Performance Comparison & Analysis of DivSufSort-based & SAIS-based FM-Index

Sumayyea Salahuddin, Muniba Ashfaq, Madiha Sher, Laiq Hasan, Nasir Ahmad
Department of Computer Systems Engineering
University of Engineering & Technology, Peshawar Pakistan
sumayyea@nwfpuet.edu.pk, muniba@nwfpuet.edu.pk, madiha@nwfpuet.edu.pk, laiqhasan@uetpeshawar.edu.pk, n.ahmad@uetpeshawar.edu.pk

***Abstract*— FM-Index and Suffix Array are closely related to each other and are both extremely popular indexes for genomic sequences. They are used in several popular read alignment tools, including Bowtie, Bowtie 2, BWA, and GEM. In literature, there exist several Suffix Array Construction Algorithms (SACA). We have considered two popular SACA techniques: DivSufSort and SAIS. Both techniques construct suffix array in linear time. We have constructed FM-Index using both of these SACA algorithms. In this paper, we comprehensively describe our FM-Index construction approach and compare performance of these two indexes in terms of time for different string types in the Dataset. Our result shows that DivSufSort-based FM-Index performs 3.67% time efficient than SAIS-based FM-Index on 10 out of 11 strings in the Dataset.**

***Index Terms*—String Matching, Suffix Array, Suffix Sorting, Burrows-Wheeler Transform, Wavelet Tree, FM-Index**

I. Introduction

One of the most simple and handy data representations is strings, and for their efficient processing many algorithm exists in computer science. Suffix array is one such data structure used for string processing [1, 2]. It is a lexicographically sorted array of suffixes of a string. It is applied for pattern matching (counting or finding all the occurrences of a specific pattern) [1, 2], pattern discovery and mining (counting or finding generic, previously unknown, repeated patterns in data), information retrieval [3], genomic analysis (given genome's suffix array, perform binary search to find suffix intervals that have P pattern as a prefix or align sequences or to find similarities) [5, 6, 9], and data compression [8, 9]. In all these applications, suffix array construction – a process also known as suffix sorting – is one of the main computational bottlenecks.

For the last 15 years or so, research is focused on Suffix Array Construction Algorithms (SACAs) [11, 12, 13]. Most recent algorithms tend to use little memory as possible, or find a clever way to trade runtime, or by using compressed data structures [14], or by using disk, or some combination of these techniques. Another possible solution is to take advantage of multicore processors, GPUs, and clusters. Overall analysis time can be reduced when suffix array is constructed in parallel. Many sequential suffix array construction algorithms are developed but only few parallel algorithms can be found. Thus, the problem of practical and efficient techniques remains open.

Suffix Array is used to compute Burrows-Wheeler Transform (BWT) of a string, where BWT is a reversible transformation of a string that allows the string to be easily and efficiently compressed [15]. The BWT was discovered independently of the suffix array, but it is now known that the two data structures are equivalent. For human genome, suffix array needs greater than 12 GB space. BWT reduces it by keeping the size of the index same as the string size, thereby needing only 3 GB space. Fig. 1 shows BWT string and Suffix Array for X = googol$ [16].

FM-index [8, 4] is a BWT-based compressed index proposed by Ferragina & Manzini. It is used in many state-of-the-art software tools for mapping DNA short reads onto a known reference genome, for example, BWA [16], Bowtie2 [17], and SOAP3-DP. These aligners keep reference genome's BWT in main memory and perform highly optimized operations on BWT indices to allow very fast mapping of individual short reads. In literature, numerous methods exist for efficient implementation of FM-Index. These methods are briefly reviewed in the next section. In this paper, we have implemented FM-Index using RRR wavelet trees and fast suffix sorting algorithms using DivSufSort [13] and SAIS [12].

Paper consists of following sections: Section 2 gives overview of Suffix Array (SA) and FM-Index methods. Section 3 describes brief overview of DivSufSort and SAIS. Section 4 describes the proposed approach used for FM-Index construction. Section 5 presents results and analysis of our experiments. Section concludes with discussion and future work.

II. Overview of SA and FM-Index Methods

In literature, numerous space and time efficient methods are proposed for suffix sorting and FM-Index construction. Succinct full-text self-index takes less space and allows efficient searching for the occurrence of the pattern in text. Many such self-indexes have been designed, few of them are covered in this section. In [21], run-length FM-Index is presented, which takes less space than the text. In [22], Djamal introduces the use of relative FM-index that leads to significant

space savings in practice. In [19], Grabowski presents an alphabet independent and compressed FM-Index representation. It first compresses the text using Huffman code and then applies the Burrow-Wheeler transform on the compressed sequence. In [20], text is compressed using Kautz-Zeckendorf coding and then applied the Burrow-Wheeler transform that gives better results than existing succinct data structures.

Fast and efficient suffix sorting of big data is very useful in many applications. In [23], Larsson discusses the connection between suffix trees and context trees and how context trees can be used to represent BWT in a compressed and computationally efficient manner. In [24], Sadakane proposes another time-efficient computation of BWT based on a hybrid comparison-based sorting algorithm. Later, suffix tree based algorithms are improved [25, 26]. Heng Li [28] present a new method that is fastest for indexing short reads and long reads as well.

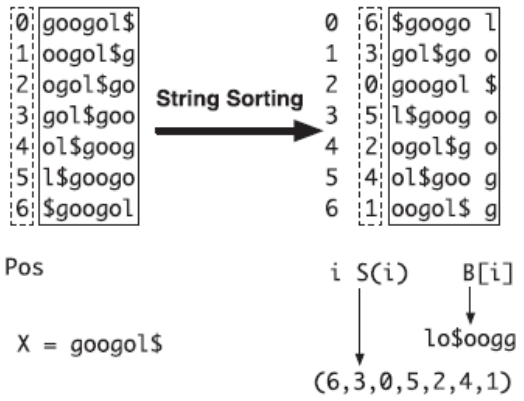

Fig. 1. BWT and Suffix Array for X = googol$. [16]

III. Overview of Suffix Array Algorithms

Let $S = s_0 s_1 ... s_{N-1}$ be a string of length N. Let $S[i, j] = s_i s_{i+1} ... s_j$ be substrings of S. Each s_i is a member of finite alphabet $\sum$ and size of alphabet is $|\sum|$. String is ended by special character "$" known as sentinel. It is not the member of alphabet. Let $Suf_i = S[i, N] = s_i, s_{i+1}, ... s_{N-1}, \$$ denotes the suffix starting at i and running to $. The suffix array SA built on S is an array of length N storing sequence of indexes $p_0, p_1, ..., p_{N-1}$ such that $Suf_{p_0} < Suf_{p_1} < ... < Suf_{p_{N-1}}$, where lexicographical order is represented via "<" sign [29]. For a sample string "googol$", Fig. 1 shows its suffix array as well as Burrows-Wheeler transform string [16].

For time-efficient and space-efficient construction of suffix array, several suffix array construction algorithms (SACA) such as Ko-Aluru algorithm (KA) [27], DivSufSort [13], Induced Sorting algorithm (SAIS) [12], Bucket-Pointer Refinement algorithm (BPR), and MSufSort have been proposed. In this work, we consider two linear-time algorithms a) SAIS and b) DivSufSort for suffix array construction. In this section, we briefly discuss each of these techniques.

```
SA-IS(S, SA)
    ▷ S is the input string;
    ▷ SA is the output suffix array of S;
    t: array [0..n − 1] of boolean;
    P1, S1: array [0..n1 − 1] of integer; ▷ n1 = ||S1||
    B: array [0..||Σ(S)|| − 1] of integer;
 1  Scan S once to classify all the characters as
    L- or S-type into t;
 2  Scan t once to find all the LMS-substrings in S into P1;
 3  Induced sort all the LMS-substrings using P1 and B;
 4  Name each LMS-substring in S by its bucket
    index to get a new shortened string S1;
 5  if Each character in S1 is unique
 6     then
 7          Directly compute SA1 from S1;
 8     else
 9          SA-IS(S1, SA1); ▷ Fire a recursive call
10  Induce SA from SA1;
11  return
```

Fig. 2. SAIS Algorithm. [12]

A. SAIS

SAIS stands for Suffix Array Induced Sorting [12]. It constructs suffix array in linear time and is considered as benchmark in suffix sorting algorithms. It is implemented in the state-of-the-art BWA [16] sequence aligner.

Fig. 2 shows SAIS algorithm. According to Mori [12], it is a recursive divide-and-conquer procedure that consists of two linear components: problem reduction and solution induction. In problem reduction: string is scanned first to classify each character s_i as S- or L- type. A character s_i is S-type if $s_i < s_{i+1}$ or $s_i = s_{i+1}$ *and* Suf_{i+1} is S-type; and L-type if $s_i > s_{i+1}$ or $s_i = s_{i+1}$ *and* suf_{i+1} is L-type. These types are stored in n-bit Boolean array t where S-type is represented by 1 and L-type is represented by 0. A character s_i is called LMS if s_i is S-type and s_{i-1} is L-type. A suffix Suf_i is called LMS if s_i is an LMS character. An LMS-substring is a substring $S[i, j]$ with both s_i and s_j LMS characters and there is no other LMS character in substring for $i \neq j$ or sentinel. Array t is scanned to find all LMS-substrings that locates first occurrence of s-type and stores in P1 array. Then all LMS-substrings are induced sorted using array P1 and bucket B. Each LMS-substring is named by its bucket index to get a new shortened string. In solution induction: problem is solved by traversing recursively once to induce sort all L-type suffixes from the sorted LMS suffixes and traversed another time to induce sort all the type-S suffixes from the sorted L-suffixes.

Fig. 3 shows the running example of SAIS algorithm for string "mmiissiissiippii$" [12]. Line 3 shows type array t entries and at line 4 all the LMS-suffixes in S are marked by *. These are 2, 6, 10, and 16. Line 6 shows the five buckets for all the suffixes marked by their first character i.e. $, i, m, p, and s.

Line 7 shows suffix array SA with each bucket delimited by a pair of braces and content initialized by -1. Next indices of all LMS-Suffixes i.e. 2, 6, 10, and 16 are put into their respective buckets from end to head. Then all the L-type LMS-prefixes are induced sorted as follows. Line 11 shows the current head of each bucket marked by "^" symbol. SA is scanned from left to right to visit the index marked by "@" symbol. When visiting the "@" symbol index, previous index type is checked to know if it is L-type or not. If L-type than that index is appended in SA as shown in lines 12-28 and bucket head is respectively forwarded one step to the right. Next, all the LMS-prefixes are induced sorted from stored L-type prefixes as follows. Each bucket's end is marked and then SA is scanned from right to left. When visiting "@" symbol, previous index type is checked to know if it is S-type or not. If S-type than that is appended in SA as shown in lines 32-44 and bucket head is forwarded to the left. At the end, all LMS-prefixed are sorted in their order shown in line 44.

```
00         0                   1
01 Index: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
02      S: m m i i s s i i s s i i p p i i $
03      t: L L S S L L S S L L S S L L L L S
04    LMS:     *       *       *           *

05 Step 1:
06 Bucket:   $               i                   m       p           s
07      SA: {16} {-1 -1 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}

08 Step 2:
09 Bucket:   $               i                   m       p           s
10      SA: {16} {-1 -1 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}
11           @^  ^                           ^       ^
12          {16} {15 -1 -1 -1 -1 10 06 02} {-1 -1} {-1 -1} {-1 -1 -1 -1}
13            ^   @   ^                           ^       ^   ^
14          {16} {15 14 -1 -1 -1 10 06 02} {-1 -1} {13 -1} {-1 -1 -1 -1}
15            ^       @   ^                       ^       ^   ^
16          {16} {15 14 -1 -1 -1 10 06 02} {-1 -1} {13 -1} {09 -1 -1 -1}
17            ^       ^       @                   ^       ^   ^
18          {16} {15 14 -1 -1 -1 10 06 02} {-1 -1} {13 -1} {09 05 -1 -1}
19            ^       ^           @               ^       ^   ^
20          {16} {15 14 -1 -1 -1 10 06 02} {01 -1} {13 -1} {09 05 -1 -1}
21            ^       ^               @           ^       ^   ^
22          {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 -1} {09 05 -1 -1}
23            ^       ^                   @       ^       ^   ^
24          {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 -1 -1}
25            ^       ^                       ^ @     ^       ^
26          {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 08 -1}
27            ^       ^                       ^       ^ @     ^
28          {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 08 04}
29            ^       ^                       ^       ^   @

30 Step 3:
31 Bucket:   $               i                   m       p           s
32      SA: {16} {15 14 -1 -1 -1 10 06 02} {01 00} {13 12} {09 05 08 04}
33            ^       ^                   ^       ^       ^           @^
34          {16} {15 14 -1 -1 -1 10 06 03} {01 00} {13 12} {09 05 08 04}
35            ^       ^                   ^       ^       ^           @   ^
36          {16} {15 14 -1 -1 -1 10 07 03} {01 00} {13 12} {09 05 08 04}
37            ^       ^                   ^       ^       @^          ^
38          {16} {15 14 -1 -1 -1 11 07 03} {01 00} {13 12} {09 05 08 04}
39            ^       ^               @   ^       ^       ^           ^
40          {16} {15 14 -1 -1 02 11 07 03} {01 00} {13 12} {09 05 08 04}
41            ^       ^           @       ^       ^       ^           ^
42          {16} {15 14 -1 06 02 11 07 03} {01 00} {13 12} {09 05 08 04}
43            ^       ^                   ^       ^       ^           ^
44          {16} {15 14 10 06 02 11 07 03} {01 00} {13 12} {09 05 08 04}
45            ^       ^           @       ^       ^       ^           ^

46     S1: 2 2 1 0
```

Fig. 3. SAIS Algorithm Example. [12].

B. DivSufSort

DivSufSort [13] also constructs suffix array in linear time. It is an open source library developed by Yuta Mori. It is used for the forward BWT in ncomp (the engine for WinRK) and in early version of bcm. It is an improvement of Itoh-Tanaka's two-stage (ITTS) algorithm [29], which is very fast and efficient suffix array method for both small and large texts. It comprises of four steps: In first step, type B* suffixes are selected. It is done by dividing suffixes into two types A and B such that a character s_i is type A if $Suf_i >_1 Suf_{i+1}$ and type B if $Suf_i \leq_1 Suf_{i+1}$ where symbols $\leq_1, >_1$ denotes the lexicographic order between two strings. Next suffixes of type B whose subsequent suffix is a type A are selected. These suffixes are called type B*. In second step, type B* suffixes are sorted using substring sorting technique given in KA [27]. To reduce complexity in this step, it detects and induces tandem repeats using MSufSort. In step 3, sorted suffixes are scanned from right to left. For each suffix $SA[i]$, if previous suffix array index i.e. $SA[i]-1$ is a type B suffix than current suffix is moved to the last empty position of its bucket. In final step, suffix array is completed by scanning sorted suffixes from left to right like Itoh-Tanaka algorithm or KA algorithm [27].

IV. FM-Index Methodology

Fig. 4 shows the block diagram of our FM-Index construction method. For a given string, first suffix array (SA) is constructed using DivSufSort and SAIS techniques. Next using SA, Burrows-Wheeler Transform (BWT) is computed. The combination of SA with BWT forms FM-Index, which enables backward search and self- indexing. Next, wavelet tree is used to encode BWT into balanced binary-tree of bit vectors. Finally, Wavelet tree nodes are stored as RRR sequences for fast binary rank queries and compression. Each of these blocks is discussed briefly as follows.

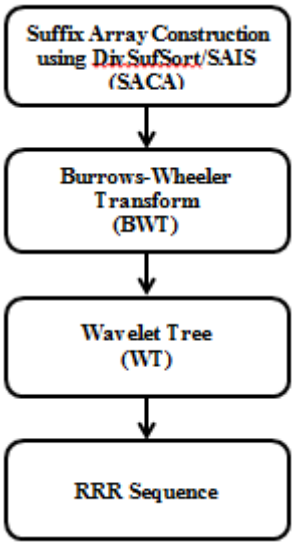

Fig. 4. Block Diagram showing FM-Index Construction & Efficient Rank Query Implementation

A. Suffix Array (SA)

For string S, Suffix Array can be constructed as:

1) Make an array of pointers to all suffixes Suf_i

2) Use lexicographical order of suffixes to sort their respective pointers.

Fig. 5 shows an example string $T = ATGACGGATCA\$$ and its suffix array. We construct suffix array using two linear time algorithms: DivSufSort and SAIS in this paper. Each of these techniques is discussed in section 3.

SA			SA	
1	*ATGACGGATCA$*		12	$
2	*TGACGGATCA$*		11	*A$*
3	*GACGGATCA$*		4	*ACGGATCA$*
4	*ACGGATCA$*		8	*ATCA$*
5	*CGGATCA$*		1	*ATGACGGATCA$*
6	*GGATCA$*	sort →	10	*CA$*
7	*GATCA$*		5	*CGGATCA$*
8	*ATCA$*		3	*GACGGATCA$*
9	*TCA$*		7	*GATCA$*
10	*CA$*		6	*GGATCA$*
11	*A$*		9	*TCA$*
12	$		2	*TGACGGATCA$*

Fig. 5. Suffix Array for $T = ATGACGGATCA\$$

BWT	SA	
A	12	$
C	11	*A$*
G	4	*ACGGATCA$*
G	8	*ATCA$*
$	1	*ATGACGGATCA$*
T	10	*CA$*
A	5	*CGGATCA$*
T	3	*GACGGATCA$*
G	7	*GATCA$*
C	6	*GGATCA$*
A	9	*TCA$*
A	2	*TGACGGATCA$*

Fig. 6. SA and BWT for $T = ATGACGGATCA\$$

B. Burrows-Wheeler Transform (BWT)

Given a string of length N, Burrows-Wheeler Transform (BWT) is computed from Suffix Array and the original string T as shown in Eq. 1.

$$B[i] = \begin{cases} T\left[S(i)-1\right], & S(i) \neq 0 \\ \$, & S(i) = 0 \end{cases} \quad (1)$$

Fig. 6 shows BWT for string $T = ATGACGGATCA\$$.

C. Wavelet Tree (WT)

Rank query on the string S is defined as $rank(i, sym) = m$, where *m* represents occurrence of symbol *sym* in range $S[1, i]$ e.g. $rank(5, A) = 2$ for string $T = ATGACGGATCA\$$. If $i \leq 0$ then $rank(i, sym) = 0$.

To answer rank queries, BWT is encoded using the wavelet tree as follows:

1) Encode half the alphabet as 0, and other half as 1, for example:

$$\Sigma = \{\$, A, C, G, T\}$$
$$encode(\Sigma) = \{0, 0, 0, 1, 1\} \quad (2)$$

2) Represent every 0-coded symbols $\{\$, A, C\}$ as one sub-tree and 1-coded symbols $\{G, T\}$ as other sub-tree.
3) Encoding and branching is applied to each sub-tree until only one symbol remains.

Fig. 7 shows the Wavelet Tree encoding for $T = ATGACGGATCA\$$ BWT. After tree construction, rank query on WT is performed using $\log \upsilon$ binary rank queries on the bit vectors.

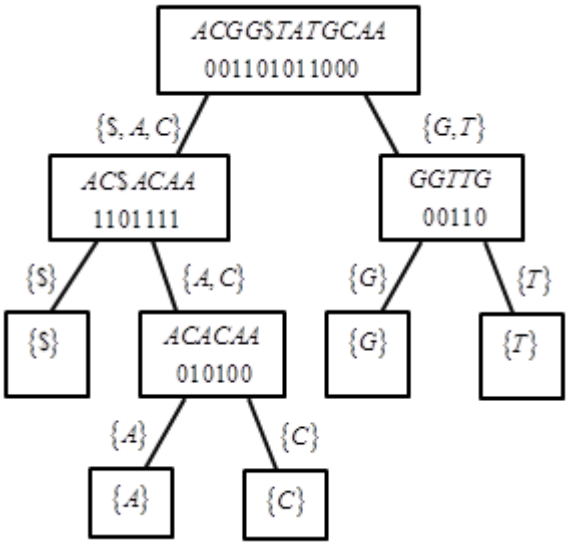

Fig. 7. Wavelet Tree for BWT of string $T = ATGACGGATCA\$$

D. RRR Sequence

Raman [10] purposed a method to encode bit sequence that answers the binary rank queries in *O(1)* time. Resultant sequence is known as RRR sequence. It also provides implicit compression. Given the wavelet tree, bit vectors corresponding to its node are encoded as RRR sequences.

V. Results & Discussion

We have implemented FM-Index in C++. For SAIS and DivSufSort, we have taken the code available at [12] and [13]. For Wavelet Tree and RRR sequence, libcds library is used. All the programs are compiled in gcc with following system specifications: 3.6 GHz Intel Core i3, 4GB RAM, and Ubuntu 14 operating system.

To evaluate the performance of implemented index structure, dataset available at [7] is used. It consists of three categories: Artificial Strings, DNA Sequences, and Real World Strings. Table 1 shows the strings in our dataset. It can be seen that we have taken into account strings of varying size and alphabets. For each sample string, each index algorithm is executed 10 times and average execution time is measured.

TABLE I. DATASET

Dataset			
Category	**Name**	**Size**	**Alphabets**
Artificial Strings	Fibonacci	20000000	2
	period_1000	20000000	26
	random	20000000	26
DNA Sequence	3Ecoli.dna	14776363	5
	4Chlamydophila.dna	4856123	6
	H_sapiens_Chr22.dna	34553758	5
Real World Strings	world	2473399	94
	jdk_50M	50000000	110
	howto	39422104	197
	jdk	69728898	113
	gcc	86630400	150

TABLE II. AVERAGE EXECUTION TIME OF SUFFIX ARRAYS AND FM-INDEX FOR DATASET STRINGS

Name	**DivSufSort SA**	**SAIS SA**	**DivSufSort FMI**	**SAIS FMI**
Fibonacci	2.72	1.29	6.48	5.01
period_1000	1.09	1.38	7.78	8.04
Random	2.07	2.81	10.02	10.79
3Ecoli.dna	1.36	1.49	5.11	5.22
4Chlamydophila.dna	0.41	0.44	1.64	1.67
H_sapiens_Chr22.dna	3.01	3.72	11.92	12.63
World	0.15	0.18	1.25	1.29
jdk_50M	3.62	4.43	25.58	26.42
Howto	3.10	4.26	22.86	24.13
Jdk	5.32	6.06	35.98	36.59
Gcc	6.12	8.24	49.26	51.09

Table 2 shows the average execution time of both suffix arrays (DivSufSort & SAIS) and their corresponding FM-Indexes. It can be seen that DivSufSort suffix array and its FM-Index takes less execution time for 10 out of 11 strings in dataset compared to SAIS suffix array and its FM-Index. The minimum execution time of 0.15 for DivSufSort, 0.18 for SAIS, 1.25 for DivSufSort-based FM-Index, and 1.29 for SAIS-based FM-Index is observed for "world" string while the maximum execution time of 6.12 for DivSufSort, 8.24 for SAIS, 49.26 for DivSufSort-based FM-Index, and 51.09 for SAIS-based FM-Index is observed for "gcc". The reason DivSufSort & its FM-Index performed better than SAIS & its FM-Index is that DivSufSort is hybrid approach using Itoh-Tanaka's two-stage (ITTS) algorithm for character representation and KA & MSufSort algorithms for suffix sorting while SAIS uses LMS substrings and induced sorting for suffix array construction. If induced sorting is optimized in terms of time or rigorous characterization and analysis is conducted as implemented in [18], it can outperform DivSufSort suffix array and correspondingly FM-Index.

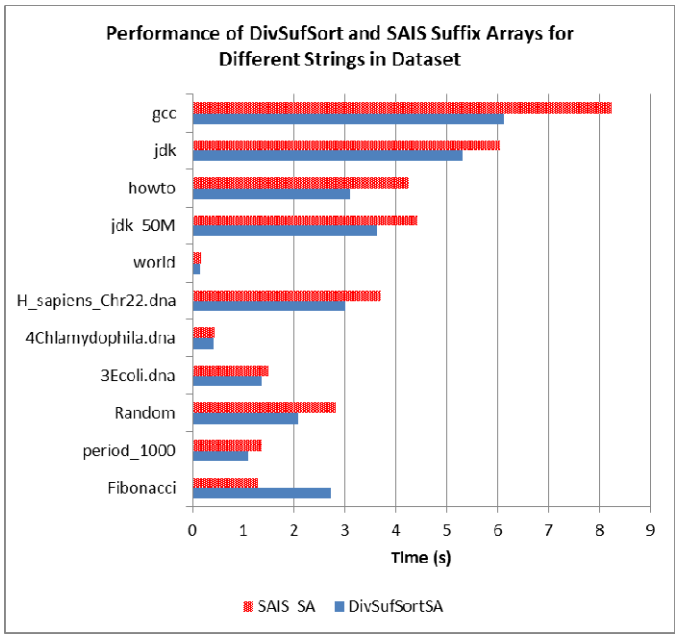

Fig. 8. Performance Comparison of Suffix Arrays for Strings in Dataset

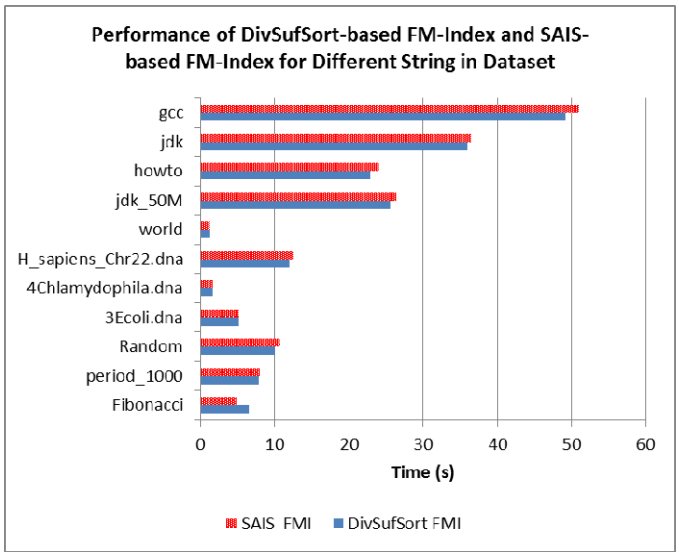

Fig. 9. Performance Comparison of FM-Indexes for Strings in Dataset

The trend in terms of time efficiency of two suffix arrays with respect to different strings is shown in Fig. 8 while the

trend in terms of time efficiency of two FM-Indexes with respect to different strings is shown in Fig. 9. It can be seen that DivSufSort Suffix Array performs 18.21% time efficient than SAIS Suffix Array while DivSufSort-based FM-Index performs 3.67% time efficient than SAIS-based FM-Index on 10 out of 11 strings in the dataset.

VI. Conclusion

To find information & infer patterns in large texts and DNA sequences, FM-Index is used. We have implemented two FM-Index data structures using DivSufSort and SAIS suffix sorting algorithms. BWT of given string is encoded using RRR wavelet trees to solve rank queries efficiently in *O(1)* time. Our results show that DivSufSort-based FM-Index performs 3.67% better than SAIS-based FM-Index on 10 out of 11 strings in dataset.

References

[1] Manber, Udi, and Gene Myers. "Suffix arrays: a new method for on-line string searches." *SIAM Journal on Computing* 22, no. 5 (1993): 935-948.

[2] Gonnet, Gaston H., Ricardo A. Baeza-Yates, and Tim Snider. "New Indices for Text: Pat Trees and Pat Arrays." Eds.: Information Retrieval: Data Structures & Algorithms, Prentice-Hall (1992): 66-82.

[3] Culpepper, J. Shane, Gonzalo Navarro, Simon J. Puglisi, and Andrew Turpin. "Top-k ranked document search in general text databases." In *Algorithms–ESA 2010*, pp. 194-205. Springer Berlin Heidelberg, 2010.

[4] Ferragina, Paolo, and Giovanni Manzini. "Opportunistic data structures with applications." In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium* on, pp. 390-398. IEEE, 2000.

[5] Abouelhoda, Mohamed Ibrahim, Stefan Kurtz, and Enno Ohlebusch. "Replacing suffix trees with enhanced suffix arrays." *Journal of Discrete Algorithms* 2, no. 1 (2004): 53-86.

[6] Flicek, Paul, and Ewan Birney. "Sense from sequence reads: methods for alignment and assembly." *Nature methods* 6 (2009): S6-S12.

[7] Schürmann, Klaus-Bernd, and Jens Stoye. "An incomplex algorithm for fast suffix array construction." *Software: Practice and Experience* 37, no. 3 (2007): 309-329.

[8] Ferragina, Paolo, and Giovanni Manzini. "Indexing compressed text." *Journal of the ACM (JACM)* 52, no. 4 (2005): 552-581.

[9] Deogun, Jitender S., Jingyi Yang, and Fangrui Ma. "EMAGEN: An efficient approach to multiple whole genome alignment." In *Proceedings of the second conference on Asia-Pacific bioinformatics-Volume* 29, pp. 113-122. Australian Computer Society, Inc., 2004.

[10] Raman, Rajeev, Venkatesh Raman, and S. Srinivasa Rao. "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets." In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 233-242. Society for Industrial and Applied Mathematics, 2002.

[11] Puglisi, Simon J., William F. Smyth, and Andrew H. Turpin. "A taxonomy of suffix array construction algorithms." *ACM Computing Surveys (CSUR)* 39, no. 2 (2007): 4.

[12] Mori, Y. "SAIS: An implementation of the induced sorting algorithm." (2008).

[13] Mori, Y. "Short description of improved two-stage suffix sorting algorithm." (2005).

[14] Grossi, Roberto, Ankur Gupta, and Jeffrey Scott Vitter. "High-order entropy-compressed text indexes." *In Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 841-850. Society for Industrial and Applied Mathematics, 2003.

[15] Burrows, Michael, and David J. Wheeler. "A block-sorting lossless data compression algorithm." *Technical report* 124, Palo Alto, CA, Digital Equipment Corporation (1994).

[16] Li, Heng, and Richard Durbin. "Fast and accurate short read alignment with Burrows-Wheeler transform." *Bioinformatics* 25, no. 14 (2009): 1754-1760.

[17] Langmead, Ben, and Steven L. Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature methods* 9, no. 4 (2012): 357-359.

[18] Timoshevskaya, Nataliya, and Wu-chun Feng. "SAIS-OPT: On the characterization and optimization of the SA-IS algorithm for suffix array construction." In *Computational Advances in Bio and Medical Sciences (ICCABS), 2014 IEEE 4th International Conference on*, pp. 1-6. IEEE, 2014.

[19] Grabowski, Szymon, Gonzalo Navarro, RAFAŁ PRZYWARSKI, Alejandro Salinger, and Veli Mäkinen. "A simple alphabet-independent FM-index." *International Journal of Foundations of Computer Science* 17, no. 06 (2006): 1365-1384.

[20] Przywarski, Rafal, Szymon Grabowski, Gonzalo Navarro, and Alejandro Salinger. "FM-KZ: An even simpler alphabet-independent FM-index." In *Stringology*, pp. 226-241. 2006.

[21] Mäkinen, Veli, and Gonzalo Navarro. "Succinct suffix arrays based on run-length encoding." In *Combinatorial Pattern Matching*, pp. 45-56. Springer Berlin Heidelberg, 2005.

[22] Belazzougui, Djamal, Travis Gagie, Simon Gog, Giovanni Manzini, and Jouni Sirén. "Relative FM-Indexes." In *String Processing and Information Retrieval,* pp. 52-64. Springer International Publishing, 2014.

[23] Larsson, N. Jesper. "The context trees of block sorting compression." In *Data Compression Conference, 1998. DCC'98*. Proceedings, pp. 189-198. IEEE, 1998.

[24] Sadakane, Kunihiko. "A fast algorithm for making suffix arrays and for Burrows-Wheeler transformation." In *Data Compression Conference, 1998. DCC'98*. Proceedings, pp. 129-138. IEEE, 1998.

[25] Larsson, N. Jesper, and Kunihiko. Sadakane, "Faster suffix sorting", *Technical Report LU–CS–TR: 99–214, LUNDFD6/(NFCS–3140)/1–20/(1999)*, Dept. of Computer Science, Lund University, 1999.

[26] Larsson, N. Jesper, "Structures of string matching and data compression", PhD thesis, Dept. of Computer Science, Lund University, 1999.

[27] Ko, Pang, and Srinivas Aluru. "Space efficient linear time construction of suffix arrays." In *Combinatorial Pattern Matching*, pp. 200-210. Springer Berlin Heidelberg, 2003.

[28] Li, Heng. "Fast construction of FM-index for long sequence reads". *Bioinformatics* (2014): btu541.

[29] Itoh, Hideo, and Hozumi Tanaka. "An efficient method for in memory construction of suffix arrays." In *String Processing and Information Retrieval Symposium, 1999 and International Workshop on Groupware*, pp. 81-88. IEEE, 1999.