International Journal of Information Sciences and Techniques (IJIST) Vol.3, No.4, July 2013

Space Efficient Suffix Array Construction using Induced Sorting LMS Substrings

Rajesh. Yelchuri[1], Nagamalleswara Rao.N[2]

Department of Computer Science and Engineering, R.V.R & J.C College of Engg.
Chowdavaram, Guntur, Andhra Pradesh -522119,India
[1]rajesh.yelchuri@gmail.com
[2]nnmr_m@yahoo.com

ABSTRACT

This paper presents, an space efficient algorithm for linear time suffix array construction. The algorithm uses the techniques of divide-and-conquer, and recursion. What differentiates the proposed algorithm from the variable-length leftmost S-type (LMS) substrings is the efficient usage of the memory to construct the suffix array. The modified induced sorting algorithm for the variable-length LMS substrings uses efficient usage of the memory space than the existing variable length left most S-type(LMS) substrings algorithm

KEYWORDS

Divide and Conquer, Suffix Array.

1. INTRODUCTION

This document describes, the concept of suffix arrays was introduced by Manber and Myers in SODA'90 [4] and SICOMP'93 [3] as a space efficient alternative to suffix trees. It has been well recognized as a fundamental data structure, useful for a broad range of applications, for e.g., string search, data indexing, searching for patterns in DNA or protein sequences, data compression and also in Burrows-Wheeler transformation. For an n-character string, denoted by STR, its suffix array, denoted by SAR(STR), is an array of indices pointing to all the suffixes of STR, sorted according to their ascending(or descending) lexicographical order. The suffix array of STR itself requires only n[log n]-bit space. However, different suffix array construction algorithms may require different space and time complexities. During the past decade, a many researches have been developing suffix array construction algorithms that are both time and space efficient, for which we suggest a detailed survey from Puglisi [5]. Time and space efficient suffix array construction algorithms has become popular because of their wide usage. Construction of suffix arrays are needed for large scale applications, e.g., biological genome database and web searching and, where the size of a huge data set is measured in billions of characters [6], [7], [8], [9], [10].Time and space efficient linear time algorithms are crucial for large-scale applications to have predictable worst-case performance. The three known algorithms are KSP [1],KA [12], [13],KS [11], [2] all are reported in 2003.

2. BASIC NOTATIONS

In this section we bring out some basic terminology, used in the presentation of the algorithm. Let STR be a string of n characters in an array [0..n-1], and ∑(STR) be the alphabet of STR. To denote a substring in STR where i and j ranges from 0 to n-1,i<j, we denote it as STR[i..j]. For simplicity assume, STR is supposed to be terminated by a character called as sentinel and

DOI : 10.5121/ijist.2013.3402

represented by $, which is the unique lexicographically smallest character in STR. Let suffix(STR, i) be the suffix in STR starting at STR[i] and running to the end of the character array i.e. to the sentinel.

A suffix suffix(STR,i) is called as S-type or L-type, if suffix(STR, i) < suffix(STR,i+1) or suffix(STR, i) > suffix(STR,i+1), respectively. The last suffix suffix(STR,n-1) consisting of only the single character $ (the sentinel) which is predefined as S-type. We can classify a character STR[i] to be S-type or L-type. To store the type of every character/ suffix, we introduce an n-bit Boolean array b, where b[i] records the type of character STR[i] as well as suffix(STR, i): 1 for S-type and 0 for L-type. From the S-type and L-type descriptions, we observed the following properties:

Property 1:STR[i] is S-type, if STR[i] < STR[i+1] or
STR[i]=STR[i+1] and suffix(STR,i+1) is S-type.

Property 2:STR[i] is L-type, if STR[i] > STR[i+1] or
STR[i]=STR[i+1] and suffix(STR,i+1) is L-type.

By reading STR once from right to left, we can store the type of each character/suffix into type array ‘b’ in O(n) time.

As defined earlier, SAR(STR) (the notation of SAR is used for it when there is no confusion in the context), i.e., the suffix array of STR, stores the indices of all the suffixes of STR according to their lexicographical order. We observe that the pointers for all the suffixes beginning with a same character must span successively. Let us call a sub array in SAR for all the suffixes with the same first character as a bucket, where the head and the tail of a bucket refer to the first and the last items of the bucket. There must be no tie between any two suffixes sharing the identical character but of different types i.e., in the same bucket, all the suffixes of the same type are grouped together and the S-type suffixes are to the right of the L-type suffixes [12], [13]. Therefore, each bucket can be divided into two sub-buckets with respect to the types of suffixes inside i.e. the L and S-type buckets, where the S-type bucket is on the right of the L-type bucket.

3. Existing Algorithm: INDUCED SORTING VARIABLE LENGTH LMS SUBSTRINGS

A. Algorithm Framework

The framework of existing linear time suffix array sorting algorithm SAR-IS[15] that samples and sorts the variable-length LMS-substrings, is given in section III-C. Lines 1 to 4 give the reduced problem, which is then again recursively solved by the lines 5-8, and finally from the solution of the reduced problem, Line 9 induces the final solution for the original problem.

B. Basic Definitions

We start by introducing the terms of leftmost S-type (LMS) character, suffix, and substring as follows:

Definition 1:(LMS Character/Suffix) A character STR[i], i€[1,n-1] is called LMS, if STR[i] is S-type and STR[i-1] is L-type. A suffix suffix(STR,i) is called LMS, if STR[i] is an LMS character.

Definition 2: (LMS-Substring) An LMS-substring is (i) a substring STR[i..j] with both STR[i] and STR[j] being LMS characters and there exists no other LMS character in the substring, for $i \neq j$; or (ii) the sentinel itself. If we treat the LMS-substrings as elementary blocks of the string, we can effortlessly sort all the LMS substrings, then by using the order index of each LMS substring as its name, and replace all of the LMS-substrings in STR by their names. Therefore, the string STR can be represented by a shortened string, denoted by R_1, thus the problem size can be further minimized to fast up solving the problem in divide-and-conquer manner

Definition 3: (Order of Substring) To find out the order of any two LMS-substrings, first compare their corresponding characters from left to right. For each pair of characters, compare their lexicographical values first and then their types, if the two characters are of the same lexicographical value, where the S-type is taken as highest priority than the L-type. From this definition ,we see that two LMS-substrings can be of the same order index, i.e., the same name, if they have same, in terms of the lengths, and the characters, and the types. Assigning the S-type character a higher priority is based on a property directly from the definitions of L-type and S-type suffixes in [12]: suffix(STR, i)> suffix(STR, j), if (1) STR[i] > STR[j], or (2) STR[i]=STR[j], suffix(STR, i)and suffix(STR, j) are S-type and L-type, respectively. To sort all the LMS-substrings, no excess physical space is essential for storing them. We simply maintain a pointer array, denoted by P_1, which contains the pointers for all the LMS-substrings in STR and can be made by scanning STR or by reading the Boolean array b once from right to left in O(n) time.

Definition 4: (Pointer Array P_1) is an array which has the pointers for all the LMS substrings in STR with their original positional order being conserved. If we have all the LMS substrings sorted in the buckets in their lexicographical order, where all the LMS substrings in a bucket are identical, now we name each and every item of the pointer array P_1 by the index of its bucket to result in a revived string R_1. We say the two equal size substrings STR[i..j] and STR[i′..j′] are identical, if and only if STR[i + k]=STR[i′ +k] and b[i +k]=b[i′ +k], for k Є [0,j-i].

C. Algorithm

```
SAR-IS(STR,SAR)
        STR- is input string;
        SAR-output of suffix array of STR;
        b:array[0..n-1] of Boolean;
$P_1$,$R_1$:array[0....$n_1$] of integer; $n_1$=||$R_1$||
BKT:array[0..||∑(STR)||-1] of integer;
Step 1. Scan STR once to classify all the characters as L-Type or S-Type into b;
Step 2. Scan b once to find all LMS –substrings in STR into $P_1$;
Step 3. Induced sort all the LMS-substrings using $P_1$ and BKT;
Step 4. Name each LMS-substring in STR by its bucket index to get a new shortened string $R_1$;
Step 5. if each character in $R_1$ is unique then
Step 6. Directly compute $SAR_1$ from $R_1$;
Step 7. else
Step 8. SAR-IS($R_1$,$SAR_1$); //Recursive call
```

```
Step 9.  Induce SAR from R_1;
Step 10. Return
```

The above mentioned algorithm is the existing one.

4. Proposed Algorithm

In SA-IS, the additional working space is mainly composed of the bucket counter array ‘BKT’ and the type array ‘t’ at each recursion level. Our proposed algorithm differs from the existing one in two cases. They are

1. We use the MSB bit of the suffix array to store the type of the character(S-type or L-type) thereby avoiding the space needed for the type array ‘t’ suggested in the existing algorithm.

2. We reuse the unused space in SAR for the bucket array BKT.

We have observed that the input STR has been reduced to at least n/3 at the initial level (level-0) for the standard suffix array datasets .So, we can use of the unused space of SAR for the variable BKT in deeper levels rather than creating memory using malloc. As, in the existing algorithm -1 is used as initialization (default) value for suffix array SAR. In the proposed algorithm we use 0X7FFFFFFF as initialization value for suffix array SAR as the MSB bit is used to classify the S-type or L-type characters. Here we assume a 32-bit machine and the integer occupies 4-bytes.

The variable Buf_ptr is used which records the start address of the unused space of SAR at initial level(i.e level-0) so that we can reuse this space in the next levels (i.e. from 1st Level) for the bucket array (See Fig 1). We can also make use of this space for the L or S-type arrays if the space is still available.

As we can see the space of SAR_0 is reused for the level-1 because the size of the problem gets decreased as the level progresses.

4.1 Algorithm

```
SAR-IS (STR, SAR)
STR- is input string;
SAR-output of suffix array of STR;
P_1, R_1: array [0...n_1] of integer; n_1=||R_1||
BKT: array [0...||∑ (STR) ||-1] of integer; //uses unused space in subsequent iterations
Buf_ptr : pointer to unused space in SAR
Step 1. Scan STR once to classify all the characters as L-Type or S-Type into MSB bits of SAR;
Step 2. Scan MSB's of SAR once to find all LMS substrings in STR into P_1;
Step 3.  if level Not Equal to 0 then
            BKT=buf_ptr;//assign the start address of unused  buffer
Step 4. Induced sort all the LMS-substrings using P_1 and BKT;
```

```
Step 5. Name each LMS-substring in STR by its bucket index to get a new shortened string $R_1$;
Step 6. If level Equal to 0 then assign the start address of
          unused space of SAR to buf_ptr.
Step 7. if Each character in $R_1$ is unique  then
Step 8. Directly compute $SAR_1$ from $R_1$;
Step 9. else SAR-IS($R_1$,$SAR_1$); //Recursive call
Step 10.Once again scan STR to classify all the characters as L-Type  or S-Type into MSB bits
          of  SAR;
Step 11.Induce SAR from $SAR_1$;
Step 12.return
```

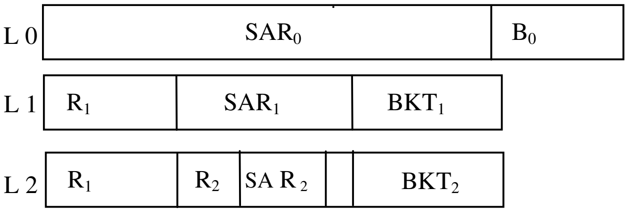

Fig 1.Example for the re usage of the buffer SAR

The re usage of the buffer is illustrated in Fig 1.The notation L 0, L 1, L 2 stands for Level-0, Level-1, Level-2.

4.2 Experimental Results

The algorithm was implemented in VC++ using the Microsoft Visual Studio under Windows XP platform. The Table II and Fig 2 give the overview of the space consumed by the existing and the proposed algorithms. The data sets in Table I used in our experiment are downloaded from Canterbury [14] and Manzini-Ferragina[16].

Dataset	‖∑‖,Characters
bible.txt	63,4047392
chr22.dna	4,34553758
e.coli	4,4638690
howto	197,39422105
world192.txt	94.2473400
sprot34.dat	66,109617186
etext99	146,105277340
rfc	120,116,421,901
rctail196	93,114,711,151
linux-2.4.5.tar	256,21,508,430
w3c2	256,104,201,579
alphabet	26,100000
random	26,100000

TABLE I Datasets used in the Experiment

Dataset	Space(in Mega Bytes)	
	Existing Algorithm	Proposed Algorithm
bible.txt	21.81	20.10
chr22.dna	179.10	165.85
e.coli	25.25	22.9
howto	204.47	189.11
world192.txt	13.61	12.58
sprot34.dat	556.57	524.48
etext99	544.14	503.74
rfc	590.53	556.99
rctail196	577.29	548.81
linux-2.4.5.tar	130.82	103.53
w3c2	521.11	498.60
alphabet	1.35	1.23
random	1.48	1.23

TABLE II Space Consumed by the Existing and Proposed Algorithm

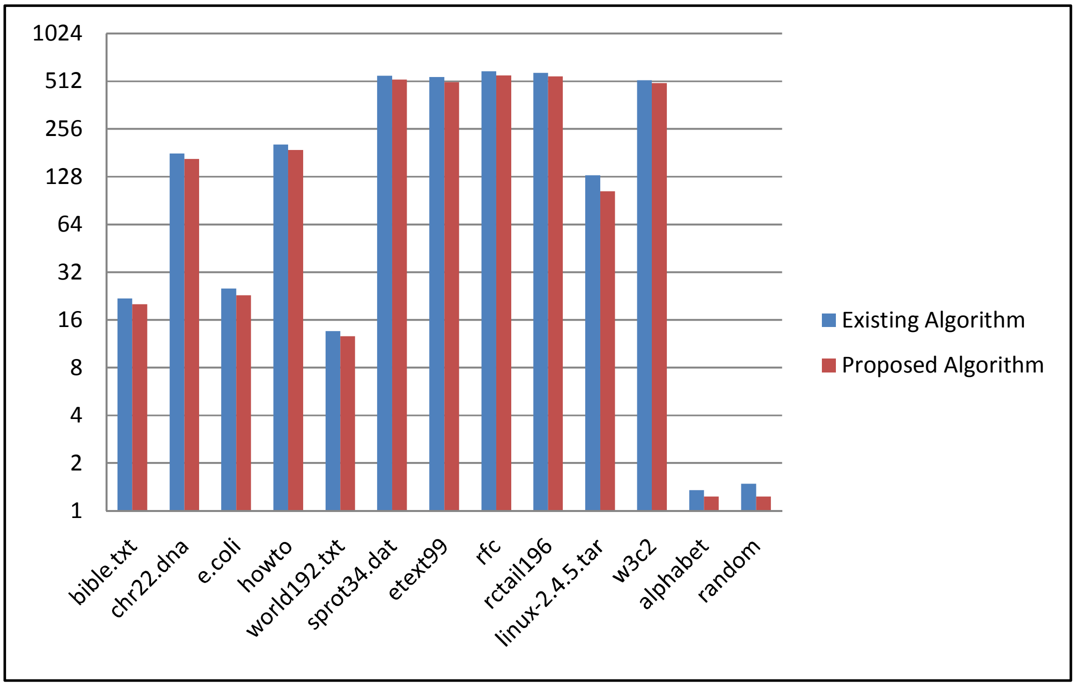

Fig 2. Logarithmic graph (base 2) showing the comparison between Existing and Proposed Algorithm

The datasets that are in Table I are downloaded from the benchmark repositories for SACAs, which includes Canterbury [14], Manzini-Ferragina[16].These datasets have constant alphabets with sizes less than or equal to 256 and one byte is taken for each character.

4.3 Conclusions

The proposed algorithm makes the algorithm space efficient by using the MSB bit of SAR to classify L-type and S-type characters and reuses the space of SAR for the bucket array at each level there by reducing nearly 25% of the space needed when compared to the existing algorithm. The results for the various data sets are shown in the Table II.

REFERENCES

[1] D.K. Kim, J.S. Sim, H. Park, and K. Park, "Linear-Time Construction of Suffix Arrays," Proc. Ann. Symp Combinatorial Pattern Matching (CPM '03), pp. 186-199. 2003.

[2] J. Karkkainen, P. Sanders, and S. Burkhardt, "Linear Work Suffix Array Construction," J. ACM, no. 6, pp. 918-936, Nov. 2006.

[3] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," SIAM J. Computing, vol. 22, no. 5, pp. 935-948, 1993.

[4] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," Proc. First Ann. ACM-SIAM Symp. Discrete Algorithms (SODA '90), pp. 319-327, 1990.

[5] S.J. Puglisi, W.F. Smyth, and A.H. Turpin, "A Taxonomy of Suffix Array Construction Algorithms," ACM Computing Surveys, vol. 39, no. 2, pp. 1-31, 2007.

[6] R. Grossi and J.S. Vitter, "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching," Proc. Symp. Theory of Computing (STOC '00), pp. 397-406, 2000.

[7] T.W. Lam, K. Sadakane, W.K. Sung, and S.M. Yiu, "A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays," Proc. Int'l Conf. Computing and Combinatorics, pp. 401-410, 2002.

[8] G. Manzini and P. Ferragina, "Engineering a Lightweight Suffix Array Construction Algorithm," Algorithmica, vol. 40, no. 1, pp. 33- 50, Sept. 2004.

[9] S. Kurtz, "Reducing the Space Requirement of Suffix Trees," Software Practice and Experience, vol. 29, pp. 1149-1171, 1999.

[10] W.K. Hon, K. Sadakane, and W.K. Sung, "Breaking a Time-and-Space Barrier for Constructing Full-Text Indices," Proc. 44th Ann. IEEE Symp. Foundations of Computer Science (FOCS '03), pp. 251-260, 2003.

[11] J. Karkkainen and P. Sanders, "Simple Linear Work Suffix Array Construction," Proc. 30th Int'l Conf. Automata, Languages, and Programming (ICALP '03), pp. 943-955, 2003.

[12] P. Ko and S. Aluru, "Space Efficient Linear Time Construction of Suffix Arrays," Proc. Ann. Symp. Combinatorial Pattern Matching(CPM '03), pp. 200-210. 2003.

[13] P. Ko and S. Aluru, "Space-Efficient Linear Time Construction of Suffix Arrays," J. Discrete Algorithms, vol. 3, nos. 2-4, pp. 143-156, 2005

[14] The Canterbury Corpus website. [Online]. Available: http://corpus.canterbury.ac.nz/.

[15] GeNong, Sen Zhang, Wai Hong Chan, "Two Efficient Algorithms for Linear Time Suffix Array Construction", *IEE Transactions on Computers,* vol. 60, pp.1471-1484,Oct.2011.

[16] Light weight corpus datasets [Online].Available: http://people.unipmn.it/manzini/lightweight/corpus