Memory Efficient Suffix Array Construction

Maninder Kaur
Haryana College of Technology and Management
Kaithal,Haryana
17122003.

***ABSTRACT*—Suffix array is an indexing data structure that stores all the suffixes (Suffixes means substrings of a string) of a string in sorted order (lexicographically). In numerous applications like pattern matching, data compression, string processing, and in the field of biology suffix array is the choice of the most. Over 20 years many researchers have put their efforts to make suffix array construction space efficient. In this paper we have proposed a new algorithm for constructing space efficient suffix array. Also we have made a memory comparison of our algorithm with MM Prefix-Doubling algorithm introduced by Manber and Mayer. Experimental results show our approach is better than MM Prefix Doubling algorithm in terms of memory.**

Keywords: suffix array, lexicographically, Prefix Doubling

I. INTRODUCTION

Suffix array is an indexing data structure that stores all the suffixes (Suffixes means substrings of a string) of a string in sorted order (lexicographically). Manber and Mayer in 1993[1] introduced suffix array as a space efficient alternative to suffix tree. As the size of textual database is increasing day by day so there is a need to store and maintain it efficiently. Large size of textual database requires more space for storing the data in them. There are several indexing data structures that full fill this goal. Suffix array is one of them. Suffix array store the indexes of the sorted suffixes of a string. Introducing a memory efficient construction of suffix array is a bottleneck. Over 20 years many researchers have put their efforts to make suffix array construction space efficient.

In this paper we have introduced a new algorithm named HeapSA for constructing space efficient suffix array. Also we have made a memory comparison of this algorithm with MM Prefix-Doubling algorithm [9] proposed by Manber and Mayer.

Also in Section 2 we present the overview of basic notations used. In Section 3 we present our new algorithm HeapSA. Then in Section 4 Experimental results are given and draw conclusions about their space efficiency in Section 5.

II. BASIC NOTATIONS

Let $|\Sigma|$ be a constant, indexed alphabet consisting of symbols α_i, i= 1, 2.....$|\Sigma|$ ordered α_1, α_2... $\alpha_{|\Sigma|}$. In this paper we will assume the common case that $|\Sigma|$ Є 0...255, where each symbol requires one byte of storage. Throughout we consider a finite, nonempty string s= s[0....n] = s [0] s [1]........s[n] of n+1symbols. The first n symbols of s are drawn from $|\Sigma|$ and comprise the actual input. The final character s[n] is a special "end of string" character,$, defined to be lexicographically smaller than any other character in $|\Sigma|$. For i = 0.......... n we write s[i..n] to denote the suffix of s of length n-i + 1, that is s[i....n] = s[i]s[i + 1] .... s[n]. For simplicity we will frequently refer to suffix s[i...n] simply as "suffix i". We are interested in computing the suffix array of s, which we write SAs or just SA. The suffix array is an array SA [0....n] which contains a permutation of the integers 0....n such thats [SA[0]....n] < s[SA[1]...n] < .... < s[SA[n]....n].

III. NEW ALGORITHM

We have proposed a new algorithm HeapSA. In HeapSA an array of type TreeMap is used to store <key,value> pair. To sort the elements of array Heap Sort is used. Heap can be a min heap or max heap but satisfy the heap property. In max heap the largest element is stored at the root, and the subtree rooted at a node contains values no larger than the root node. The description of the algorithm is given later in this section. HeapSA algorithm is as follow...

HeapSA

1. Create an array A of type TreeMap
2. string← file
3. create_suffixes(String s)
4. Intialize index=0;
5. Repeat steps 6 to 7 for all suffixes
   6. index← index + 1.s
   7 . Insert into array A (suffix, index)
8. heapsort_TREEMAP(array A)
9. for all elements of array A
   SA← A[i].index

create_suffixes (String s)

1. Intialize i=0
2. Repeat steps 3 to 4 while length≠ zero
   3. Create suffix by reading string from string[i] to string [length]
   4. i←i+1.

return suffixes

heapsort_TREEMAP(array A)

1. BUILD-HEAP(A)
2. for i → length[A] down to 2
3. do exchange A[1] ↔ A[i]
4. heap-size [A] ← heap-size[A] -1

IJERT

5. M_HEAPIFY(A, 1)

BUILD-HEAP(A)

1. heap-size [A] ← length [A]
2. for i ← length [A]/2 down to 1
3. do M_HEAPIFY (A, i)

M_HEAPIFY(A, i)

1 l ← LEFT(i)
2 r ← RIGHT(i)
3 if l ≤ heap-size [A] and A[l].suffix > A[i].suffix
4 then largest ← l
5 else largest ← i
6 if r ≤ heap-size[A] and A[r].suffix > A[largest].suffix
7 then largest ← r
8 if largest != i
9 then exchange A[i] ← A[largest]
10 M_HEAPIFY (A, largest)

First of all, we created an array of objects one object of which is capable of taking a <key value> pair as information part. The suffix will be the key part and index will be value part of node. Next, the file is read in a string and by leaving one character at a time from starting of string, the suffixes are created. Index values are incremented by one every time a new suffix is added to array. When all the suffixes are stored in array as (suffix, index) pair, heapsort_TREEMAP (array A) is used to sort the suffixes. In heap sort root is maximum among its left and right child. For sorting root node is replaced by the last node and array A is updated. Then check the heap property if it is violated then maintains the max heap property by using M_HEAPIFY. After that heap is built and again replace the root with last node and do the same. At last the index part of every object from array is retrieved as suffix array.
For example consider the string s= “Apple$”

1	2	3	4	5	6
A	p	P	L	E	$

Step 1: Its Suffixes are stored in array A along with indexes.

A<suffix,index>

Suffix	Index
Apple$	1
pple$	2
ple$	3
le$	4
e$	5
$	6

Step 2: Apply Heap sort on array A for sorting the suffixes

IJERT

- Build heap with 6nodes

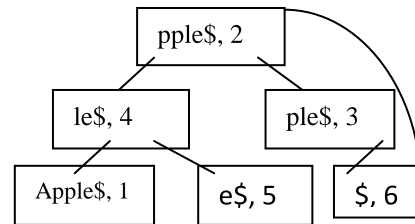


Fig.1. Heap of array A

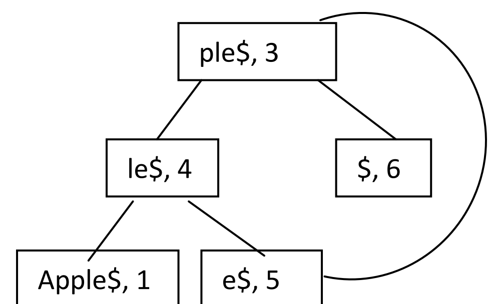
pple$	2
le$	4
ple$	3
Apple$	1
e$	5
$	6

Fig. 2. Updated array A

- Replace Root node with the last node and update array A. Heap size is reduced by 1. Again build heap if maxheap property is violated. ‘$’ is smallest among all the suffixes but when ‘pple$’ is exchanged with ‘$’, ‘$’become the root of heap violated the max heap property. So build the heap again with 5 nodes. Again do the same process.

$	6
le$	4
ple$	3
Apple$	1
e$	5
pple$	**2**

a)



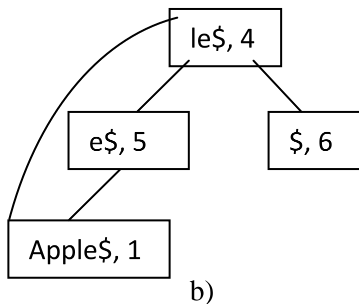
b)

Fig. 3.a) updated array A_1 and b) Heap

- Now exchange 'ple$' with 'e$'and update the array A. Now build heap for 4 nodes by maintaining max heap property.

e$	5
le$	4
$	6
Apple$	1
ple$	**3**
pple$	**2**

a)



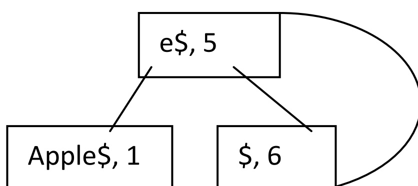
b)

Fig. 4. a)updated array A_2 and b)Heap

- Now exchange 'le$' with 'Apple$' and update the array A. Build heap with 3 nodes

Apple$	1
e$	5
$	6
le$	**4**
ple$	**3**
pple$	**2**

a)



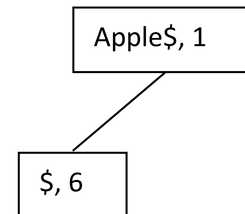
b)

Fig. 5. a) updated array A_3 and b) Heap

- Now exchange 'e$' with '$'and update the array A. Build heap with 2 nodes.

$	6
Apple$	1
e$	**5**
le$	**4**
ple$	**3**
pple$	**2**

a)



b)

Fig. 6. a) updated array A_4 and b)Heap

- Now exchange 'Apple$' with '$' and update array A. Finally suffixes are sorted as shown in fig.7.

Suffix	Index
$	6
Apple$	1
e$	5
le$	4
ple$	3
pple$	2

Fig.7: Array A

Step 3: Retrieve indexes of sorted array and store in suffix array SA.

SA={6,1,5,4,3,2}

In this approach memory used by HeapSA is reduced for constructing suffix array SA. Experimental results shows the memory used by HeapSA and MM prefix doubling algorithm in the next section.

IV. EXPERIMENTAL RESULTS

We have implemented our algorithm in Java. Also the implementation of MM prefix doubling algorithm is taken from http://algs4.cs.princeton.edu/63suffix/Manber.java.html.For

IJERT

testing the memory performance of both the algorithms we have used YourKit Java Profiler tool. Different text files are taken from http://textfiles.com/directory.html.Testing results are shown in table 1 and Graph 1.

File	File size(KB)	Memory used by algorithms(MB)	
		MM	*HeapSA*
Gnu	24	2.2	**1.6**
Email	22	2.1	**1.5**
Aaa	24	2.2	**1.7**
alphabet	11	1.2	**0.9**
protogen	30	2.6	**1.9**
Fbi	28	2.4	**1.8**

Table 1: Memory used by algorithm MM and HeapSA

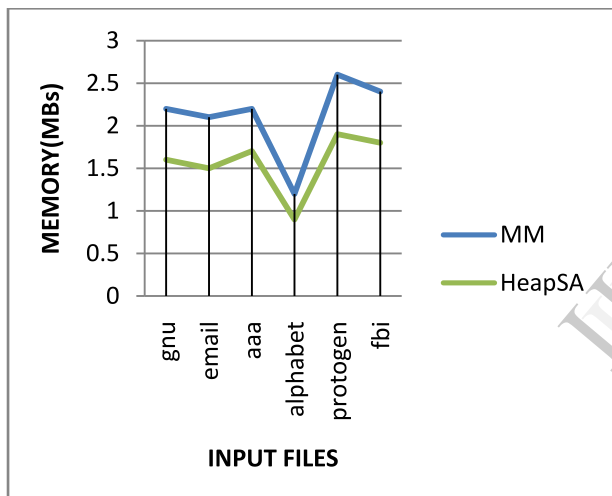


Fig.8. Graphical representation of Memroy Usage by algorithms.

V. CONCLUSION

The present work has focused on reducing the space in suffix array construction. we have proposed a new suffix array construction algorithms that is memory efficient than existing algorithm MM Prefix Doubling. By using 2d data structure and different sorting methods, we have achieved the goal of reducing memory in suffix array construction.

In this approach an array of type TreeMap stores the suffixes along with indexes and then Heap sort is applied for sorting the suffixes. At last indexes are retrieved and stored in suffix array. In this approach 30% of memory is reduced than MM Prefix Doubling.

From results it is clear that memory used by new algorithm is less but time taken by them is more than existing algorithm. In large scale applications as biological genome analysis, the space requirement is a severe drawback. As new suffix arrays require less space therefore these can be used to index and analyse very large biological genome which was not feasible before.

REFERENCES

[1]. U. Manber and G. Myers, "Suffix arrays: a new method for on-line search", *SIAM Journal on Computing*, vol. 22, pp. 935-48,1993.

[2] M. A. Maniscalco and S. J Puglisi, " Faster lightweight suffix array construction", *In J. Ryan and Dafik ,editors, Proceedings of 17th Australasian Workshop on Combinatorial Algorithms*, pp. 16–29, 2006.

[3] H. Itoh and H. Tanaka, "An efficient method for in memory construction of suffix arrays", *In Proceedings of the sixth Symposium on String Processing and Information Retrieval, Cancun, Mexico, IEEE Computer Society*, pp. 81-88.

[4] J. Dhaliwal, S. J. Puglisi and Andrew Turpin, "Trends in suffix sorting: a survey of low memory algorithms", *In Proceedings of the 35th Australasian Computer Science Conference (ACSC'12)*, 2012.

[5] J. Karkkainen and P. Sanders, "Simple linear work suffix array construction", *In Proceedings of the 30th International Colloq. Automata, Languages and Programming, Lecture Notes in Computer Science, Springer-Verlag, Berlin*, vol. 2971, pp. 943-955, 2003.

[6] D. K. Kim, J. S. Sim, H. Park, and K. Park, " Linear-time construction of suffix arrays*", In Baeza-Yates, E. Chavez, and M. Crochemore, editors, Proceedings of the 14th Annual Symposium CPM 2003, Lecture Notes in Computer Science, Springer-Verlag, Berlin*, vol. 2676, pp. 186-199, 2003.

[7] K. Schurmann and J.Stoye , "An incomplex algorithm for fast suffix array construction", *In Proceedings of The Seventh Workshop on Algorithm Engineering and experiments (ALENEX05) SIAM*, pp.77–85,2005.

[8] G. Manzini and P. Ferragina, "Engineering a lightweight suffix array construction algorithm", Algorithmica, vol. 40, pp. 33–50, 2004.

[9] S.Puglisi, W.Smyth and A.Turpin, "A taxonomy of suffix array construction algorithms", *ACM Computing Surveys*, vol. 39, no. 2, doi:10.1145/1242471.1242472.

[10] G. Nong, S. Zhang and W. Hong Chan, "Linear time suffix array construction using d-critical substrings", *In Proceedings of CPM, France*, Jun. 2009.

[11] R. Grossi , J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching", *In Proceeding of 32nd ACM Symposium on Theory of Computing,* pp. 397–406, 2000.

[12] J.L. Bentley, R. Sedgewick, "Fast algorithms for sorting and searching strings", *In Proceeding of 8th Annual ACM-SIAM Symposium On Discrete Algorithms*, pp. 360–369,1997.

[13] S. Kurtz, "Reducing the space requirement of suffix trees", *Software—Practice & Experience, vol. 29, no.13,* pp. 1149–1171, 1999.

[14] T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications", *In Proceeding of12th Annual Symposium on Combinatorial Pattern Matching CPM, Lecture Notes in Computer Science*, vol. 2089, pp. 181–192, 2001.

[15] N. J. Larsson, K. Sadakane, "Faster Suffix Sorting*", Technical Report LU–CS–TR, Lund University*, pp .99–214, 1999.

[16] G.Nong, S. Zhang, W. Hong Chan, "Two Efficient Algorithms for Linear Time Suffix Array Construction", *IEEE Transactions on Computers*, vol. 60, pp. 1471-1484,Oct. 2011.

[17] S. Rajasekaran , M. Nicolae, "An elegant algorithm for the construction of suffix arrays", *Journal of Discrete Algorithm,* DOI: 0.1016/j.jda.2014.03.001, 2014.

[18] S. Burkhardt, J. Karkkainen, "Fast lightweight suffix array construction and checking", *In R. Baeza-Yates, E. Chavez and M. Crochemore ,editors, Proceeding of* 14th *Annual Symposium on Combinatorial Pattern Matching , LNCS 2676, Springer-Verlag*, pp. 55–69, 2003.

IJERT