IJCSE International Journal of Computer Sciences and Engineering Open Access
Review Paper Volume-3, Issue-6 E-ISSN: 2347-2693

Role of Suffix Array in String Matching: A Comparative Analysis

Nagendra Singh

Maulana Azad National institute of technology, Bhopal, India
nagendra11manit@gmail.com

www.ijcseonline.org

Received: Jun/02/2015 Revised: Jun/08/2015 Accepted: Jun/17/2015 Published: Jun/30/2015

***Abstract*—** Text search is a classical problem in Computer Science, which reside in many data-intensive applications. For this problem, suffix arrays are the most widely known and used data structures, which enabling fast searches for phrases, terms, substrings and regular expressions in large texts. Potential application domains of this method includes large-scale search services, such as Web search engines, plagiarism checker where it is necessary to efficiently process intensive traffic streams of on-line queries. Suffix array is an effective way to construct the index of the full text i.e. sorted array of all suffix of string which is important for different kind of applications, perhaps most notably string matching, string discovery and block-sorting data compression. This paper elucidates intensive research toward efficient construction of suffix arrays with algorithms striving not only to be fast, but also "lightweight" (in the idea that they use small working memory).

***Keywords*—**Suffix sorting, Suffix array, fm index, trie structure.

I. INTRODUCTION

String search is a well known problem: given a text S[0...$n-1$] over some alphabet Σ of size σ =|Σ|, and a pattern P[0 . . .m − 1], locate the occurrences of P in S. Several different query modes are possible: asking whether or not P occurs (*existence* queries); asking how many times P occurs (*count* queries); asking for the byte locations in T at which P occurs (*locate* queries); and asking for a set of extracted contexts of S that includes each occurrence of P (*context* queries) [3].

When T and P are provided on a one-off basis, sequential pattern search methods take $O(n + m)$ time. When T is fixed, and many patterns are to be processed, it is liable to be more efficient to pre-process T and construct an *index*. Suffix tree is trie like indexed data structure which computed and stored in O(n) time and space [13], where n = |S|. Once constructed, it allows one to answer queries of the type "Is P a substring of S?" in O(m) time, where m = |P|. Moreover, all z occurrences of a pattern P can be found in $O(m+n)$ time, totally independent of the size of S. Still, in large scale applications such as genome analysis, the space requirements of the suffix tree are a severe minus. The suffix array is a more space economical index structure. The *suffix array* [1] is one such index, permitting *locate* queries to be answered in $O(m+\log n+k)$ time when there are k occurrences of P in S, using $O(n\log n)$bits of space in addition to S. Using it and an additional table, Manber and Myers (1993) showed that decision queries and enumeration queries can be answered in O(|P|+log |S|) and O(|P|+log |S|+z) time, respectively .

This paper is organized as follows. Our data structure is described in section 2. Various practical implementation that show the advantages of our method are given in next section which contains a description of the construction algorithm for our index and shows how pattern matching queries are implemented. We are currently working on further optimizations.

II. SUFFIX ARRAY

A suffix array is a data structure for efficiently answering queries on large strings. As a data structure, it is widely used in areas such as string matching, bio-informatics [6] and, in general, any region that deals with strings and string matching problems, so, as you can see, it is of great significance to know efficient algorithms to construct a suffix array for a given string. The suffix array of a string S is an array giving pointers to the suffixes of S sorted in lexicographical order of the suffixes. For concreteness, let suffix (S) denote the suffix of S starting at position i.

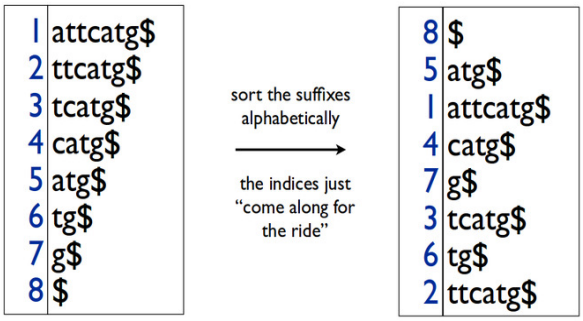

Figure I: Example

As an example, consider the string S is "attcatg" which results in the following suffix array:

Table I. Resultant Suffix Index

i	1	2	3	4	5	6	7	8
S[i]	8	5	1	4	7	3	6	2

in this example, the suffix array indicates that suff8 < suff5 < suff61< suff4 < suff7 < . . . , where < denotes the lexicographical ordering.

III. CONSTRUCTION ALGORITHM

A naive approach to construct a suffix array is to use a comparison-based sorting algorithm. These algorithms need *O(nlogn)* in suffix comparisons, but a suffix comparison runs in *O(n)* time, so the whole runtime of this approach is $n^2 \log n$. More advanced algorithms take advantage of the fact that the suffixes to be sorted are not arbitrary strings but related to each other. These algorithms strive to achieve the following goals:

- Minimal asymptotic complexity *O(n).*
- Lightweight in space, means small or no working memory alongside the text and the suffix array itself is needed.
- Fast in practice.

Now we consider better approach to construct the suffix array. The idea is to use the fact that strings that are to be sorted are suffixes of a single string. first sort all suffixes according to first character, then sort according to first 2 characters, then first 4 characters and so on even though the number of characters to be considered is less than 2n. The main point is, if we have sorted suffixes according to first 2^i characters, then we can sort suffixes according to first 2^{i+1} characters in *O(nlogn)* time using sorting algorithm like Merge Sort which takes nlogn. This is possible as two suffixes can be compared in *O(1)* time. The sort function is called *O(logn)* times (Note that we increase number of characters to be considered in powers of 2). Therefore overall time complexity becomes *O(nlognlogn). Here* we can use here radix sort to reduce time complexity to *O(nlogn)*. Third approach to build suffix array uses indirect method to form. Ukkonen's Suffix Tree [13] Construction takes *O(n)* time and space to build suffix tree for a string of length N and traversal of tree take *O(n)* to form suffix array. So overall, it's linear in time and space. Can you see why traversal is *O(n)* ? Since a suffix tree of string of length N will have at most N-1 internal nodes and N leaves. Traversal of these nodes can be done in *O(n).*

Algorithm I. Psuedo code for construct Suffix array

```
for( int  i = 0; i < s. size();i++){
m[s. substr(i,s.size()-i)] = i;
v.push back (s.substr (i,s.size()-i));}
sort(v.begin(),v.end());
for(int i = 0; i < v.size();i++{
cout << m[v[i]] << endl }
return 0;}
```

IV. RELATED WORK

3.1 Q-gram Based Database searching Using Suffix Array

With the increasing amount of DNA sequence data dumped in databases, examining for similarity to a query sequence has become an elementary operation in molecular biology. But even today's fast algorithms reach their limits when applied to all-versus-all comparisons of large databases. Here we discuss a new Database searching algorithm called QUASAR [5] (Q-gram Alignment based on Suffix Arrays) which was designed to quickly detect sequences with strong similarity to the query in a context where many searches are conducted on one database. Our algorithm a plies a modification of q-tuple filtering implemented on top of a suffix array.

Disadvantage is its requires large amounts of internal memory and takes rather more absolute time. Furthermore, for large data sets internal memory is likely to be insufficient for the construction and secondary memory versions are required to keep its running time within reasonable limits. This is an area of future research.

3.2 Search result clustering based Suffix array and VSM

With the rapid growth of big data such as web pages, search engines will usually present a long ranked list of documents. The users must sift through the list with "title" and "snippet" to find the required document. This manner may be good for some simple and specific tasks but less effective and efficient for ambiguous queries such as "apple" or "jaguar". To improve the effect and efficiency of information retrieval, an another method is to automatically organize string retrieval results into clusters. This paper proposed an improved Lingo algorithm called Suffix Array Similarity Clustering (SASC) for clustering search results. This method builds the clusters using improved suffix array, which disregards the redundant suffixes, and computing document similarity based on the title and short document snippets returned by Web search engines. Here we show that the SASC algorithm has not only a better performance in time-consuming than Lingo but also in cluster description

quality and precision than Suffix Tree Clustering.

The algorithm is based on four principles mentioned by the section 1. Firstly, the key terms discovered by suffix array can be taken as candidates for cluster labels. Then we can measure similarity between term and document vectors. So all the cluster labels can be separated from key terms (the candidates for cluster labels) and similarity documents are organized into groups according to the hierarchical relationship.

The whole process of clustering algorithm is composed of the four steps:

- Document parsing.
- Key terms extraction.
- Organizing into clusters and extracting cluster labels.
- Analyzing the hierarchy relation of clusters.

The efficiency of suffix array is improved by ignoring the redundant suffixes. This method takes far less time than Lingo. Furthermore, SASC supports hierarchical structure. In the future, we intend to further improve the time efficiency and the accuracy as well as consider other information such as the user's interaction with the clustering results for adaptive clustering.

3.3 Cloud based parallel suffix array construction based on MPI

Next Generation Sequencing machines are producing massive amount of genomics data currently. for indexing genomics data The suffix array is presently the best choice because of its efficiency and large number of applications. In this paper, they address the problem of constructing the suffix array on cluster in the cloud. We shows a solution that automates the formation of a computer cluster in a cloud and automatically constructs the suffix array in a distributed fashion over the cluster nodes. This has the advantage of summarizing all set-up details and Execution of the algorithm. Due to distributed nature of the techniques we use overcomes the problem that arises when the user wishes, because of cost or low memory machines in the cloud.

Our experiments show that our implementation scales well with the increasing number of processors. The cloud cost is affordable and it provides cost effective solution. The creation of the cluster involves creation of a master node and worker nodes from a certain machine image which includes pre-installed middleware and programs. After creating the nodes, the storage (virtual hard-disks) are created and associated to them. In this step, the MPICH2 is installed with all necessary configuration steps to enable parallel programming. Once the cluster is created, the user starts the suffix array construction by invoking the respective programs.

The cloud SACA system developed which is a solution that automates the establishment of a computer cluster in the cloud and automatically constructs the suffix array in a distributed fashion on the computer cluster.

IV. STRING MATCHING USING SUFFIX ARRAY

String search is a well known problem: given a text S[0...$n-1$] over some alphabet of size σ and a pattern P[0 . . . $m-1$], locate the occurrences of P in S. Given a text string 'S', the problem of string matching deals with finding whether a pattern 'P' of size m occurs in text 'S' of size n. If 'P' does occur then returning position in 'S' where 'P' occurs. Suffix array [1,6,7,8] based pattern matching based in preprocessing text rather than preprocess pattern (in classical methods).

Remember that we wanted to find occurrences of a pattern P in S. It is not hard to see that the pattern P occurs at position i of S if and only if P is a prefix of the suffix (S). This means

The occurrences of P in S = all suffixes of S having P as a prefix.

This observation allows us to transform the String matching problem into a prefix matching problem on all the suffixes.
But we haven't actually made any progress. To proceed, we will make use of the sorted property of suffix arrays. Since suffix arrays are sorted, we could perform a binary search to locate an occurrence of the pattern.
The number of comparisons required by a binary search is logarithmic in the size of the domain (i.e., the length of the suffix array, which has |S| entries). Each comparison is made between P and a suffix of S, which we know can be done in O (|P|) work. Therefore, for a total of O (|P| log |S|) work, we can locate an occurrence of P, or report that none exist.

Observation: For any prefix A, all suffixes of S that have prefix A are contiguous entries in the suffix array.

Pseudo code for indexed text searching using suffix array:

```
def search(P):
l = 0; r = n
while l < r:
mid = (l+r) /2
if P > suffixAt(A[mid]):
l = mid +1
```



```
else:
r = mid
s = l; r = n
while l < r:
mid = (l+r) /2
if P < suffixAt(A[mid]):
r = mid
else:
l = mid +1
return (s, r)
```

Algorithm II. Querying with Suffix array(binary search).

Finding the substring pattern P of length m in the string S of length n takes *O(mlogn)* time, given that a single suffix comparison needs to compare m characters. Manber & Myers (1990)[1] describe how this bound can be improved to *O(m+logn)* time using LCP information. The idea is that a pattern comparison does not need to re-compare certain characters, when it is already known that these are part of the longest common prefix of the pattern and the current search interval. Abouelhoda, Kurtz & Ohlebusch (2004)[2] improve the bound even further and achieve a search time of *O(m)*.
We saw three ways to query (Binary search) the suffix array:

- Typical binary search. Ignore LCP, *O(mlogn)*.
- Binary search using some skipping and LCP array, near to *O(m+logn)*.
- Binary search with skipping using all LCPs among suffixes.

VI. COMPARISON AND ANALYSIS

This work categorizes the algorithms into various categories to emphasize the data structure that drives the matching. These categories are online queries processing based and offline queries processing based.

Online algorithms are those where pattern can be preprocess before searching phase but cannot allow to preprocess text. Online technique do searching pattern without index. A string matching is called offline if we allowed to preprocess the text and make an index data structure. Although very fast online techniques are exist but their performance on large data set is unacceptable. Some offline methods are Trie structure, suffix tree, suffix array. Based on all the data represented in the paper, a comparative analysis of all the searching algorithms is presented in Table II.

VI. CONCLUSION

Performing query processing using suffix arrays. This research reviews and profiles some typical string search suffix array algorithms to observe their performance under various conditions and gives an insight into choosing the efficient algorithms. Classical method processes online queries which is used methods like Brute force, KMP, Boyer-Moore. We have studied how to process large streams of offline queries by indexing approaches like suffix array. We draw the final conclusion that the suffix arrays are a very useful data structure, extremely easy to implement. Thus it's not strange that during the last years many problems that were using it appeared in programming contests.

Table II: Comparative Analysis Between Different Algorithms

ALGORITHMS	TIME COMPLEXITY	SPACE COMPLEXITY	ONLINE/OFFINE PROCESSING	KEY IDEA	APPROACH
Brute Force	O(n*n)	O(1)	Online	Searching one by one	Linear searching
Robin Karp	O(mn)	O(m)	Online	Compare with hash function both text and pattern	Hashing based
KMP	O(m+n)	O(m)	Online	Construct an automation from pattern	Heuristic based
Boyer Moore	O(mn)	O(kn)	Online	Bad character and suffix rule	Heuristic based
Suffix Array	O(m)	O(nlogn)	Offline	Construct suffix index	Indexing based

REFERENCES

[1] Member, Ubi;Myers, Geene. "Suffix arrays:a new method for online string search" First annual ACM-SIAM Journel on Computing 22, (1993).

[2] Abouelhoda, Mohamed Ibrahim; Kurtz, Stefan; Ohlebusch, Enno "Replacing suffix trees with enhanced suffix arrays". Journal of Discrete Algorithms 2: 53, (2004).

[3] Gog, Simon, Alistair Moffat, J. Culpepper, Andrew Turpin, and Anthony Wirth. "Large-scale pattern search using reduced-space on-disk suffix arrays." IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 26, NO. 8, AUGUST 2014.

[4] Ahmed Abdelhadi, A. H. Kandil1 and Mohamed Abouelhoda. "Cloud-based Parallel Suffix Array Construction based on MPI " Middle East Conference on Biomedical Engineering (MECBME) ,(2014).

[5] Stefan Burkhardt, Andreas Crauser, Eric Rivals, Hans, martin. "q-gram based database searching using suffix array (quasar)" ,2006.

[6] Diego Arroyuelo, Carolina Bonacic, Veronica Gil-Costa, Mauricio Marin Gonzalo Navarro. "Distributed text search using suffix arrays" Elsevier Journal ,28 june 2014.

[7] Maan Haj Rachid, Qutaibah Malluhi, and Mohamed Abouelhoda. "A space-efficient solution to find the maximum overlap using a compressed suffix array ." Middle East Conference on Biomedical Engineering (MECBME) , 2014.

[8] Shunlai Bai, Wenhao Zhu, Bofeng Zhang , Jianhua Ma. "Search Results Clustering Based on Suffix Array and VSM ." IEEE/ACM International Conference on Green Computing and Communications & 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing , 2010.

[9] Juha Ka¨rkka¨inen , Peter Sanders , Stefan Burkhardt. "Linear Work Suffix Array Construction ." ACM Journal, Volume 53 Issue 6, (2006).

[10] Mohammadreza Ghodsi . "Approximate String Matching using Backtracking over Suffix Arrays ." Computer Science Department of University of Maryland at College Park , 2009.

[11] Nataliya Timoshevskaya, Wu-chun. "SAIS-OPT: On the Characterization and Optimization of the SA-IS Algorithm for Suffix array construction" IEEE Transaction, 2014.

[12] Hongwei Huo, Longgang Chen, Jeffrey Scott Vitter and Yakov Nekrich. "A Practical Implementation of Compressed Suffix Arrays with Applications to Self-Indexing." IEEE Journel DOI 1109.2014.49, 2014.

[13] Ricardo Baeza-Yates . "Modern information retrieval." ACM press, 1999.

[14] Esko Ukkonen , "On–line construction of suffix trees."Algorithmatica , Volume 14, Issue 3, 1995, pp 249-260.

AUTHORS PROFILE

Nagendra singh received B.tech degree in computer science and engineering From Uttar Pradesh technical university, Lucknow, India. He is now a research scholar at Maulana Azad national institute of technology, bhoapal. His current research interest includes space efficient data structure, data compression and information retrieval.