World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:11, No:2, 2017

# Compressed Suffix Arrays to Self-Indexes Based on Partitioned Elias-Fano

Guo Wenyu, Qu Youli



***Abstract*—**A practical and simple self-indexing data structure, Partitioned Elias-Fano (PEF) - Compressed Suffix Arrays (CSA), is built in linear time for the CSA based on PEF indexes. Moreover, the PEF-CSA is compared with two classical compressed indexing methods, Ferragina and Manzini implementation (FMI) and Sad-CSA on different type and size files in Pizza & Chili. The PEF-CSA performs better on the existing data in terms of the compression ratio, count, and locates time except for the evenly distributed data such as proteins data. The observations of the experiments are that the distribution of the φ is more important than the alphabet size on the compression ratio. Unevenly distributed data φ makes better compression effect, and the larger the size of the hit counts, the longer the count and locate time.

***Keywords*—**Compressed suffix array, self-indexing, partitioned Elias-Fano, PEF-CSA.

## I. Introduction

As the number of digitally available information grows at an exponential rate, text indexing becomes more important. Suffix arrays [1], [2] and suffix trees [3] are powerful data structures with numerous applications in such areas as computational biology. Both of them enable to retrieve sequences of a text in almost-optimal or optimal time and occupy O(nlgn)bits. Actually, these text indexing schemes are greedy with reference to space usage. When the alphabet set Σ is of constant size, the indexes are larger than the original text by a multiplicative factor of $\Omega(\log|\Sigma|)$.

The data structures CSA [4]-[7] and FMI [8]-[10] reduce the size of the space, which takes advantage of the index regularities and the text compressibility, and also, support all the application of the suffix trees and suffix arrays. Grossi and Vitter proposed the GV-CSA [4], [5] which overcomes the space limitation from nlgn bits to $O(n\lg|A|)$ bits and answers string matching queries from constant time to $O(\frac{p}{\log_\sigma n} + occ\log_\sigma n)$ time, where the length of the string pattern P is p, and occ is defined as the times of P occur in the original text T. Sadakane made some changes of the original CSA [6], [7] (Sad-CSA), and self-indexing is proposed. For any $h \le \alpha\lg_{|A|}n$, the size has reduced to $nH_h + o(n)$bits with $0 < \alpha < 1$, it can search for a string pattern of length p in $O(p\log n + occ\log^{\epsilon}n)$time, for $0 < \epsilon \le 1$. Ferragina and Manzini [8], [9] designed FMI based on Burrows-Wheeler transform (BWT) [10], a kind of CSA of size s at most $5nH_k(T) + O(n\log\sigma)$bits for $k \le \log_\sigma\left(\frac{n}{\log n}\right) - \omega(1)$, $H_k$ is the order-k entropy of T, and it can search for a string pattern of length p in $O(p + \lg^{1+\epsilon}n)$ time without T. Another type of CSA was proposed by Ferragina and Manzini [11], which needs $O(nH_k\lg^{\epsilon}n)$bits of space and supports $O(p + occ)$ time query. Grossi and Vitter [5] further reduced the size of self-index to $nH_h + o(n)$bits where $h \le \alpha\lg_{|A|}n$ with $0 < \alpha < 1$ and achieved $O(m\lg|A| + \text{polylog}(n))$ query time. In the view of Ferragina and Manzini [8], huge improvements and considerable results can be applied from the compressed indexes.

In this paper, we develop a linear time construction of PEF-CSA data structure to self-indexes based on PEF indexes, and we measure it by compression ratio, count time, and locate time. Compared with the two algorithms FMI and Sad-CSA on the Pizza & Chili, the PEF-CSA works better than the other two on the data except for the protein data, and performs better on the query time. It turns out that the distribution of the φ is more important than the alphabet size on the compression ratio, imbalanced distribution of data φ makes better compression effect, and the size of the hit counts is a significant factor on count and locate time.

In Section II, the details of the preliminaries are introduced to be the basic of the algorithm. We take the recent PEF indexes approach for our algorithm compressing the array φ mentioned in Section III *A* and give the frequency of character to retrieve the original text in Section III *B*. In Section IV, the PEF-CSA is constructed step by step. The query functions such as count query, locate query, and extract function are described in Section V. The experimental analysis is shown in Section VI to evaluate the PEF-CSA performance compared with FMI and Sad-CSA.

## II. Preliminaries

### A. Suffix Array

A suffix array [1], defined as SA, is simply a permutation of all the suffixes of original text T so that the suffixes are lexicographically sorted.

**Definition 1.** Let T[1,n]=T[1]T[2]…T[n] be a long string of length n on an alphabet Σ of size σ and assume that T[n+1]='$' is a special symbol whose order is assigned to 0. A suffix of text $T_{1,n}$ is a substring of the form $T_{k,n}$, where $1 < k < n$. The suffix array SA[1..n] of T is array of integers k that represent the suffixes $T_{k,n}$ containing a permutation of the interval [1,n].

SA[i] = k means that the suffix $T_{k,n}$ is the i-th smallest among all the suffixes starting at the position k in T.

The research is supported by the Fundamental Research Funds for the Central Universities (2015JBM035).

Guo Wenyu is with the Department of Computer science and technology, School of computer and information technology, Beijing Jiaotong University, Beijing 100044, China (e-mail:guowenyu418@163.com).

Qu Youli (Dr.) is with the Beijing Jiaotong University, Beijing 100044, China (e-mail:ylqu @bjtu.edu.cn).

The pattern P is a string of length m over alphabet Σ, all the suffixes prefixed by p in SA occupy a contiguous range. Thus, the count and locate query of P for the interval [l, r] over SA can be accomplished by the method of two binary searches.

**Definition 2.** Let the suffixes be grouped by the one symbol prefix, named p. So, the group where the suffixes start with the same character p is called p-list.

### *B. Compressed Suffix Array (CSA)*

The CSA of Grossi and Vitter [4], [5] struck a balance between achieving fast query performance and the large storage of SA, which reduces the size of a text of length n from $n \log n$ bits to O(n) bits. CSA is a self-indexing structure, and the size of it is expressed by the order-0 entropy of the original text. GV_CSA used decomposition scheme based on a partial function φ.

**Definition 3.** Given the suffix array SA[1,n], function φ is defined as: $SA[\varphi[i]] = SA[i] + 1$. The especial situation is SA[1]=n, in that case $SA[\varphi[i]] = 1$, it is same as (1)

$$\boldsymbol{\varphi_k}[\mathbf{i}] = \begin{cases} \boldsymbol{i'} \textbf{ such that } \boldsymbol{SA_k}[\boldsymbol{i'}] = \boldsymbol{SA_k}[\mathbf{i}] + \mathbf{1} \textbf{ (if } \boldsymbol{SA_k}[\mathbf{i}] < \boldsymbol{n_k}) \\ \mathbf{1} \textbf{ ( if } \boldsymbol{SA_k}[\boldsymbol{i}] = \boldsymbol{n_k}) \end{cases} \quad (1)$$

The biggest difficulty in GV_CSA data structure is the representation and storage of φ, so a new practical method based on PEF indexes comes out because some properties of φ make it appeal to compression. The observation that we will propose is φ is monotonically increasing in the suffix array SA that corresponds to suffixes starting with the same character.

**Lemma 1.** Given the original text T that points to the suffix array SA[1,n]. When $T_{SA[i]} = T_{SA[i+1]}$, it gives the corresponding function $\varphi[i] < \varphi[i+1]$, for $1 \le i < n$.

We assume that $T_{SA[i],n} = xM$ and $T_{SA[i+1],n} = xN$, the 1-symobol prefix of the two suffixes is the character x, thus xM < xN, which means that the position of the suffix M is in front of suffix N in SA, and then M<N. So that, $T_{SA[i]+1,n} = T_{SA[\varphi[i]],n} = M$, in the same way $T_{SA[i+1]+1,n} = T_{SA[\varphi[i+1]],n} = N$. Obviously, $T_{SA[\varphi[i]],n} < T_{SA[\varphi[i+1]],n}$,so $\varphi[i] < \varphi[i+1]$ has been proved.

In Table I, the text T="alabar_a_la_alabarda$ " is a string of length n=21 on an alphabetΣ={$,_,a,b,d,l,r}of size $\sigma = 7$, every SA order is classified by the first character in Σ, which is an increasing sequence. For example, all the suffixes start with a character named a-list with ranks 5-13, whose rankings form a monotonically increasing sequence of positions; namely, 1, 3, 4, 14, 15, 18, 19, 20, 21.

### *C. Partitioned Elias-Fano (PEF) Indexes*

The Elias-Fano representation of monotone sequences is a simple and elegant data structure which has been recently applied into the compression of inverted indexes. Elias-Fano data structure has the excellent characteristics that support fast search operations and random access. While the space occupancy of Elias-Fano is competitive with frequently-used methods such as PForDelta and $\gamma - \delta -$ Golomb codes, it fails to perfectly exploit the local clustering that inverted lists usually exhibit, namely the presence of long subsequences of close identifiers. Ottaviano and Venturini [12] tackle the problem describing a new presentation based on partitioning the monotone sequences into contiguous chunks and encoding both the chunks with different ways. The two-level data structure as shown in Fig. 1 is given to improve compression and support fast queries on the original text. The first level gives the Elias-Fano description of the whole sequence based on juxtaposing the endpoint of every chunk of it. The second level is the specific collection of the chunks represented by three different methods.

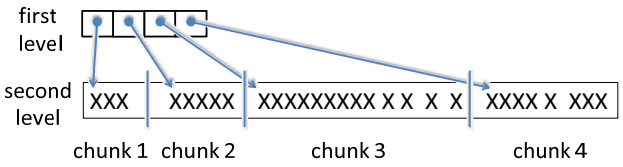


Fig. 1 Two -level of Elias-Fano

**Definition 4.** Consider the monotonically increasing sequences S[0,m-1], for any $0 \le i < m-1$, $S[i] \le S[i+1]$,and S[i] is a non-negative integer from an set [u]={0,1,…,u-1}. The partition P of x chunks is $S[i_0, i_1 - 1]S[i_1, i_2 - 1] \dots S[i_{x-1}, i_x]$, for $i_0 = 0$ and $i_x = m - 1$. The space occupancy of it is defined as $C(P) = \sum_{h=0}^{x-1} C(S[i_x, i_{x+1} - 1])$bits, where C(S[i, j]) represents the cost of S[i,j].

The optimal partitioning aims at decreasing the space occupancy by partitioning the chunks freely with the variable size, the optimal one can be complex in time and space which is not suitable for inputs larger than few thousands of integers. So, they give a presentation of a linear-time algorithm [13] that is a guarantee of at most $(1 + \epsilon)$ times larger than the smallest one, where $\epsilon \in (0,1)$, and then, Ottaviano and Venturini reduced the complexity of time to $O(\log_{1+\epsilon} 1/\epsilon)$ with the two parameters $\epsilon_1$ and $\epsilon_2$.

TABLE I
SUFFIX ARRAY SA AND FUNCTION Φ

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | a | l | a | b | a | r | _ | a | _ | l | a | _ | a | l | a | b | a | r | d | a | $ |
| SA | 21 | 7 | 12 | 9 | 20 | 11 | 8 | 3 | 15 | 1 | 13 | 5 | 17 | 4 | 16 | 19 | 10 | 2 | 14 | 6 | 18 |
| φ | 10 | 7 | 11 | 17 | 1 | 3 | 4 | 14 | 15 | 18 | 19 | 20 | 21 | 12 | 13 | 5 | 6 | 8 | 9 | 2 | 16 |

## III. THE PEF-CSA DATA STRUCTURE

Sadakane [6], [7] gave the representation GV-CSA which can be converted into a data structure of a self-index and meanwhile optimized it in some ways. In this paper, we apply the excellent method named PEF to the function φ combining strong theoretical guarantees and good practical performance.

The resulting index is called PEF-CSA and will be referred to PEF-CSA in this paper.

### *A. The Extensional Function* $\varphi$

The asymptotic space of research on self-indexes [4], [6] is built on the extensional function φ,which maps suffix $T_{A[i],n}$ to suffix $T_{A[i]+1,n}$, so that it can make a scanning from left to right over the original text in forward direction. The same first symbols, namely identical 1-symbol prefix of the suffix arrays, are grouped as one sequence which is monotonically increasing based on the Lemma 1. Grossi and Vitter [4], [5] represent the decomposition scheme by a simple recursion mechanism where the function φ is computed recursively. In our way, only the first level of the data about function φ is reserved, which can completely tell where in the suffix array lies the pointer following the current one with the space as small as possible such that, based on definition 2, given the position i in SA, if $SA[i] = k$, we can find the mapping position j, $SA[j] = k + 1$.

The extensional function φ is stored based on the PEF method, so that the compression of it is completed by the two-level optimal partitioning. In order to facilitate the representation of the discussion, the array of φ and the extensional function φ will be chosen to mention alternatively.

Inspired by the compression of inverted indexes, our idea is to partition the φ array that points to suffixes starting with the same character using PEF. For definition 3 and the description of the property in Section II.C, each chunk cost of the partitioning C(S[i, j]) is defined as two terms: a constant cost named F to store the information about the chunk in the first level, and the space regarding its elements in the second level. Considering the constant cost F, three integers are stored for each chunk, the universe integer in the chunk, the number of the integer in the chunk, and the pointer to the mapping second level. The upper bound of F is defined as the value $2\log u + \log m$ bits. The cost of the specific element of each chunk S[i, j] is computed by a type of self-adoption where the minimum value is chosen from three possible encodings.

Every original element in the chunk subtracts the last element in the previous chunk. By this means, increasing sequence is ensured, while the size of it is minimized in the chunk. Given the size of the universe $u' = S[j] - S[i - 1]$ or $u' = S[j]$ for i=0, the number of elements in this chunk is $m' = j - i + 1$, Vigna [14] used the method that writes the characteristic vector of the set of its elements as a bitvector to represent the sequence with $u'$ bits. So, when the chunk occurs as a dense one, the chunk covers a big fraction of the values in the universe $u'$. In the other words, $m'$ is close to $u'$. If the universe $u' = m'$,which gives the extreme case, the chunk covers the whole universe, the values given in the first level are enough to represent all the elements in the chunk without any further information. In our case, besides Elias-Fano, we use the other two encoding methods based on the relationship between $u'$ and $m'$.The costs of three possible encodings will be introduced as:

1. Elias-Fano Encoding

Vigna [14] gave a detailed description of the representation of the high bits/low bits of a monotone sequence and represented an index using a different architecture based on quasi-succinct representation of monotone sequences. If the chunk is encoded as Elias-Fano, $u'$ is the upper bound of the chunk because of the increasing property. Two-bit arrays are stored to represent the chunk, the upper bits in the upper-bits array are a chunk of unary-coded gaps, the lower $l = \lfloor u'/m' \rfloor$ bits of each S[k],$i < k < j$ are stored in the lower-bits array explicitly and contiguously. It is easily seen that each unary code uses one stop bit. It uses at most $2 + \log\lceil u'/m' \rceil$ bits one element. Indeed, the space bound is $2m' + \log\lceil u'/m' \rceil m'$ bits. The cost of the chunk that is encoded with Elias-Fano is $m'l + m' + u'/2^l$ bits, wherel $= \lfloor \log(u'/m') \rfloor$.

We show an example in Fig. 2. We consider the list 5, 8, 9, 10, 14, 32 with upper bound 32, so $l = \log\lfloor (32/6) \rfloor = 2$. The lower l bits on the right of all elements are concatenated to form the lower-bits array, the lower bits are 01 00 01 10 10 00. The upper bits of the values gap are stored sequentially in the upper-bits array in unary code, and the upper bits are 01 01 1 1 01 000001.

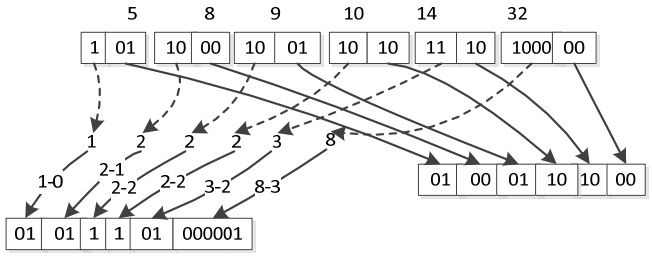


Fig. 2 A simple example of encoding Elias-Fano

2. BitMap Encoding

The chunk is dense when the elements in it cover a large part of the universe where the chunk can be represented with in $u'$ bits whenever $m'$ approaches $u'$. Writing the characteristic vector of the elements is set as a bitvector. The dense chunks are expected to occur frequently in representing the monotone increasing sequence.

The space occupancy of the dense chunk which is encoded as BitMap is $u'$ bits. Within its characteristic vector, the chunk can be stored perfectly.

In Table II, we show an example. Note that we code the dense chunk 1, 2, 3, 5, 7, 9, 10 based on BitMap, the last value in the chunk is $u' = 10$, the number of the dense chunk $m' = 7$. The chunk can be stored in 10 bits, and they are 1110101011.

3. Plain Encoding

The most special case is the densest sequence, which means $m' = u'$, the chunk covers all the elements in the universe $[u']$. Because the values $m'$ and $u'$ stored in the first level are sufficient for themselves to derive all the values in the chunk without the requirement of encoding further information, for example, the sequence is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. The only thing that we need is the value $m' = 10$ and $u' = 10$ in the first

level. We can decode each element in the chunk by the last element in the chunk and the number of the elements in the chunk.

TABLE II
A SIMPLE EXAMPLE OF ENCODING BITMAP

| u′ | 1 | 2 | 3 | | 5 | | 7 | | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| bitmap | 1 | 1 | 1 | 0 | 1 | 0 | 1 | | 1 | 1 |

The cost of the most densest chunk is 0 bits, which means if $m' = u'$, thus S[i,j] covers the whole chunk.

It remains to describe how to give the coding in $O(\log_{1+\epsilon} 1/\epsilon)$ time comparing to the most optimal one. It can be done by giving $k = O(\log_{1+\epsilon} 1/\epsilon)$ windows $w_0, \dots, w_k$ sliding in the whole sequence, these windows cover different fractions of the sequence, which start at the same position but end at different end position. In the beginning, we initialize the start and end position value of all the windows, and each window starts and ends at the 0 position. Every time in the execution, we use the shortest path method to visit the next value in the original sequence. We advance the start position of each sliding window by one position step, and the end position until the most cost of the vertex that it can visit. Every time that we move the position of a window, we should compute the cost of the vertex that represents the portion of the sequence. It can be done in linear time. At the end of the algorithm, every portion in the sequence will be computed, so visiting the value of the last vertex, we can get the smallest cost of the sequence in $O(\log_{1+\epsilon} 1/\epsilon)$ time.

### B. Frequency of Character

Self-index structure is based on the table that represents the frequency of character. Table III is given as the name of the form to map the alphabet symbols lexicographically sorted.

**Definition 5.** Let C[p] be the rank of the smallest suffix in the p-list in the lexicographic order. In the other words, C[p] is the sum number of the alphabet symbol $p'$ where $p' < p$ in the original text.

TABLE III
THE FREQUENCY OF CHARACTER IN T

| symbol | $ | _ | a | b | d | l | r | |
|---|---|---|---|---|---|---|---|---|
| frequency | 0 | 1 | 4 | 13 | 15 | 16 | 19 | 21 |

Table III represents the character frequency of the example. The extra entry n is added to the end of the form for the convenience. We can find that the suffixes $SA[C[p] + 1 \dots C[p + 1]]$ belong to the p-list.

**Lemma 2.** $T_{SA[i],n}$ can be extracted from function φ recursively: $i, \varphi[i], \varphi[\varphi[i]] \dots$ as we point to $T_{SA[i]}, T_{SA[i]+1}, T_{SA[i]+2}, \dots T_{SA[n]}$ after n-i+1 steps, the positions of the $T_{SA[i]}, T_{SA[i]+1}, T_{SA[i]+2}, \dots T_{SA[n]}$ in the lexicographic order corresponding to the symbols in Table III can be revealed. The first symbol $T_{SA[i]}$ of the suffix $T_{SA[i],n}$, in alphabetic order of SA must be the symbol p such that $C[p] < i < C[p + 1]$.

With the extensional function φ and the form C, we can reveal the suffix $T_{SA[i],n}$ corresponding to SA[i]. So, the original text can be discarded.

## IV. PEF-CSA CONSTRUCTION

We build the data structure PEF-CSA in linear time in the following steps.

Step 1. Constructing SA and form C.
Step 2. Computing value φ using C, SA, T, abandoning T after the computation.
Step 3. Sampling SA and $SA^{-1}$, abandoning SA after the sampling.
Step 4. Encoding φ using PEF, abandoning original φ after that.

The first step will not be explained because of the standard algorithm written by Manzini and Ferragina [8], [11]. We only need to explain the three last steps work.

### A. Computing Extensional Function φ

```
Algorithm ComputePhi(C, SA, T, φ)
Input: C, SA, T
Output: φ
1 end ← C[endchar]
2 for k ←1 to n do
3 temp ← SA[k]
4 if temp=1 then endpos=i
5 else
6 p ← T[temp-1]
7 φ[C[p]] ← i
8 C[p] ← C[p]+1
9 φ[end] ← endpos
```

In the pseudocode, the array C is assumed as a local value and the entries are all reset for the ComputePhi. It is obvious that the function ComputePhi runs in O(n) time. The first line above represents the inverse of suffix array that equals the last symbol in the original text, SA[end]=n, where endchar is the last symbol. The function φ is based on the following point. Assume that suffix array SA[i] =j. if p = T[j-1], C[p] gives the present number of the p-list. So $\varphi[C[p]] = i$.

### B. Sampling SA and $SA^{-1}$

$SA^{-1}$ is the inverse of permutation of SA and $SA_s$, $SA_s^{-1}$ denote the sampled SA and $SA^{-1}$. With the sampling of SA and $SA^{-1}$, $SA_s$ and $SA_s^{-1}$ are built, respectively. The step of sampling for $SA_s$ and $SA_s^{-1}$ are st and nst. The pseudocode of sampling SA and $SA^{-1}$ method is written as follows.

```
Algorithm SampleCSA (SA, st, nst, SA_s, SA_s^-1)
Input: SA, st, nst
Output: SA_s, SA_s^-1
1 sacount ← ⌈(n/st)⌉
2 for i ←1 to sacount do
3 SA_s[i] ← SA[st*i]
4 for j ←1 to n do
5 if(SA[j] mod nst =0) then SA_s^-1[SA[i]/ nst] ← j
```

$SA_s$ is built in lines 2-3 for st sampling length by reducing the suffix array, so it gives the entries SA[st*i] where the result is a multiple of st. SA is determined by $SA_s$ if we want to have a query described in the Section V *C*.

TABLE IV
$SA_s$ AND $SA_s^{-1}$ WITH ST=NST=4

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T | a | l | a | b | a | r | _ | a | _ | l | a | _ | a | l | a | b | a | r | d | a | $ |
| SA | 21 | 7 | 12 | 9 | 20 | 11 | 8 | 3 | 15 | 1 | 13 | 5 | 17 | 4 | 16 | 19 | 10 | 2 | 14 | 6 | 18 |
| $SA_s$ | 9 | 3 | 5 | 19 | 6 | | | | | | | | | | | | | | | | |
| $SA_s^{-1}$ | 14 | 7 | 3 | 15 | 5 | | | | | | | | | | | | | | | | |

$SA_s^{-1}$ is built in lines 4-6 for nst sampling length by reducing the inverse of suffix array. Based on Section IV.B, we find that it is easy to retrieve the suffix array when suffix ranking is given. When we want to pick up the string that start position is start and the length is len, the function is named extract(start,len), the problem turning the position start into ranking i must be resolved. The solution to transform the start to ranking i is as follows. First, we give the $SA^{-1}$ a sampling step nst and it means that we collect all entries SA[j] mod nst =0 in which SA[j] is exactly divisible by nst, and then give $SA_s^{-1}$ the value j. Given the ranking i of position start, we can restore the string $T_{start,start+len+1}$ according to lemma 2 in Section III.B. Obviously, we find the algorithm SampleCSA can be completed in O(n) time. Table IV shows the $SA_s$ and $SA_s^{-1}$.

### *C. Encoding Extensional Function φ with PEF*

Function coding Phi describes the method that compresses φ in $O(\sigma n \log_{1+\epsilon} 1+\epsilon)$ time shown in Section II.C. Assume that the $\epsilon_1$ and $\epsilon_2$ have been defined as 0.03 and 0.3, that balance the construction time and the space occupancy.

```
Algorithm codingPhi (φ, C,bvb)
Input: φ, C
Output: bvb
1 phicount ← 0
2 for i ←1 to C.length() do
3 phinum ← C[i+1]-C[i]
4 universe ← 0
5 for j ←1 to C[i] do
6 begin.addphi(φ[phicount])
7 phicount++
8 universe = φ[phicount]
9 optimal_partition(opt, begin, universe, phinum, esp1,esp2)
10 it ← begin
11 cur_base = *begin
12 for p ←0 to opt.partition.size do
13 for cur_i ←0 to opt.partition[p] ++it do
14 value = *it
15 cur_partition.pushback(value – cur_base)
16 base_partition_write(bvb, cur_partition.begin(),
       cur_partition.back()+1, cur_partition.size())
17 cur_base = value + 1
```

Line 1 keeps the ranking of the array φ in phicount, i.e. phicount =0. phinum gives the number of the increasing sequence that is the subsection of φ according to the same p-list, phinum = C[i+1]-C[i]. Lines 5-7 initialize the value which keeps the information of φ in the p-list. After that, every increasing sequence will be partitioned optimally based on the function optimal_partition, the result of partitioning is stored in the value opt. Lines 13-15 make values in the chunk minus the last value in the former chunk and store the last value in the present chunk for the following computing. We write the final result bvb in the method base_partition_write in Line 16.

The pseudocode of the algorithm optimal_partition describes how to generate the partitions in $O(n \log_{1+\epsilon} 1/\epsilon)$ time mentioned in Section IV.A. In the other words, we find the optimal partition for the increasing sequence in the linear time based on the following pseudocode.

```
Algorithm optimal_partition(begin, universe, size, esp1, esp2, opt)
Input: begin, universe, size, esp1, esp2
Output: opt
1 singleblock_cost ← cost_base(universe, size)
2 costmin (size+1, singleblock_cost,mincost[])
3 cost_lb ← cost_base(1,1)
4 cost_bound ← cost_lb
5 while esp1=1 or cost_bound < cost_lb /esp1 do
6 windows.emplace_back(begin, cost_bound)
7 if (cost_bound >= single_block_cost) break
8 cost_bound ← costbound * (1+esp2)
9 for i ←0 to size do
10 last_end ← i + 1
11 for window: windows do
12 while window.end < last_end do
13 window.advance_end();
14 while true do
15 window_cost ← cost_base(window.universe,window.size)
16 if opt.mincost[i] + window_cost < opt.min_cost[window.end] then
17 opt.min_cost[window.end] ← opt.min_cost[i] + window_cost
18 opt.path[window.end] ← i
19 last_end ← window.end
20 if window.end = size break
21 if window_cost >= window.cost_upper_bound break
22 window.advance_end()
23 window.advance_start()
24 curr_pos ← size
25 while curr_pos != 0 do
26 opt.partition.pushback(curr_pos)
27 opt.curr_pos ← opt. path[curr_pos]
28 opt.cost_opt ← opt. min_cost[size]
```

Lines 5-8 build k sliding windows as $k = O(n \log_{1+\epsilon} 1/\epsilon)$ and give the ending position of every window the upper bound. Armed with these windows, every time that the algorithm visits the next vertex, we advance the start position as mentioned in line 23. In lines 15-22, when we move the start or the end position of the windows, we need to evaluate the cost of the current portion of the sequence. In line 28, cost_opt is the optimal cost of the array φ in the same list.

## V. INDEXING FUNCTIONALITIES OF PEF-CSA

Given the array C shown in Section III.B, we used the sampling suffix array $SA_s$ and inverse suffix array $SA_s^{-1}$ to support two pattern matching queries for self-index: locate

function, count function, and we also accomplished the extract function mentioned in Section III.B for the PEF-CSA.

### *A. Count Function*

Count function defined as count based on the backward search gets rid of the normal framework that is a sequential scan. It gives the occurrences of pattern P in the original text T. From the whole count procedure, we use φ to reduce the scope between L and R that report the positions of P in T. When the function φ iterates to the first character, all the suffix arrays in SA[L,R] contain the prefix P. It returns L and R, R-L+1 is the count of the P in T.

```
Algorithm count(P, L, R)
Input: P
Output: L, R
1 p ← P[m]
2 L ← C[p]+1
3 R ← C[p+1]
4 for i ←m-1 to 1 do
5 templ ← C[c]+1
6 tempr ← C[c+1]
7 p ← P[i]
8 cl ← min{xl,xl ∈ {templ, tempr}, φ[xl] ∈ {L, R}}
9 cr ← max{xr,xr ∈ {templ, tempr}, φ[xr] ∈ {L, R}}
10 L ← cl
11 R ← cr
12 if L>R then return -1
13 return R-L+1
```

Notice that m is the length of the pattern P, lines 8-9 mean to determine the new boundary of the final position. The algorithm based on the actual characteristic of the function φ begins with the last character of the pattern, and ends with the first character of the loop. Obviously, this algorithm in the implementation process to maintain the following invariants: when the algorithm is executed from the kth character, the suffixes in the range of [L, R] have the prefix that is the last k characters in P.

In lines 1-3, we give the initialization of L, R and character p, L corresponds to the first symbol of p-list and R maps the last element of p-list, so the interval [L, R] is the range of p-list. Lines 8-9 describe the list of backward character for P, cl represents the start position, and rl gives the end position of the list. When cl>cr just like line 12, the algorithm returns -1, if the original text T has the pattern P, it will return R-L+1.

We use pattern P=“ala”as an example to give the count process.

After we initialize the values in lines 1-3, $L = C[a] + 1 = 5$, $R = C[a + 1] = C[b] = 13$. The character ‘a’ is the start of suffixes in [5, 13], corresponding to a-list. When we first iterate L and R in the loop, templ=C[l] +1=17, tempr=C[l+1]=19. The suffixes in [17, 19] start with ‘l’ and map with the φ values are $\{6, 8, 9\}$ where $\varphi[17] = 6$ and $\varphi[18] = 8, \varphi[9] = 9$ in [5, 13]. Thus [cl, cr] = [17, 19]. In line10-11, the [L, R] = [17, 19] has been updated. Therefore, the suffixes in this range are prefixed with “la”. When it starts the second iteration, templ=C[a]+1=5, tempr=C[a+1]=13, the corresponding values are $\{1,3,4,14,15,18,19,20,21\}$ for which $\varphi[10] = 18$ and $\varphi[11] = 19$ in [17,19]. So, [cl, cr] = [10, 11] and [L, R] = [10, 11], so there are R-L+1=2 “ala” in T.

### *B. Locate Function*

To locate the counting positions, we describe the locate function to give the specific values of the interval [L, R]. Suppose that we already get the counting interval [L, R], the algorithm finds the corresponding original position SA[L,..,R].

The pseudocode of the locate algorithm is given below.

```
Algorithm locate(P, L, R,pos)
Input: P , L, R
Output: pos
1 Initialize L,R to be the result of count query return
2 Initialize pos[1,…,R-L+1]to be 0
3 R ← C[p+1]
4 for i ←L to R do
5 tmstep ← 0
6 while i mod st != 0 do
7 tmstep ← tmstep ++
8 i ← φ[i]
9 i ← i/st
10 pos[i-L] ←SA_s[i] - tmstep
11 return pos
```

After we get the count query interval [L, R], suppose that the certain value in the interval is named i, we can determine the SA[i] through the function φ. We walk along φ to reach the index that is stored in $SA_s$. Let tmstep be the number of steps in the walk, we return $SA_s[i]$ – tmstep.

We continue to have pattern P=”ala” as an example to know how the function locate works. As we know, we get the occurrence interval [L,R] from the count query which contains the ranking of the position. So that what we have to do is to get SA[L]..SA[R] as L=10 and R=11. We give the case computing SA[10] as st=4, and st is the sampling step for SA mentioned in 4.2. If we want to get SA[10], i =10, then $10 \bmod 4! = 0$, so we illustrate $i = \varphi[10] = 18$ and $tmpstep = 1$, continuing the loop’s iteration, then 18mod 4!=0, $i = \varphi[18] = 8$, so the loop ends at $i = 18$. Finally, the algorithm returns $SA_s[i] - 1$=$SA_s[18] - 1 = 2 - 1 = 1$, so SA[10]=1.

### *C. Extract Function*

In Section IV.B, we get the arrays $SA_s$ and $SA_s^{-1}$ which support the function extract (start, len) that returns the string $T_{start,start+len+1}$. From lemma 2 in Section III.B, how to restore the suffix array is described with φ and array C.

```
So, the Algorithm extract(start, len,str)
Input: start, len
Output: str
1 i ← SA_s^{-1}[⌊start/nst⌋]
2 tmpstep ← start mod nst
3 for j←1 to tmpstep do
4 i ← φ[i]
5 for j←0 to len-1 do
6 str[j] ← inverseC(i)
7 i ← φ[i]
8 return str
```

Most important thing to restore the string is to convert the starting position start to the corresponding rank i. We update $i = \varphi[i]$, and repeat the above process, then the suffix can be determined.

nst is the sampling step of $SA_s^{-1}$. The algorithm first finds the most sampling points $\lfloor start/nst \rfloor$ smaller than start in line 1, then the point ranking i is derived. After that, in lines 3-4, we continue to operate $i = \varphi[i]$ for start mod nst time. Now, the current i is the final one mapping the start position. In lines 5-7, we get the string according to the array C, and the inverse C gives the corresponding character of position i.

In the case of Section IV *B*, extract(5,4) runs as following. To determine the suffix rank i of position start = 5, we suppose that nst= 4, the maximum sampling step less than 5 is 4, corresponding to the 11th of the $SA_s^{-1}$, i=14, that means the rank of the suffix which starts at position 4 is 14, 5 mod 4 = 1, we continue the iteration $i = \varphi[i]$, $i=\varphi[1]=12$, 12 mod 4 = 0, so we get the suffix rank 12 of position 5. We know len=4, in lines 5-7, when $i = 12$, it is in the a-list, so the first character is 'a'; $i = \varphi[12] = 20$ is in the r-list, the second is 'r'; $i = \varphi[20] = 2$ is in the r-list, the second is '_'; $i = \varphi[2] = 7$ is in the r-list, the second is 'a'. So, the string $T_{5,8} = "ar\_a"$.

## VI. Experimental Analysis

### A. Experimental Setup and Environment

All the algorithms were fully implemented in C++ and compiled with G++ 4.8.4 with the highest optimization setting. The experiments ran on the machine with a 3.1 GHz Intel(R) Core(TM) i5-3450 CPU. The machine runs 64-bit Ubuntu14.04 LTS OS and has 16 GB internal memory.

Except the fact that the algorithm construction of suffix array is based on the C code of Mäkinen and González (SAu.tgz) [15], all the parts of the algorithms have been implemented. We used the dataset from Pizza&Chili that has different type or size to test the efficiency of our algorithms. We gave the compression ratio defined to the ratio of the space occupancy of the PEF-CSA structure to the size of the text, and we test the locate and count query time of the PEF-CSA algorithm.

### B. Experimental Result Analysis

In this section, we measure the performance of PEF-CSA by the compression ratio, count time, and locate time. We use five different types of original text to be the dataset from the Pizza&Chili that are dna data, english data, proteins data, sources data, and xml data. The files can be classified by the size of them; 50 MB and 100 MB. The results are compared with Sad-CSA and FMI.

The PEF-CSA compression ratio compared with FMI and Sad-CSA is shown in Fig. 3. It reflects that the smallest result of the ratio is 0.29, which means the index can be approximately compressed to the 1/4 of the original text. PEF-CSA is better than SAD-CSA on compression ratio, and performs better than FMI except for the DNA and proteins data.

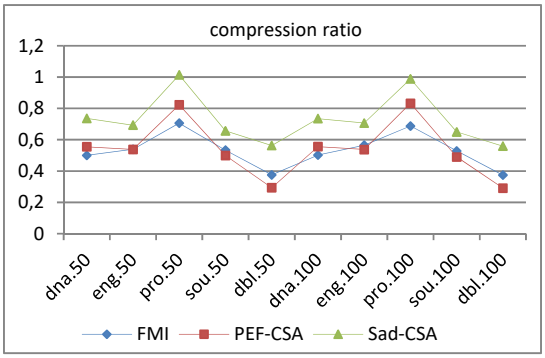


Fig. 3 Compression ratio of PEF-CSA with FMI and Sad-CSA

The best compression performing data are xml data and source data, and the crucial feature of both data is highly structured, that means the distribution of data φ is uneven. From Table VI, we can know that the φ distribution of the original text plays a greater role than the alphabet size of original text, which means uneven data φ performs better than uneven one.

We randomly choose 10000 patterns of length 20 from each original text with the genpatten program in Pizza&Chili, and then, we can generate 10 pattern string files.

TABLE V
The Occurrences of the Pattern Strings in the Original Texts

| pattern file | hit counts |
|---|---|
| dna.pattern.50MB | 220410 |
| eng.pattern.50MB | 308563 |
| pro.pattern.50MB | 321003 |
| sou.pattern.50MB | 18057976 |
| dbpl.pattern.50MB | 84431027 |
| dna.pattern.100MB | 378812 |
| eng.pattern.100MB | 1942363 |
| pro.pattern.100MB | 482312 |
| sou.pattern.100MB | 36764894 |
| dbpl.pattern.100MB | 150904999 |

Table V gives the occurrences of the pattern string stored in the pattern string files from the original texts and we called hit counts. The two pattern string files sou.pattern file and dbpl.pattern file have far more hit counts than other files.

We searched for the patterns for the count and locate function, and the microsecond is the measure of the search time. From Figs. 4 and 5, we know the PEF-CSA is faster than the Sad-CSA and FMI and performs better than the FMI on locate function. Combined with Table V, although the size of original file can affect the count and locate time, the hit counts of the file are the most important factor to impact the time, especially the locate time.

TABLE VI
The Alphabet Size of the Texts

| Text | dna.50 | eng.50 | pro.50 | sou.50 | dlbl.50 | dna.100 | eng.100 | pro.100 | sou.100 | dblp.100 |
|---|---|---|---|---|---|---|---|---|---|---|
| size | 16 | 176 | 25 | 227 | 96 | 16 | 215 | 25 | 227 | 96 |

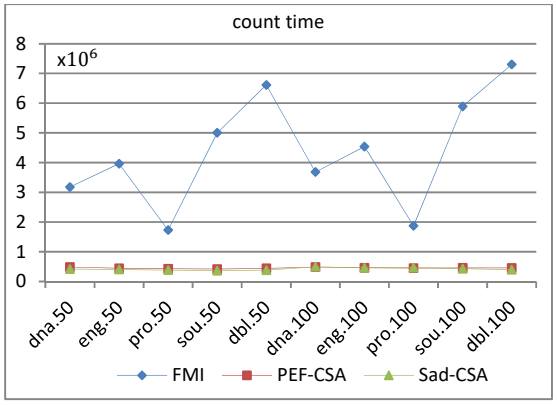


Fig. 4 Count time

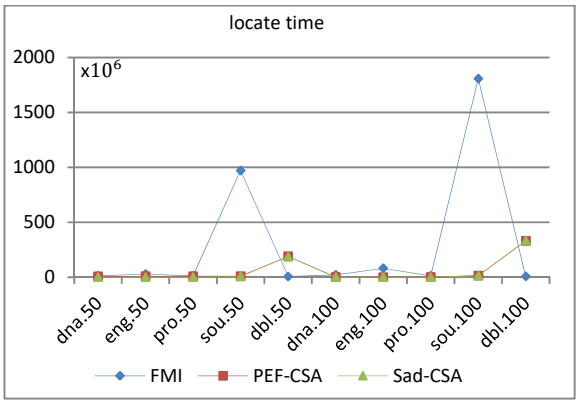


Fig. 5 Locate time

To sum up, the three algorithms are all suitable for self-compressing the files such as sources data, xml data, and DNA data, but PEF-CSA can perform better in the compression ratio and query time. The distribution of the φ is more important than the alphabet size on the compression ratio, and the size of the hit counts is a significant factor on count and locate time.

## VII. Conclusion

In our paper, we give a simple storage data structure for the PEF-CSA. The PEF-CSA can be developed in linear time. The experiments on the Pizza&Chili are accomplished comparing PEF-CSA with the two established standard methods FMI and Sad-CSA for the datasets of different types and size on the compression ratio, count, and locate time. Moreover, we present how PEF-CSA works in linear time where the sliding windows choosing scheme has been given. Taken together, PEF-CSA is a competitive data method on the uneven distributed data like xml and sources data. The distribution of the φ is more significant than the alphabet size on the compression ratio, and the size of the hit counts is an important factor on count and locate time. The bigger of the hit counts, the longer of the query time.

## References

[1] Manber U, Myers G. Suffix arrays: a new method for on-line string searches (J). Siam Journal on Computing, 1993, 22(5):935-948.

[2] Ukkonen E. On-line construction of suffix trees (J). Algorithmica, 1995, 14(3):249-260.

[3] Mccreight E M. A Space-Economical Suffix Tree Construction Algorithm (J). Journal of the Acm, 1976, 23(2):262-272

[4] Grossi R, Vitter J S. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching (Extended Abstract) (C)// ACM Symposium on Theory of Computing. ACM, 2000:397-406.

[5] Grossi R, Vitter J S. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching (J). Siam Journal on Computing, 2006, 35(2):397--406.

[6] Sadakane K. Compressed Text Databases with Efficient Query Algorithms Based on the Compressed Suffix Array (M)// Algorithms and Computation. Springer Berlin Heidelberg, 2000:410--421.

[7] Sadakane K. New text indexing functionalities of the compressed suffix arrays (J). Journal of Algorithms, 2003, 48(2):294-313.

[8] Ferragina P, Manzini G. Opportunistic data structures with applications (M). University of Pisa, 2000.

[9] Ferragina P, Manzini G. An experimental study of an opportunistic index (C)// Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2001:269-278.

[10] Burrows M. A block-sorting lossless data compression algorithm (J). Technical Report Digital Src Research Report, 1995, 57(4):425.

[11] Ferragina, Paolo, Manzini, Giovanni. On compressing and indexing data (J). Università Di Pisa, 2002.

[12] Ottaviano G, Venturini R. Partitioned Elias-Fano indexes (C)// International ACM SIGIR Conference on Research & Development in Information Retrieval. ACM, 2014:273-282.

[13] Ferragina P, Nitto I, Venturini R. On Optimally Partitioning a Text to Improve Its Compression (J). Algorithmica, 2011, 61(1):51-74.

[14] Vigna S. Quasi-succinct indices (J). 2012:83-92.

[15] Navarro G, Mäkinen V. Compressed full-text indexes (J). Acm Computing Surveys, 2007, 39(1):2.

**Guo Wenyu** received B . E . from the Lanzhou Jiao Tong University. Currently, he is working toward the M.S. degree at the Beijing Jiaotong University. His research interests include information retrieval, data compression.

**Qu Youli** received B.E,M.E and D.E from the Beijing Institute of Technology. During the graduate studies, he studied the big data management and analysis, cloud computing and Information retrieval. He is a Senior Engineer. He is an associate professor and Master's tutor of the School of Beijing Jiaotong University. His research interests include natural language processing, knowledge engineering, and web service and information management.