully-Compressed Suffix Trees // Proceedings of the 8th Latin American conference on Theoretical informatics, LNCS. -2008. -P. 362-373.
5. **Ajtkulov P**. Symbol array processing, UBS, 28 (2010), -P. 126–178.
6. **Pang Ko, Srinivas Aluru**, Space-efficient linear time construction of suffix arrays, Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, -P. 200-210, 2003.
7. **Dong Kyue Kim, Jeong Seop Sim, Heejin Park, Kunsoo Park**, Linear-time construction of suffix arrays, Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, -P. 186-199, 2003.
8. **P. Ferragina, J. Fischer**, Suffix arrays on words, In Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, volume 4580 of LNCS, 2007.
9. **A. Andersson, N. J. Larsson, K. Swanson**, Suffix Trees on Words, Algorithmica 23, 1999.
10. **D. Gusfield**. Algorithms on Strings, Trees, and Sequences. Cambridge University Press, 1997.
11. **M. Burrows and D. J. Wheeler**. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.
12. **Cormen T., Leiserson C., Rivest R.; Stein, Clifford** (2001). Introduction to Algorithms (second ed.). MIT Press and McGraw-Hill.