A Practical Implementation of Compressed Suffix Arrays with Applications to Self-indexing

Hongwei Huo[*+], Longgang Chen[*], Jeffrey Scott Vitter[+] and Yakov Nekrich[+]

[*]*Xidian University*
No.2 Taibai South Road
Xi'an, Shaanxi 710071, China
{hwhuo,lgchen}@mail.xidian.edu.cn

[+]*The University of Kansas*
1450 Jayhawk Blvd.
Lawrence, KS 66045, USA
{jsv, yakov}@ittc.ku.edu

***Abstract**:* In this paper, we develop a simple and practical storage scheme for compressed suffix arrays (CSA). Our CSA can be constructed in linear time and needs $2nH_k + n + o(n)$ bits of space simultaneously for any $k \leq c\, log_\sigma n - 1$ and any constant c <1, where H_k denotes the k-th order entropy. We compare the performance of our method with two established compressed indexing methods, viz. the FM-index and the Sad-CSA. Experiments on the Canterbury Corpus and the Pizza&Chili Corpus show significant advantages of our algorithm over two other indices in terms of compression and query time. Our storage scheme achieves better performance on all types of data present in these two corpora, except for evenly distributed data, such as DNA. The source code for our CSA is available online.

1. Introduction

Suffix trees[9,10] and suffix arrays[1] are versatile data structures playing a key role in numerous string processing applications in such areas as string matching, information retrieval, genome analysis, and text compression. Both the suffix tree and the suffix array support pattern matching queries in optimal or almost-optimal time and use linear space of $O(n \log n)$ bits. However in practice these data structures occupy 5 to 20 times more space than the raw string data; the latter needs only $n \log \sigma$ bits of space, where σ denotes the alphabet size.

The compressed suffix array (CSA) [2,11,12,7,8] and the FM-index [4,13,14] overcome the space limitation by exploiting the text compressibility and index regularities, while supporting the functionalities of suffix arrays and suffix trees. A powerful concept is that of a self-index, which requires space close equal to the space occupied by the input data in compressed format.

Grossi and Vitter [2,11] introduced the compressed suffix array (GV-CSA), which uses $O(n \log \sigma)$ bits* of space and answers string matching queries in $o(p/\log_\sigma n + occ \log_\sigma n)$ time, where p is the length of the query pattern P and occ denotes the number of times P occurs in the source string. Sadakane [7,8] showed how to convert the GV-CSA into a self-index and the resulting index, called Sad-CSA, needs $\frac{1}{\epsilon}nH_0 + O(n \log\log \sigma) + \sigma \log \sigma$ bits of space and answers queries in $O(p \log n + occ \log^\epsilon n)$ time, where $0 < \epsilon \leq 1$ is an arbitrary constant. Henceforth H_k for $k \geq 0$ denotes the k-th order empirical entropy of the source string T. Ferragina and Manzini designed the FM-index that relies on the Burrows-Wheeler transform (BWT) [15] and the backward searching approach[3,4]. Their original index uses at most $5nH_k(T) + o(n \log \sigma)$ bits of space for $k \leq$

* Logarithms in this paper are in base 2 unless otherwise stated.

$\log_\sigma(n / \log n) - \omega(1)$ and retrieves the *occ* occurrences of a pattern P within a text T in $O(p + occ \log^{1+\epsilon} n)$ time for any $0 < \epsilon < 1$.

Grossi et al. [3] gave the first self-index that provably achieves asymptotic space optimality, i.e., with constant factor of 1 in the leading term. It uses $nH_k(T) + o(n)$ bits of space and achieved $O(p \log \sigma + occ(\log^4 n)/((\log^2 \log n) \log \sigma))$ query time. The analysis also applies to the FM-index. Foschini et al. [16] reported on a new method of storage for entropy-compressed suffix arrays; their method retains the theoretical performance of previous works and achieves good results in practice. Ferragina et al. [17] investigated the existing implementations of compressed indexes from a practitioner's point of view. Many further results and considerable improvements appeared in the literature; we refer the reader to the survey of Navarro and Mäkinen [18].

In this article we develop a practical representation of compressed suffix arrays that uses $2nH_k+n+o(n)$ bits of space simultaneously for any $k \leq c \log_\sigma n - 1$ and any constant $c < 1$. Moreover, we present a linear time construction algorithm for our storage scheme. We evaluate our CSA using compression ratio, compression time, and query time as performance measures. Our results are compared with the FM-index [4,14] and the Sad-CSA [7,8] on the *Canterbury Corpus* and the *Pizza&Chili Corpus*. The source code for our CSA is available at *https://sites.google.com/site/compressedsa/*.

It turns out that the distribution of alphabet symbols is more important than the alphabet size for comparative performance of our method. All three algorithms suit well for dealing with text data and metadata, such as English texts, sources, and XML, but our CSA shows the best performance. For more evenly distributed data, such as DNA and protein sequences, the FM-index performs best. For all types of data, our CSA outperforms the SAD-CSA in terms of compression rate. For counting and locating queries, our CSA has a significant advantage over the FM-index. For counting queries, our CSA outperforms the Sad-CSA. For locating queries, our CSA has a slight advantage over the Sad-CSA, except for the DNA and sources.

This paper is organized as follows. In section 2, we describe the basic ideas behind the compressed representation of suffix arrays, including the function Φ. Details of our structure for the representation of function Φ are given in section 3. In section 4 we show how to build our compressed index. In section 5, we give a technique to access Φ in almost-constant time. Algorithms for answering counting and locating queries are also described in this section. In section 6, we evaluate the performance of our CSA compare it with the FM-index and Sad-CSA.

2. Preliminary

2.1 Suffix arrays

Let text T be a (long) string of length n and pattern P a (comparatively short) string of length m, both over alphabet Σ of size σ. A prefix of T is a substring of the form $T[1..j]$, where $1 \leq j \leq n$, and a suffix of T is a substring of the form $T[k..n]$, where $1 \leq k \leq n$. A suffix array, denoted SA, is a permutation of all the suffixes of T so that the suffixes are lexicographically sorted. $SA[i] = j$ means that the suffix $T[j..n]$ starting at the position j in T ranks the i-th smallest among all the n suffixes. All the suffixes prefixed by P occupy a contiguous range in the sorted array *SA*. Thus we can search for the interval [*L*, *R*] of *SA* containing the suffixes prefixed by *P* by two binary searches on *SA*, which is what necessary for counting and locating queries.

2.2 Compressed suffix arrays

Grossi and Vitter [2,11] introduced the compressed suffix array (GV-CSA), which settled the open problem of whether it is possible to simultaneously achieve fast query performance and break the (n log n)-space barrier. GV-CSA use a hierarchical decomposition of *SA* based on the neighbor function Φ defined as follows.

$$\Phi(i) = j, \text{ if } SA[j] = (SA[i] + 1) \bmod n \qquad (1)$$

In GV-CSA structure, the major challenge is how to efficiently represent and store Φ. Based on the following observations we propose a practical representation of Φ.

We conceptually group all the suffixes based on the 1-symbol prefix, say x, of each of the suffixes, so that all suffixes within a group start with character x. We call this conceptual group an *x-group*. For the example in Figure 1, the first four smallest suffixes correspond to the *a-group*, since these suffixes have the first character *a*. Obviously, the 1-symbol prefix, x, of the suffixes corresponding to the x-group does not contribute to the relative order of the suffixes. If we remove the 1-symbol prefix x from the suffixes in an x-group, then the relative order of the resulting suffixes is not changed, though they may be no longer in a contiguous segment. We call the collection of the resulting suffixes obtained from the x-group an *x-list*. In other words, if an *x-group* contains suffix positions $k = SA[i]$ and $h = SA[i+1]$ (corresponding to suffixes $T[k..n]$ and $T[h..n]$ respectively, and $T[k] = T[h] = x$), and $T[SA[p]..n] = T[k+1..n]$ and $T[SA[q]..n] = T[h+1..n]$, then $p < q$. For example, in Figure 1, the *a*-group contains four suffixes with ranks 0, 1, 2, 3, respectively. All these suffixes start with *a*. We remove *a* from these suffixes and obtain four new suffixes, whose rankings form an increasing sequence of positions, namely, 6, 14, 17 and 23. The collection of these *x-list*s is exactly the Σ lists introduced by Grossi and Vitter [2,11].

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
T	a	b	f	g	d	b	f	b	g	d	f	c	c	b	g	a	c	e	f	c	e	g	c	d	e	f	g	b	f	c	a	d	b	g	a	f
SA	0	15	30	34	5	27	1	13	32	7	29	12	11	22	16	19	4	31	23	9	17	24	20	35	6	28	10	18	25	2	14	33	26	21	3	8
Φ	6	14	17	23	24	25	29	30	31	35	2	7	11	18	20	22	4	8	21	26	27	28	33	0	9	10	12	15	32	34	1	3	5	13	16	19

Figure 1: Suffix array *SA* and neighbor function Φ

3. A Simple CSA Structure

3.1 Representing neighbor function Φ

Grossi et al.[3] gave the first self-index that provably achieved asymptotic space optimality. They used the notion of Σ lists (hereafter known as x-lists) that partition the suffixes and associated values of Φ according to their prefixes. These x-lists can be obtained by partitioning the suffix pointers according to their prefixes of length 2^k for k = 0,1,… . While Grossi et al. .[3] apply this approach recursively, in our case only the first level of recursion is used. Our partitioning is always based on prefixes of length 1. The simple concatenation of these x-lists is exactly what is needed to represent neighbor function Φ. Recall that Φ maps a position i in *SA*, such that *SA[i]=p* for some p, into the position j, such that *SA[j]= p +1*.

As we discussed in section 2.2, each x-list forms an increasing sequence of positions from text. Thus, compression is achieved since we can encode the lengths of gaps. Following in order to facilitate the discussion, we use the neighbor function Φ and the array Φ alternatively.

Inspired by the context-based partition scheme of GGV[3] (here the 1-symbol prefix plays the role of a context) and directory methods [5,19], our idea is to partition the x-lists into equal size superblocks and blocks; this partitioning is combined with sampling of Φ and Elias gamma encoding, to represent neighbor function Φ. The following three main steps are employed to obtain a succinct representation of Φ:

Step 1. We partition the array Φ of size n into superblocks of size $a = O(\log^3 n)$ each.

Step 2. Then we divide each superblock into r contiguous blocks of size $b = O(\log^2 n)$.

Step 3. We encode the differences $\Phi(i) - \Phi(i-1)$ of two adjacent entries using Elias gamma encoding, except for the first entry in each block, which is assumed to be 0 for the sake of gamma coding; see Figure 2. We build S by concatenating the differential Elias gamma encodings of all T's blocks. In other words, S is the string obtained by encoding the gap sequence via Elias gamma encoding and simple concatenation of the Elias gamma encoding representation.

However, we also have to handle the case when *gap* is negative, that is, $gap = \Phi[i] - \Phi[i-1] < 0$. In this case, we set $gap = \Phi[i] - \Phi[i-1] + n$, since $(x - z) \bmod n = (x + n - z) \bmod n$, where x and z are nonnegative integers.

To provide access to the encoded sequence S and to ensure their decoding, we keep three table structures *SB*, *B* and *SAM*. $SB[0, \frac{n}{a}-1]$ stores the starting position of the encoding of each superblock in S, i.e., the total number of bits in superblocks preceding the current superblock. $B[0, \frac{n}{b}-1]$ stores the starting position in S of the encoding of every block relative to the beginning of its enclosing superblock. $SAM[0, \frac{n}{b}-1]$ contains sampling values of Φ, so that the first value in each block is stored. An example in Figure 2 illustrates the representation of Φ by the three above structures and an encoded sequence S. The gap sequence is conceptual, it is used only for illustration purposes. In Figure 2, the text size $n = 36$, the superblock size $a = 9$, the number of blocks in a superblock $r = 3$ and the block size $b = 3$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
Φ	6	14	17	23	24	25	29	30	31	35	2	7	11	18	20	22	4	8	21	26	27	28	33	0	9	10	12	15	32	34	1	3	5	13	16	19
gap	0	8	3	0	1	1	0	1	1	0	3	5	0	7	2	0	18	4	0	5	1	0	5	3	0	1	2	0	17	2	0	2	2	0	3	3
S	0001000011			11			11			01100101			00111010			00001001000100			001011			00101011			1010			000010001010			010010			011011		
SB	0									14									43									61								
B	0			10			12			0			8			16			0			6			14			0			12			18		
SAM	6			23			29			35			11			22			21			28			9			15			1			13		

Figure 2: The representation of Φ ($n = 36$, $a = 9$, $r = 3$ and $b = 3$)

All suffixes in each x-list of Φ have the same 1-symbol prefix and the values of Φ within a list form an increasing sequence of positions. *gap* is obtained by calculating the difference between two consecutive entries $\Phi[i]$ and $\Phi[i-1]$ relative to a block. That is, $gap[i] = \Phi[i] - \Phi[i-1]$ if $\Phi[i] \geq \Phi[i-1]$ and $\Phi[i] - \Phi[i-1] + n$ otherwise. We do not need to store the difference value for the first entry of each block, marked with 0, since the first entry is stored in *SAM*. Notice that special cases occur in the 4-th, 6-th and 8-th blocks, where the difference is negative. In the example of the 4-th block, we assign 3 to $gap[10]$, since $2-35+36 = 3$. Thus the corresponding encoding is 011.

SB represents the starting position of the encoding of every superblock in *S*, and also the offset relative to the superblock in *S*. For example, the encoding of the second difference 3 (as the first Φ is sampled and does not have to be stored) in the second superblock has the offset 14 in *S*, since $SB[1] = 14$.

To access $\Phi[i]$, we first query *SB* and *B* to determine the position of the block in *S*, then, decode the corresponding sequence of *S*. We obtain the final value of $\Phi[i]$ by adding the sampling of the block, $SAM[i]$ to the value of decoding. The formula is given in (2).

$$\Phi[i] = \left(SAM\left[\left\lfloor\frac{i}{b}\right\rfloor\right] + decompress\left(S, SB\left[\left\lfloor\frac{i}{a}\right\rfloor\right] + B\left[\left\lfloor\frac{i}{b}\right\rfloor\right], i \bmod b\right)\right) \bmod n \qquad (2)$$

where a and b is the *superblock* size and *block* size, respectively. The operation *decompress* performs decoding of the encoded sequence *S*. The starting position for decoding is determined by the second parameter $SB[\lfloor i/a \rfloor] + B[\lfloor i/b \rfloor]$ of *decompress*, the beginning position for the i-th value in that block. $i \bmod b$ is the number of gaps to be decoded.

To retrieve $\Phi[20]$ in the example of Figure 2, we find that $SAM[\lfloor 20/b \rfloor] = SAM[6] = 21$. We have $SB[\lfloor 20/a \rfloor] + B[\lfloor 20/b \rfloor] = SB[2] + B[6] = 43$ and $20 \bmod 3 = 2$. Thus we start at the 43-th position of *S*, decode two successive gaps, and add them to $SAM[6]$ to get the desired result. The first and second gaps are 5 and 1. Thus, $\Phi[20] = (21 + 5 + 1) \bmod 36 = 27$.

3.2 Space Occupancy

Let g_j be the length of the j-th gap within an x-list, and n_x the number of entries in the list. By the definition of the x-list, we know that n_x is exactly the number of occurrences of the character x in the text. We use Elias gamma coding to encode $g_j = \Phi(j+1) - \Phi(j)$. The length of the encoding has $2\lfloor \log g_j \rfloor + 1$ bits. Thus the overall length of the gaps within the list can be bounded by

$$\sum_{j=1}^{n_x-1} |\Phi(j+1) - \Phi(j)| \leq n \qquad (3)$$

Thus the space required to encode all the gaps inside a list is bounded by $\sum_{j=1}^{n_x-1}(2\log(\Phi(j+1) - \Phi(j)) + 1)$ in bits, which is $2n'_x\log(n/n'_x) + n'_x$ in the worst case, where $n'_x = n_x - 1$.

By summing over all the lists, we get the following bound on the space within all the lists

$$\sum_{x\in\Sigma}(2n'_x\log(n/n'_x) + n'_x) \qquad (4)$$

which is $2nH_1 + n'$ in bits, since each x-list corresponds to suffixes which are preceded with x, where $n' = n - \sigma$.

We can obtain a better bound by a more careful analysis of encoding the gaps within lists using the method from [16,20]. Let the k-context of $\Phi(i)$ denote the length k prefix of $SA[\Phi(i)]$. Every list can be subdivided into at most σ^k sublists according to k-contexts. It can be shown that $\sum \log(\Phi(j+1) - \Phi(j)) \leq nH_k$, where the sum is taken over all gaps that are within the same list and within the same context. The number of gaps that are not within the same context can be bounded by $\sigma^{k+1.}$ The total length of encoding all gaps within the same context is bounded by $2nH_k + n$ and the total length of encoding all other gaps is bounded by $2\sigma^{k+1}\log n$. It is worth to be mentioned that this analysis holds for any k simultaneously. We refer to [16] for details.

However, we have to deal with one case. There are some gaps (the first entry in each block) that we do not need to encode, because they are sampled. We need to remove space occupied by these gaps. The space occupied sums to $\frac{n}{b}(2\log g_i + 1)$, which is bounded by $o(n)$ bits, where block size $b = \log^2 n$.

Let us now consider the space required by tables *SB*, *B*, and *SAM*. Suppose that the superblock size $a = \log^3 n$ and block size $b = \log^2 n$. Note that the starting position of each superblock is not larger than $|S| = O(\frac{n}{a}\log(nH))$, where H is the average number of bits per symbol using Elias gamma coding to the gap sequence, whereas the relative position of each block within its superblock is $|b| = O(\log^3 n)$. We also need to store $O(\frac{n}{b}) = \frac{n}{\log^2 n}$ absolute samples of Φ, each of $\log n$ bits. Consequently, the overall space required by tables *SB*, *B*, and *SAM* is $O(\frac{n}{a}\log|S| + \frac{n}{b}\log|b| + \frac{n}{b}\log n)$ bits. Substituting the values $a = \log^3 n$ and $b = \log^2 n$, we get the following bound on the space:

$$O(\frac{n}{\log^3 n}\log(\frac{n}{\log^3 n}\log(nH)) + \frac{n}{\log^2 n}\log(\log^3 n) + \frac{n}{\log^2 n}\log n) = o(n)$$

Now we put the space for each piece all together and get the following bound on the total space: $2nH_k + n + 2\sigma^{k+1}\log n + o(n)$. Then we have the following theorem:

Theorem 1. Our representation for Φ requires at most $2nH_k + n + o(n)$ bits in the worst case for any $k \leq c\,log_\sigma n - 1$ and any constant $c < 1$, where H_k denotes the k-th order entropy.

3.3 Character frequency statistics

Character frequency statistics are used to support the self-index. We create the table C with entries corresponding to alphabet symbols in lexicographical order, so that $C[x]$ is the sum of the number of occurrences of characters a, satisfying $a < x$ in T. In other words, $C[x]$ represents the ranking of the smallest suffix among the suffixes with 1-symbol prefix x in the lexicographic order.

For the example given in Figure 1, we show C in table 1. For convenience, we add one more entry at the end of the table and write in n.

Table 1. The table C

C	a	b	c	d	e	f	g	
	0	4	10	16	20	23	30	36

Notice that all the suffixes $SA[C[c]+1...C[c+1]]$ start with character c. It is obvious to see that the space required by the table C is $\sigma \log n$.

With the function Φ and the table C, we can restore the suffix $T[SA[i]\dots n-1]$ corresponding to $SA[i]$. Thus the text can be discarded. Because the first characters $T[SA[i]]$ of the suffixed $T[SA[i]..n]$ for $i = j$, $\Phi[j]$, $\Phi[\Phi[j]]\dots$, are in alphabetic order in SA, $T[SA[i]]$ must be the character c such that $C[c] < i \le C[c+1]$.Thus each $T[SA[i]]$ can be found by an $O(\log \sigma)$-time binary search on C.

4. Index Construction

We build our compressed suffix arrays in $O(n)$ time in the following four steps.

Step 1. (Preprocessing). Computing C, and SA.
Step 2. (Constructing Φ). Using C, SA, T to construct Φ, discarding T after that.
Step 3. (Sampling). Sampling SA and SA^{-1} at regular step, c, discarding SA after that.
Step 4. (Encoding Φ). Sampling and encoding Φ. Using S, SB, B, and SAM to represent Φ, discarding Φ after that.

In the following, we explain how each of the last three steps work. We omit *Step* 1 because the suffix array can be constructed using a standard method and the rest of *Step* 1 is trivial.

4.1 Constructing neighbor function Φ

The pseudocode of the algorithm *constructPhi* for the construction of Φ is given below.

```
constructPhi(C, SA, T, Φ)
Input: C, SA, T
Output: Φ
1  temp ← C[lastchar]
2  for i ← 0 to n −1 do
3          pos ← SA[i]
4          if pos = 0 then
5                  h ← i
6          else
7                  c ← T[pos −1]
8                  Φ[C[c]] ← i
9                  C[c] ← C[c] + 1
10 Φ[temp] ← h
```

Line 1 keeps the ranking of the suffix $T[n-1]$ in *temp*, i.e., $SA[temp] = n - 1$, where *lastchar* is the last character in T. Lines 4–5 keep the value of i satisfying $SA[i] = 0$ in h. By (1), $\Phi[temp] = h$, since $SA[h] = (SA[temp] + 1) \bmod n = 0$. In the example of Figure 1, $\Phi[23] = 0$, because $h = 0$ since $SA[0] = 0$ and $temp = 23$ since $C[lastchar] = C[f] = 23$. Obviously, *constructPhi* runs in $O(n)$ time. The *constructPhi* algorithm is based on the following observations. Suppose we traverse the suffix array and reach $SA[i] = j$. If $c = T[j-1]$ and c precedes a suffix for the k-th time, then $c\text{-}list[k] = i$, i.e., $\Phi[C[c]] = i$. In the pseudocode description we assume that the array *C*[] is a local variable and values of *C* are re-set when *constructPhi* is finished.

4.2 Sampling suffix arrays

The sampled *SA*s, denoted SA_l, are used to restore the unsampled *SA*s. During the sampling, SA_l and SA^{-1}_l are built, where SA_l and SA^{-1}_l are the sampling points for *SA* and the *inverse SA*, respectively. Let the sampling steps for *SA* and SA^{-1} are *c* and *d*, respectively. The pseudocode of the sampling algorithm is given as follows.

```
samplingSA($SA$, $c$, $SA_l$, $SA_l^{-1}$)
Input: $SA$, $c$
Output: $SA_l$, $SA_l^{-1}$
1  $k \leftarrow \lceil (n-1)/c \rceil$
2  for $i \leftarrow 0$ to $k-1$ do
3          $SA_l[i] \leftarrow SA[c \cdot i]$
4  for $i \leftarrow 0$ to $n-1$ do
5          if ($SA[i]$ mod $d$) = 0 then
6                  $SA_l^{-1}[SA[i]/d] \leftarrow i$
```

Lines 2–3 build SA_l by sampling the suffix array at regular text position intervals, that is, they collect all entries $SA[c \cdot i]$ in which its index is a multiple of *c*, given a sampling step *c*. SA_l is used to determine $SA[i]$ when we perform a locating query, which is described in section 5.2.

Lines 4–6 build SA_l^{-1} by sampling the inverse suffix array, which is used to support *extract*(*start*, *len*), returning $T[start..start+len-1]$. According to section 3.3, given a ranking of a suffix, it is not hard to restore the suffix. So how to transform the position *start* into the ranking *i* becomes the only hurdle. We first determine the ranking *i* of *start*/*d*, and then perform $i = \Phi[i]$ repeatedly. After obtaining the ranking, we can restore $T[start .. start + len -1]$ by the solution mentioned in section 3.3.

4.3 Encoding neighbor function Φ

codingPhi initializes the four structures of *S*, *SB*, *B* and *SAM*. Assume that the superblock size and block size are *a* and *b* respectively, where *a* is a multiple of *b*. The pseudocode of the algorithm is given as follows.

```
codingPhi($\Phi$, $S$, $SB$, $B$, $SAM$)
Input: $\Phi$
Output: $S$, $SB$, $B$, $SAM$
1  Initialize $index_1$, $index_2$, $index_3$, $len_1$, and $len_2$ to be 0
2  for $i \leftarrow 0$ to $n-1$ do
3          if ($i$ mod $a$) = 0 then
4                  $len_2 \leftarrow len_1$
5                  $SB[index_3] \leftarrow len_2$
6                  $index_3 \leftarrow index_3 + 1$
7          if ($i$ mod $b$) = 0 then
8                  $SAM[index_1] \leftarrow \Phi[i]$
```

```
9             $index_1 \leftarrow index_1 + 1$
10            $B[index_2] \leftarrow len_1 - len_2$
11            $index_2 \leftarrow index_2 + 1$
12            $pre \leftarrow \Phi[i]$
13        else
14            $gap \leftarrow \Phi[i] - pre$
15            if $gap < 0$ then
16                $gap \leftarrow gap + n$
17            $pre \leftarrow \Phi[i]$
18            $len_1 \leftarrow len_1 + 2bl(gap) - 1$
19            $append(gap, S)$
```

When the **if** condition holds in line 3, it corresponds to a sampling point of a superblock and we need to compute the number of bits in its previous superblocks in the gamma-encoded sequence relative to the current superblock. $index_3$ is the index of superblocks. When the **if** condition holds in line 7, it corresponds to a sampling point of a block and the value of Φ at the sampling point is kept in *SAM*. This can be done in line 8. Also we need to compute the point's offset relative to its superblock, and in fact it is $len_1 - len_2$, where len_1 is the length (number of bits) of current *S* and len_2 is the absolute offset that the sampling point belongs to the superblock. Lines 12 and 17 perform the update on *pre*, which represents the previous value of Φ. Lines 12–17 compute gap and encode Φ, where $bl(gap)$ expresses the length of binary coding for *gap*. The operation *append* is responsible for appending the Elias gamma encoding of *gap* to *S*. Obviously, the algorithm *codingPhi* runs in $O(n)$ time.

5. Pattern Matching

In this section, we consider two types of string matching queries: counting and locating. A counting query computes the number of occurrences of *P* in *T*, where *P* is a pattern string of length m. Essentially a counting query identifies the range $[L,R]$ of suffixes that start with *P*. Once a counting query is answered, we can find the starting position of every single suffix in $[L,R]$ by answering a locating query. Thus each locating query reports the position where *P* occurs in *T*. We first describe the key operation of fast access to Φ. We omit the description of the extracting query because of space limitation.

5.1 Accelerating access to Φ

Suppose we are given an encoded sequence *S* and the decoding starting position *p*. For a block of size $O(\log n)$, we need $O(\log n)$ time to access Φ in the worst case. By maintaining a precomputed table *R* of width $W = \frac{\log n}{2}$ bits, we can access Φ in almost-constant time. The table consists of four parts, denoted R_1, R_2, R_3 and R_4. R_1 stores the number of consecutive zeros that start from the leftmost in a bit sequence of length *W*. R_2 represents the number of gaps contained in a bit sequence of length *W* being decoded properly. R_3 represents the number of bits in a bit sequence of length *W* to be decoded properly. R_4 is the cumulative sum of decoded values correctly.

We can store R_1, R_2, and R_3 in a total of $3 \cdot 2^W \cdot \log W$ bits, since the entries in R_1, R_2, and R_3 are in the range [1, W] and require log W bits each. Substituting W with $\frac{\log n}{2}$ in the space occupancy above, we obtain $3\sqrt{n}(\log\log n - 1)$ bits required by the three parts. We can store R_4 in $(1/4)\sqrt{n}(\log n)$ bits, since its entries are at most $2^{W/2}$. In this case, the bit sequence consists of the first $W/2-1$ 0s, following $W/2$ 1s, ending with a 0 or 1. The second row in Table 2 shows an example of this case. Therefore, the total space required by the precomputed table is $3\sqrt{n}(\log\log n - 1) + (1/4)\sqrt{n}(\log n) = o(n)$ bits.

Table 2. The precomputed table R ($W = 16$)

index	R_1	R_2	R_3	R_4
…	…	…	…	…
0000000111111110	7	1	15	255
…				
0010101110100110	2	5	15	14
0010110010101110	2	5	15	15
…				
1010101001111001	0	7	13	11
…				
1111111111111111	0	16	16	16

Table 2 gives an example of the precomputed table R for $W = 16$. For instance, along the fourth row of the table for *index* = 0010101110100110, the number of gaps to be decoded completely in the bit sequence is 5, so we have $R_2 = 5$. The total number of bits being decoding of these gaps is $R_3 = 15$, and the cumulative decoded value, $R_4 = 14$, since there are five complete gaps in the first fifteen bits of the bit sequence and its corresponding sum of decoded values is 14. The rest of the bit sequence is not a complete gamma encoding.

Suppose that the sequence S is to be decoded and the current starting decoding position is at p, and the number of gaps to be decoded is 10. The decoding process using a precomputed table is given as follows.

$\downarrow p$ $\downarrow p+13$

S =…1 010 1 010 011 1 1 00101 011 1 010 011 00100 00111 01001100100 1…

Starting from the position p, we extract 16 bits from S and find that there are seven gaps to decode by querying R_2 (the seventh row). That is, the extracted bit sequence contains seven complete gamma encodings. Then we obtain the cumulative sum of these decoding values, 11, by querying R_4 and the number of bits to be decoded correctly, 13, by querying R_3. Next, we start decoding from the position p + 13 and the remaining number of gaps to decode is 3(10−7). The newly extracted bit sequence is 00101 011 1 010 011 0. We find that there are five gaps to decode in the bit sequence by querying R_2, but the number of gaps to be decoded is 3. In this case, we decode the newly extracted bit sequence three times and obtain three values of 5, 3, and 1, adding these values to 11, giving the 20 as a result of *decompress*(S, p, 10) in (2). So the access to Φ for each block of size $O(\log n)$ takes almost-constant time.

5.2 Search algorithms

Given the sampling suffix array SA_l and inverse suffix array $SA^{-1}{}_l$, combined with the table C described in section 3.3, our compressed suffix array can support three types of pattern matching queries: counting, locating queries, and extracting queries.

Counting query, denoted *count*, returns $[L, R]$ that contains all the occurrences of P in T, i.e., $R–L+1$. Locating query, denoted *locate*, reports occurring positions of P in T. Extracting query, denoted *extract*, displays the text substring $T[start, start + len-1]$ given *start* and *len*.

Count. The *count* uses the table C and neighbor function Φ to perform backward searching [6] of the pattern, given below.

```
count(P, L, R)
Input: P
Output: L, R
1  $c \leftarrow P[len-1]$
2  $L \leftarrow C[c]$
3  $R \leftarrow C[c+1] - 1$
4  **for** $i \leftarrow m-2$ **downto** 0 **do**
5        $c \leftarrow P[i]$
6        $LL \leftarrow C[c]$
7        $RR \leftarrow C[c+1] - 1$
8        $newL \leftarrow min\{j: j\in[LL, RR]$ and $\Phi[j]\in[L, R]\}$
9        $newR \leftarrow max\{j: j\in[LL, RR]$ and $\Phi[j]\in[L, R]\}$
10       $L \leftarrow newL$
11       $R \leftarrow newR$
12       **if** $L > R$ **then**
13             **return** "pattern does not exist"
14 **return** $L$ and $R$
```

Notice that the algorithm performs comparisons against P backwards. It is easy to see that the algorithm maintains the following invariant: after comparing the k-th character, $[L, R]$ contains suffixes prefixed by the last k characters of P.

Initialization: the algorithm runs lines 1–3. Obviously, at this time, L corresponds to the first entry in c-list and R corresponds to the last entry in c-list. Thus $[L, R]$ is exactly the interval of c-list and all suffixes in the interval have prefix c, so we are done.

Maintenance: Next, we show that each iteration maintains the loop invariant. Let us first suppose that the algorithm is in its k-th iteration and $[L, R]$ contains all suffixes that are prefixed by the last $k–1$ characters of P. The algorithm determines $[LL, RR]$ for c in lines 6–7 and the Φ values in the interval form an increasing sequence. In fact, these values are exactly the rankings of suffixes obtained by removing 1-symbol prefix from the suffixes in $[LL, RR]$. If T contains the pattern P, i.e., there is $p_{len-k}p_{len-k+1}p_{len-k+2}\dots p_{len-1}$ in T, then there must exist a contiguous segment $[newL, newR]$ in $[LL, RR]$ in which all Φ values are in $[L, R]$. This is because the suffixes in $[L, R]$ are prefixed by $p_{len-k+1}p_{len-}$

$_{k+2}...p_{len-1}$. Thus, after the iteration, [*L*, *R*] = [*newL*, *newR*], i.e., [*L*, *R*] contains suffixes prefixed by the last *k* characters of *P*.

Termination: we are done when the algorithm returns in line 14. Otherwise, the pattern does not exist in *T*.

We can perform binary searches on the absolute sampling values, SAM_c, of Φ within the range [*LL*, *RR*], since the suffixes in [*LL*, *RR*] start with *c* and thus these sampling values form a monotone increasing sequence. Assume that the interval corresponding to these sampling values is $[l_s, r_s]$. If we find a minimum $s \in [l_s, r_s]$ such that $SAM_c[s] \in [L, R]$ during the binary search, then we are done (similar fashion for line 9). Otherwise, when the binary search ends, there must exist a range of size *b* (between two consecutive sampling values) in which the Φ[*i*] is in. Then we decode the corresponding values using the precomputed table until finding the wanted entry. The binary search on the sampling values runs in $O(\log(n / \log^2 n))$ time, since there are at most $n / \log^2 n$ sampling values. The time needed to decode Φ[*i*] on *b* using the precomputed table is log *n* for the block size $b = O(\log^2 n)$. Thus the count algorithm runs in $O(m \log n)$ time.

Theorem 2. The counting query of a pattern of length *m* can be answered in $O(m \log n)$ time, at the extra cost of $\sigma \log n$ bits of space for *C*, except for the space required by representing Φ given in Theorem 1, where σ is the alphabet size.

We use pattern *P* = “bga” as an example to illustrate the above counting process.

After initialization in lines 1–3, we have *L* = *C*[a] = 0, *R* = *C*[a + 1] − 1 = *C*[b] − 1 = 3. Suffixes in [0, 3] start with a, corresponding to a-list. For the first iteration of the **while** loop, *LL* = *C*[g] = 30, *RR* = *C*[g + 1] − 1 = 35, Suffixes in [30, 35] are prefixed with g and the corresponding Φ values are {1, 3, 5, 13, 16, 19} for which Φ[30] = 1 and Φ[31] = 3 in [0, 3]. So, [*newL*, *newR*] = [30, 31]. We update [*L*, *R*] = [30, 31] in lines 10–11. The suffixes therefore in this interval are prefixed with ga. For the second iteration, *LL* = *C*[b] = 4, *RR* = *C*[b + 1] − 1 = 9, the Φ values in [4, 9] are {24, 25, 29, 30, 31, 35} in which Φ[7] = 30 and Φ[8] = 31 in [30, 31]. So, [*newL*, *newR*] = [7, 8] and updating *L* and *R* in lines 10–11.

locate. To locate the occurring positions, we proceed as follows: The counting query gives an interval in the suffix array to be reported. Now, given each position *i* within this interval, we use *getpos* to find the corresponding text position *SA*[*i*].

The pseudocode of the algorithm *locate* is given as follows.

```
locate(P, ans)
Input: P
Output: ans
1  count(P, L, R)
2  ans[0.. R− L] ←0
3  for i ← L to R do
4          ans[i – L] ← getpos(i)
5  return ans
getpos(i)
1  step ← 0
```

```
2 **while** ($i$ mod $c$) $\neq 0$ **do**
3         $step \leftarrow step + 1$
4         $i \leftarrow \Phi[i]$
5 $i \leftarrow i/c$
6 **return** ($SA_l[i] - step$) mod $n$
```

Starting from a certain point $i \in [L, R]$, *getpos* determines $SA[i]$ by using function Φ to walk along indices $i'(\Phi[i])$, $i''(\Phi[i'])$,…, such that $SA[i]+1 = SA[i']$, $SA[i']+1 = SA[i'']$, and so on, until it reaches an index stored in sampled SA_l. Let *step* be the number of steps in the walk, we return $SA_l[i] - step$, as this is the value of $SA[i]$. The maximum length of each walk is at most $O(\log n)$. The access to $\Phi[i]$ is in $O(\log n)$ time using the precomputed table, as the block size $b = O(\log^2 n)$ and the number of times to access the precomputed table is $O(\log n)$. Thus *locate* runs in $O(occ\log^2 n)$ time.

Theorem 3. Given the locating interval, the locating query of a pattern of length m can be answered in $O(occ\log^2 n)$, at the extra cost of n bits of space for SA_l, except for the space required by representing Φ given in Theorem 1, where occ is the number of occurrences of P in T.

We continue to have pattern bga as an example to see how *locate* works. As we have seen, *locate* first calls *count* to return $[L, R] = [7, 8]$ that contains all the occurrences of bga in T shown in Figure 1 and then invokes *getpos* for $i = 7, 8$ to obtain $SA[7]$ and $SA[8]$. We illustrate this in the case of $SA[8]$ for $c = 3$, the sampling step for SA. Initializing $step = 0$, $i = 8$, then $(8 \bmod 3) \neq 0$, updating $i = \Phi[8] = 31$ and $step = 1$, continuing the next iteration, $(31 \bmod 3) \neq 0$, updating $i = \Phi[31] = 3$ and $step = 2$. Finally, loop ends at $(3 \bmod 3) = 0$. At the time, $i = 3/3 = 1$ and return $SA_l[i] - 2 = 34 - 2 = 32$ as the value of $SA[8]$.

6. Results and Discussion

6.1 Experimental setup and environment

We used g++ 4.4.1 with the -O3 option to build the executables of all the source code in our experiments without parallelism. The experiments were conducted on an HP Z400 with a 2.53 GHz dual-core Intel Xeon W3503. The machine runs 32-bit Ubuntu12.04 LTS OS and has 4 GB internal memory. We used the data from the Pizza&Chili Corpus (http://pizzachili.dcc.uchile.cl/indexes.html) and the Canterbury Corpus (http://corpus.canterbury.ac.nz/) to test the efficiency and usability of our method.

We fully implemented our algorithm in C++. We implemented all the parts of the algorithm except the suffix array construction. We use Veli Mäkinen and R. González's C code(SAu.tgz)[21] to build suffix arrays. Our implementation is generic and ready for public use; it is available at https://sites.google.com/site/compressedsa/.

The compression ratio is defined as the ratio of the size of the CSA structures to the size of original text; bps, that stands for bits per symbol, is obtained by multiplying the compression ratio by 8. The space of our CSA also includes what is necessary for counting, locating and extracting, in addition to that required by S, SB, B, and SAM. We omit all experiments related to extracting queries due to lack of space.

6.2 Results for the Canterbury Corpus

In this and the following sections we evaluate the performance of our CSA based on the compression ratio, compression time, and query time. We compare our results with FM-index [4,14] and Sad-CSA [7,8] on the *Canterbury Corpus* and the *Pizza&Chili Corpus*. Table 3 summarizes some general characteristics of selected files from the *Canterbury Corpus* site.

Table 3. General statistics for our indexes files

Text	paper1	news	book1	world192	bible	E.coli
Size	52KB	368KB	752KB	2.35MB	3.85MB	4.42MB
Alphabet size	95	98	82	94	63	4

We set block size b = 128, superblock size a = 18b, and sampling size for suffix arrays c = 32 in the experiments. We compare our algorithm with FM-index and Sad-CSA on compression rate on Canterbury Corpus, shown in Figures 3. Implementations of FM-index and Sad-CSA can be downloaded from http://pizzachili.dcc.uchile.cl/indexes.html.

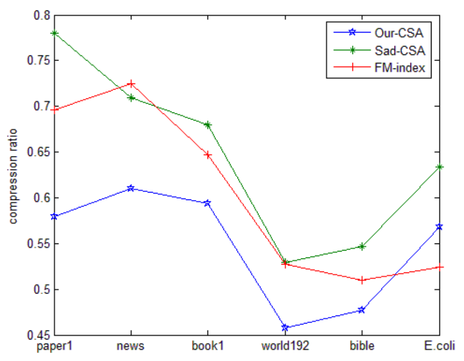

Figure 3: Compression ratio of our CSA compared with FM-index and Sad-CSA

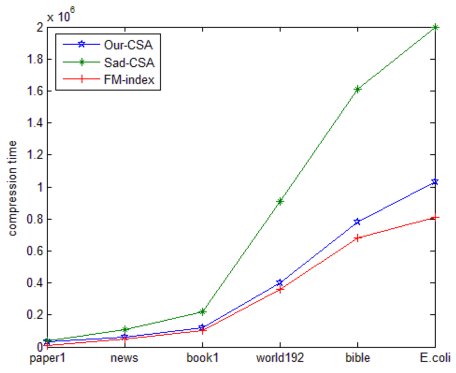

Figure 4: Construction time of our CSA compared with FM-index and Sad-CSA

Table 4. The fraction of gaps with length 1 or 2 for different files

Text	paper1	news	book1	world192	bible	E.coli
Ratio	0.650496	0.641727	0.599657	0.791822	0.738635	0.47226

We can see from Figure 3 that our CSA outperforms Sad-CSA on the compression ratio; it is also better than the FM-index except for E.coli.txt. This is because we use Elias gamma coding to encode the gap sequence, which performs well when the small values are much more frequent than large values in the sequence. For the E.coli data, we can see from Table 4 that the fraction of gaps with lengths 1 or 2 is 0.4722, which is much smaller compared to that of world192 and bible that is in the range from 0.7386 to 0.7918. Figure 4 shows construction times for files from the Canterbury Corpus. We can see that our algorithm runs faster than Sad-CSA and about 25% slower than FM-index.

We searched for 10,000 patterns of length 20, randomly chosen from the indexed text, for a total of 10,000 *count* and *locate* operations. Search time is measured in microseconds. We show the counting and locating query times for the three algorithms in Figures 5 and 6, respectively. Our CSA is the fastest among the three algorithms in answering counting and locating queries, significantly outperforming FM-index in terms of the query time.

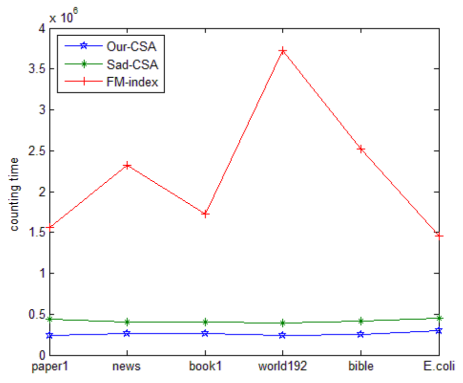

Figure 5: Counting query of our CSA compared with FM-index and Sad-CSA

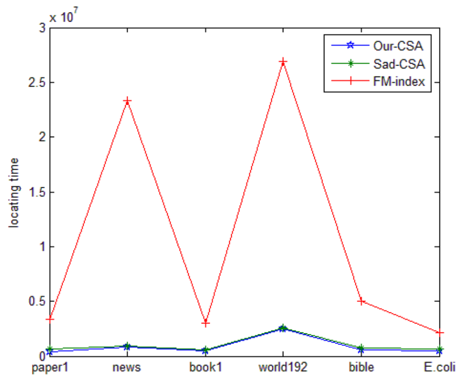

Figure 6: Locating query of our CSA compared with FM-index and Sad-CSA

6.3 Results for the Pizza&Chili Corpus

In this set of experiments, we used two collections of files with different sizes from the *Pizza&Chili Corpus*. One is with size 50M each and the other with size 100M each. We show some statistics on a set of those files with size 50M in Table 5. The data for the set of files with size 100M is the same, except for the English texts, which has the alphabet size 215.

Table 5. General statistics for our indexes files (50M)

Text	DNA	protein	XML	sources	English
Alphabet size	16	25	96	227	176

Figures 7–8 compare our algorithm with FM-index and Sad-CSA with respect to compression rate and construction time on the data from the *Pizza&Chili Corpus*. Our CSA performs better than FM-index and Sad-CSA, except for the DNA and protein data for the former. The results show that the distribution of values of Φ has more effect on the compression ratio than the alphabet size. We can see from Table 6 that the fraction of gaps with length 1 or 2 for the DNA and protein data is much smaller compared to that of XML and sources, in which the gap values distribution concentrates on the small values. They show a better compression ratio.

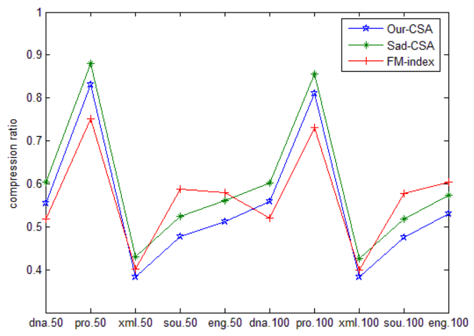

Figure 7: Compression ratio of our CSA compared with FM-index and Sad-CSA

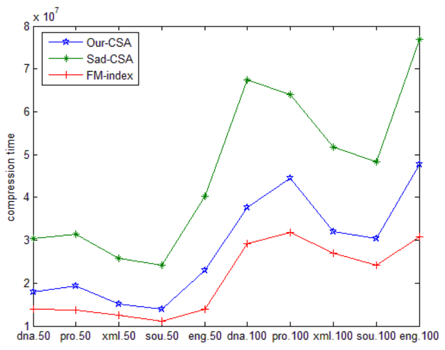

Figure 8: Compression time of our CSA compared with FM-index and Sad-CSA

Consider an extreme case, suppose that there is only one symbol, say c, in text T of length n, then the integers contained in the c-list are in the range $[1..n]$ and form an increasing sequence. The gap sequence only consists of 1's and we use only 1 bit to represent each integer, resulting in a compression ratio 0.125 for our CSA. The aaa.txt from the Canterbury Corpus is consistent with our analysis.

Table 6. The fraction of gaps with length 1 or 2 for different files

Text	dna.50	pro.50	xml.50	sou.50	eng.50	dna.100	pro.100	xml.100	sou.100	eng.100
Ratio	0.5382	0.4047	0.8748	0.7980	0.7224	0.5415	0.4430	0.8782	0.8036	0.7066

It is easy to see from Figure 8 that the construction time of our algorithm is faster than that of Sad-CSA and slower than that of the FM-index.

We show the counting and locating query time for the three algorithms in Figures 9 and 10 respectively. Our CSA is the fastest among the three algorithms on counting queries. Our CSA performs better than the FM-index on locating queries.

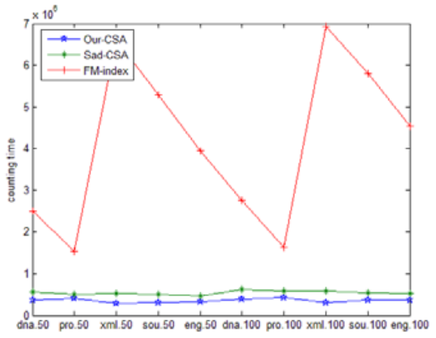

Figure 9: Counting queries of our CSA compared with FM-index and Sad-CSA

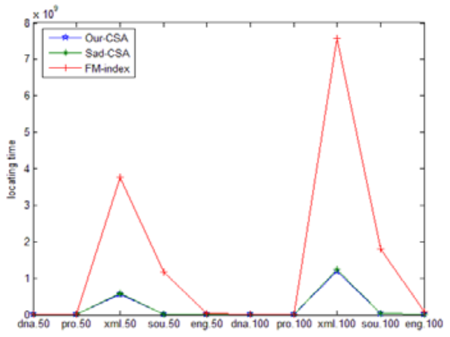

Figure 10: Locating queries of our CSA compared with FM-index and Sad-CSA

Summing up, the distribution of data is more important than the alphabet size for performance of our method. The three algorithms are all suitable for dealing with text and source code data, like English texts, sources, and XML, but our CSA performs best. For more evenly distributed data, such as DNA, the FM-index shows better performance. No matter what type of data is stored in an index, our CSA has a compression rate better than Sad-CSA. For counting and locating queries, our CSA has a significant advantage over the FM-index. For counting queries, our CSA outperforms Sad-CSA. For locating queries, our CSA has a slight advantage over Sad-CSA, except for the DNA and sources.

7. Conclusion

In this paper, we develop a practical representation for representing compressed suffix arrays within $2nH_k + n + o(n)$ bits of space simultaneously for any k, such that $k \leq c \log_\sigma n - 1$ and $c < 1$ is a constant. Experiments on standard corpora show that our method achieves competitive compression ratios. Moreover, we show how our CSA can be built in linear time. We evaluated our storage scheme using compression ratio, compression time, and query time as performance measures. Our results were compared with the FM-index and Sad-CSA on the Canterbury Corpus and the Pizza&Chili Corpus. The experiments indicate that our CSA has a significant advantage over the other two algorithms in terms of compression ratio and times needed to answer counting and locating queries. Our method is competitive on various types of data, except for the evenly distributed data, such as DNA. The reason for this is that, relative to the alphabet size, the distribution of the data itself primarily determines the comparative performance of our algorithms. Our CSA performs well on data with biased distribution.

ACKNOWLEDGMENT. The authors would like to thank Kunihiko Sadakane and Bojian Xu for their quick answer to our questions. This research was supported in part by the National Natural Science Foundation of China (61173025 and 61373044) and the Fundamental Research Funds for the Central Universities (K5051303032). Hongwei Huo and Jeffrey S. Vitter are the corresponding authors.

References

[1] U. Manber and G. Myers, “Suffix arrays: a new method for on-line search,” *SIAM Journal on Computing*, vol. 22, pp. 935–948, 1993.

[2] R. Grossi and J. S. Vitter, “Compressed suffix arrays and suffix trees with applications to text indexing and string matching,” *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378–407, 2005.

[3] R. Grossi, A. Gupta, and J. S. Vitter, “High-order entropy-compressed text indexes,” In *SODA*, pages 841–850, 2003.

[4] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *FOCS*, pages 390–398, 2000.

[5] G. Jacobson, “Space-efficient static trees and graphs,” in *FOCS*, pages 549–554, 1989.

[6] Veli Mäkinen, Gonzalo Navarro, and Kunihiko Sadakane, “Advantages of Backward Searching — Efficient Secondary Memory and Distributed Implementation of Compressed Suffix Arrays,” *Algorithms and Computation Lecture Notes in Computer Science*, vol.3341, pp. 681–692, 2005.

[7] Kunihiko Sadakane, “Compressed text databases with efficient query algorithms based on the compressed suffix array,” In *ISAAC*, Lecture Notes in Computer Science, vol. 1969pages 410–421, 2000.

[8] Kunihiko Sadakane, “New text indexing functionalities of the compressed suffix arrays, ” *J. Alg.* vol.48, no. 2, pp. 294–313, 2003.

[9] E. M. McCreight, “A space-economical suffix tree construction algorithm,” *J. ACM*, vol. 23, pp. 262–272, 1976.

[10] E. Ukkonen, “On-line construction of suffix trees,” *Algorithmica*, vol.14, pp. 249–260, 1995.

[11] R. Grossi and J. S. Vitter, “Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract),” in *STOC*, pages 397–406, 2000.

[12] S. Srinivasa Rao, “Time-space trade-offs for compressed suffix arrays,” *Information Processing Letters*, vol. 82, Issue 6, pp. 307-311, 2002.

[13] P. Ferragina and G. Manzini, “An experimental study of an opportunistic index,” In *SODA*, pages 269–278, 2001.

[14] P. Ferragina and G. Manzini, “Indexing compressed texts,” *Journal of the ACM*, vol.52, no. 4, pp.552–581, 2005.

[15] M. Burrows, D.J. Wheeler, “A block sorting lossless data compression algorithm,” *Tech. Rep.* 124, Digital Equipment Corporation,1994.

[16] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter, “When indexing equals compression: Experiments on suffix arrays and trees,” *ACM Transactions on Algorithms*, vol. 2, no.4, pp.611–639, 2006.
[17] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini, “Compressed text indexes: From theory to practice,” *Journal of Experimental Algorithmics*, vol. 13, no. 12, 1.12, 2008.
[18] Gonzalo Navarro and Veli Mäkinen, “Compressed full-text indexes,” *ACM Computing Surveys*, vol. 39, Issue 1, Article No. 2, 2007.
[19] Rajeev Raman, Venkatesh Raman, S. Srinivasa Rao, “Succinct indexable dictionaries with applications to encoding k-ary trees and multisets,” In *SODA*, pages 233–242, 2002.
[20] L. Foschini, R. Grossi, A. Gupta, and J. S. Vitter. “Fast Compression with a Static Model in High-Order Entropy,” In *DCC*, pages 62–71, 2004.
[21] http://pizzachili.dcc.uchile.cl/indexes/Suffix_Array/