New Text Indexing Functionalities of the Compressed Suffix Arrays

Kunihiko Sadakane

Department of System Information Sciences
Graduate School of Information Sciences
Tohoku University
Aramaki Aza Aoba09, Aoba-ku, Sendai 980-8579, Japan
E-mail: sada@dais.is.tohoku.ac.jp

New text indexing functionalities of the compressed suffix arrays are proposed. The compressed suffix array proposed by Grossi and Vitter is a space-efficient data structure for text indexing. It occupies only $O(n)$ bits for a text of length n; however it also uses the text itself that occupies $n \log_2 |\mathcal{A}|$ bits for the alphabet $\mathcal{A}$. In this paper we modify the data structure so that pattern matching can be done without any access to the text. In addition to the original functions of the compressed suffix array, we add new operations *search*, *decompress* and *inverse* to the compressed suffix arrays. We show that the new index can find *occ* occurrences of any substring P of the text in $O(|P| \log n + occ \log^{\epsilon} n)$ time for any fixed $1 \geq \epsilon > 0$ without access to the text. The index also can decompress a part of the text of length m in $O(m + \log^{\epsilon} n)$ time. For a text of length n on an alphabet $\mathcal{A}$ such that $|\mathcal{A}| = \mathrm{polylog}(n)$, our new index occupies only $O(nH_0 + n \log\log |\mathcal{A}|)$ bits where $H_0 \leq \log |\mathcal{A}|$ is the order-0 entropy of the text. Especially for $\epsilon = 1$ the size is $nH_0 + O(n \log\log |\mathcal{A}|)$ bits. Therefore the index will be smaller than the text, which means we can perform fast queries from compressed texts.

Key Words: text database, suffix array, compression

1. INTRODUCTION

1.1. Backgrounds

As the number of machine-readable texts grows, text search techniques become more important. Traditional algorithms perform sequential search to find a keyword from a text; however it is not efficient for huge databases. We create indices of the text in advance for querying in sublinear time. In the area of text retrieval, the inverted index [11] is commonly used due to its space requirements and query speed. The inverted index is a kind of *word indices*. It is suitable for English texts, while it is not suitable for Japanese texts or biological sequences because it is difficult to

parse them into words. For such texts a kind of *full-text indices* is used, for example suffix arrays [17] and suffix trees [18]. These indices enable finding any substring of a text quickly. Furthermore, they support more complicated operations than the inverted index, which can be used for text data mining [22]. However, the size of full-text indices are quite larger than that of word indices. Recent researches are focused on reducing the sizes of full-text indices [16, 9, 7, 13].

The compressed suffix array of Grossi and Vitter [9] reduces the size of the suffix array of a text of length n from $n \log n$ bits[1] to $O(n)$ bits. Each element of the suffix array can be extracted in $O(\log^\epsilon n)$ time. Therefore we can find any pattern P in the text in $O((|P| + \log^\epsilon n) \log n)$ time using the text and the compressed suffix array by a straightforward binary search. The compressed suffix array is also used with succinct representation of suffix trees in $O(n)$ bits to find P in $o(|P|)$ time. Though they considered only binary alphabet, it can be generalized for texts with alphabet size $|\mathcal{A}| > 2$.

The compressed suffix array has two drawbacks: One is that this representation uses the text itself to search for patterns. Therefore the total size of the index cannot be smaller than the text, which occupies $n \log |\mathcal{A}|$ bits. The other is that a direct application of the compressed suffix array with binary alphabet to multi-alphabets results $O(n \log |\mathcal{A}|)$ bits indices. Therefore the index is larger than the text size.

1.2. New Results

In this paper we propose new data structures and query algorithms for the compressed suffix array. We call the new data structure *CSA*, an abbreviation for compressed suffix array. It has the following features:

1. It supports pattern queries without using the text.
2. Its size is expressed by the order-0 entropy of the text. The analysis on time and space complexity is valid for even large alphabets ($|\mathcal{A}| = \text{polylog}(n)$).

The CSA solves the two problems in the original compressed suffix array. It is a self-indexing data structure and its size is expressed by the order-0 entropy of the text. Not only used as just a compressed version of the suffix array, it supports three basic operations to the index, *search*, *decompress* and *inverse*, without using the text itself. Since we do not need the original text, the size of the CSA can become smaller than the text size if the text is compressible. To support the operations we add new components of small size to the compressed suffix array. The *search* returns the interval in the suffix array that corresponds to a pattern P from the text of length n in

[1]The base of logarithm is two throughout this paper.

$O(|P| \log n)$ time. The *decompress* extracts a substring of length m of the text from the index in $O(m + \log^{\epsilon} n)$ time where $1 \geq \epsilon > 0$ is a constant determined before index construction. The *inverse* returns the inverse of the suffix array. It has many applications, for example the lexicographic order of a suffix in any part of the text can be efficiently computed by using the inverse of the suffix array. This enables efficient computation of word-association patterns [14, 22], which are sets of keywords which appear in the neighborhood. Though the inverse of a suffix array can be computed easily from the suffix array, it requires additional $n \log n$ bits. On the other hand, the CSA can support it by using additional $n + o(n)$ bits.

The CSA occupies only

$$n\left(\frac{1+\epsilon'}{\epsilon}H_0 + 2\log(1+H_0) + 3\right) + o(n)$$

bits where H_0 is the order-0 entropy of the text, and $1 > \epsilon > 0$ and $\epsilon' > 0$ are arbitrary constants. Note that $|\mathcal{A}| = \text{polylog}(n)$. The time complexity for lookup and inverse is $O(\frac{1}{\epsilon\epsilon'} \log^{\epsilon} n)$. For $\epsilon = 1$, the CSA has size

$$n(H_0 + 2\log(1+H_0) + 3) + o(n)$$

bits and $O(\log n)$ access time.

This is a novel analysis. The compressed suffix array has been originally proposed for a binary alphabet. Therefore if we apply it directly to non-binary alphabets, we first convert each character in a text into $\log |\mathcal{A}|$ consecutive binary characters, then construct the compressed suffix array. Then its size becomes $O(n \log |\mathcal{A}|)$ bits, which is larger than the text size. Although it is obvious to extend the compressed suffix array for non-binary alphabets, the size of the index has not been analyzed yet.

An important merit of the CSA is its index size independency on alphabet size. We relax the condition that the alphabet size is constant. We achieve the above size and time complexities for alphabets of size $\text{polylog}(n)$. We can express the index size as $\frac{1+\epsilon'}{\epsilon} nH_0 + O(n \log\log |\mathcal{A}|)$. Therefore it can be smaller than the text on large alphabets. For example, assume that $n < 2^{32}$ and $H_0 = 3$, which is practical for English texts. If we use $\epsilon = 0.5$, the size of the CSA is approximately $18n$ bits. On the other hand, the text itself and its suffix array occupies $n \log |\mathcal{A}| + n \log n = 8n + 32n = 40n$ bits if $|\mathcal{A}| = 256$. Therefore the CSA reduces the space complexity by 55%. The original compressed suffix array occupies at least $\frac{1}{\epsilon} n \log |\mathcal{A}| + 2n \geq n \log |\mathcal{A}| + 2n = 18n$ bits for the index itself, and $n \log |\mathcal{A}| = 8n$ bits for the text.

1.3. Related Works

The opportunistic data structure of Ferragina and Manzini [7] is the first data structure that solved the two problems of the original compressed suffix array. It is a *self-indexing* data structure, that is, it does not require the text itself for queries. It uses the block sorting [2] to compress the text that has good compression ratio and fast decompressing speed. Its space occupancy is $O(nH_k)+O(\frac{n}{\log n}(|\mathcal{A}|+\log\log n)+n^\epsilon|\mathcal{A}|2^{|\mathcal{A}|\log|\mathcal{A}|}))$ bits where H_k is the order-k entropy of the text. This holds for any integer $k > 0$. The first term is smaller than $n\log|\mathcal{A}|$, the text size. It allows to enumerate any pattern P in a compressed text in $O(|P| + occ\log^{1+\epsilon} n)$ time for any given $1 > \epsilon > 0$. Since $H_k \leq H_0$ always holds, the first term is smaller than that of the CSA. Unfortunately, the second term is too large for alphabets whose size is not so small. Therefore in its practical implementation [8] they reduce the index size at the cost of access time to the index.

The CDAWG (compact directed acyclic word graph) is an efficient data structure to recognize substrings in a string [1]. It is the minimum deterministic automaton that accepts all suffixes in a string and it is more compact than the suffix tree. However in an implementation of CDAWG [4] its size is about $24n$ bytes for DNA sequences, which is still large.

Another direction in space-efficient data structures for pattern matching is to develop sequential search algorithms over compressed texts. Though such algorithms have been proposed [6, 15], the algorithms have to scan the whole compressed text. As a result, their query time is proportional to the size of the compressed text and they are not efficient for huge texts.

Another index using the suffix array of a compressed text has also been proposed [19]. It is however difficult to search arbitrary strings because the compression is based on word segmentation of the text. Furthermore, this search index can be also compressed by our algorithm.

1.4. Paper Organization

The rest of this paper is organized as follows. In Section 2 we describe the suffix array [17], the original compressed suffix array [9], and succinct dictionaries heavily used in it. In Section 3 we propose new representation of the compressed suffix array and propose new operations to the index. In Section 4, we describe data structures for the CSA ,analyze its size, and describe how to construct it.

2. PRELIMINARIES

2.1. Suffix Arrays

Let $T[1..n] = T[1]T[2]\cdots T[n]$ be a text of length n on an alphabet $\mathcal{A}$. For each symbol in the alphabet, a distinct number in $\{1, 2, \ldots, |\mathcal{A}|\}$ is assigned. The order of symbols is defined by the numbers. We assume

that $T[n+1] = \$$ is a unique terminator whose order is assigned to 0. A substring $T[j..n]$ is called a suffix of T. The suffix array $SA[1..n]$ of T is an array of integers j that represent suffixes $T[j..n]$. The integers are sorted in lexicographic order of corresponding to suffixes. Lexicographic order of two suffixes are defined as follows:

DEFINITION 2.1. $T[i..n] < T[j..n] \iff T[i] < T[j]$, or $T[i] = T[j]$ and $T[i+1..n] < T[j+1..n]$.

Because $T[n+1]$ is unique, no suffix has the same lexicographic order. Furthermore, because $T[n+1]$ is smaller than any character in the alphabet, $SA_k[0] = n_k + 1$ always holds.

The existential query, that is, whether a pattern P of length m exists in the text T or not, can be performed by a binary search on the suffix array in $O(m \log n)$ time. Since suffixes which match with a pattern P exist in a consecutive region of the suffix array, the counting query that returns the number of overlapped occurrences of the pattern is done in the same time complexity as the existential query. The counting query is performed by finding the rightmost and the leftmost indices r and l of the suffix array that correspond to the pattern. Therefore positions of all occurrences of the pattern can be enumerated in time proportional to the number of occurrences $occ = r - l + 1$ after the counting query. The positions are stored in $SA[l], SA[l+1], \ldots, SA[r]$. This is called enumerative query.

The size of the suffix array is $n \log n$ bits. The text T is also used for string comparisons. Its size is $n \log |\mathcal{A}|$ bits.

2.2. Static Dictionaries with Constant Query Time

We review some space efficient data structures for static dictionaries that support the following operations:

DEFINITION 2.2. For a bit-vector $B[1..n]$, $rank(B, i)$ is defined as the number of ones in $B[1..i]$, $select(B, i)$ is defined as the position of i-th one from left in $B[1..n]$, and $pred(B, i)$ is defined as the position of the rightmost one in $B[1..i]$.

A basic data structure is the one for rank and select queries:

THEOREM 2.1 (Jacobson and Munro [12, 20]). *A static dictionary supporting constant time rank and select queries can be stored in $n + o(n)$ bits.*

Note that $pred(B, i)$ can be computed by $select(B, rank(B, i))$ and $rank(B, select(B, i)) = i$ holds for any i. More compact dictionary for *rank* and *pred* queries is due to Pagh [21].

THEOREM 2.2. (Pagh [21, Prop. 4.3]). *A static dictionary with constant query time, supporting rank, select and pred queries, can be stored in*

$k \log \frac{n}{k} + k(1 + \log e) + O(\frac{n \log\log n}{\log n})$ *bits where* k *is the number of ones in the bit-vector* B *if* $n \leq k(\log k)^{O(1)}$.

Because we need not to store B explicitly, the size will be smaller than n bits if k is small. For *select* operation we use the data structure used in the original compressed suffix array:

LEMMA 2.1 (Grossi and Vitter[10]). *Given* s *integers in sorted order, each containing* w *bits, where* $s < 2^w$*, we can store them with at most* $s(2 + w - \lfloor \log s \rfloor) + O(s/\log\log s)$ *bits, so that retrieving the* h*-th integer takes constant time.*

We also use another compact dictionary for *select* queries.

THEOREM 2.3 (Tarjan and Yao[23]). *A sequence of integers* $x_1, \ldots, x_k$*, where for all* $1 \leq i \leq k$ *we have* $|x_i| = n^{O(1)}$ *and* $\min\{|x_i|, |x_i - x_{i-1}|\} = \mathrm{polylog}(n)$*, can be stored in a data structure allowing constant time random access, using* $O(k \log\log n)$ *bits.*

2.3. The Original Compressed Suffix Array

The compressed suffix array has size $O(n)$ bits whereas it stores the same information as the suffix array. It calculates an element of the suffix array $SA[i]$ in $O(\log^\epsilon n)$ time where $1 \geq \epsilon > 0$ is a fixed constant. Therefore both an existential and a counting queries are done in $O((m + \log^\epsilon n) \log n)$ time by a straightforward binary search. An enumerative query takes additional $O(occ \log^\epsilon n)$ time.

The compressed suffix array has a hierarchical data structure. The k-th level implicitly stores indices of suffixes which are multiple of 2^k. An array $SA_k[1..n_k]$ ($n_k = \frac{n}{2^k}$) stores the indices that are divided by 2^k. The indices are stored in the same order as in the suffix array SA.

The array SA_k becomes the suffix array of a new string $T_k[1..n_k]$. A character $T_k[j]$ consists of a concatenation of 2^k characters $T[j2^k..(j+1)2^k - 1]$. Then the array SA_k coincides with the suffix array of the string T_k. Therefore we use the same technique of representation of SA_k recursively.

We use levels $0,\ e,\ 2e, \ldots, l$ where $e = \lceil \epsilon \log\log n \rceil$ $(1 \geq \epsilon > 0)$ and $l = \lceil \log\log n \rceil$. The l-th level explicitly stores $\frac{n}{\log n}$ indices. Its size is at most $\frac{n}{\log n} \log n = n$ bits. The k-th level stores a bit-vector $B_k[1..n_k]$ and a function $\Psi_k[i]$ $(1 \leq i \leq n_k)$ instead of $SA_k[1..n_k]$.

An element $B_k[i]$ of the bit-vector represents whether $SA_k[i]$ is a multiple of 2^k or not. If $B_k[i] = 1$, $SA_k[i] = j2^k$ and the index j is stored in SA_{k+e} implicitly if $k + e < l$ or explicitly if $k + e = l$. The lexicographic order i' of a suffix in SA_{k+e} corresponding to $SA_k[i]$ is calculated by $i' = rank(B_k, i)$.

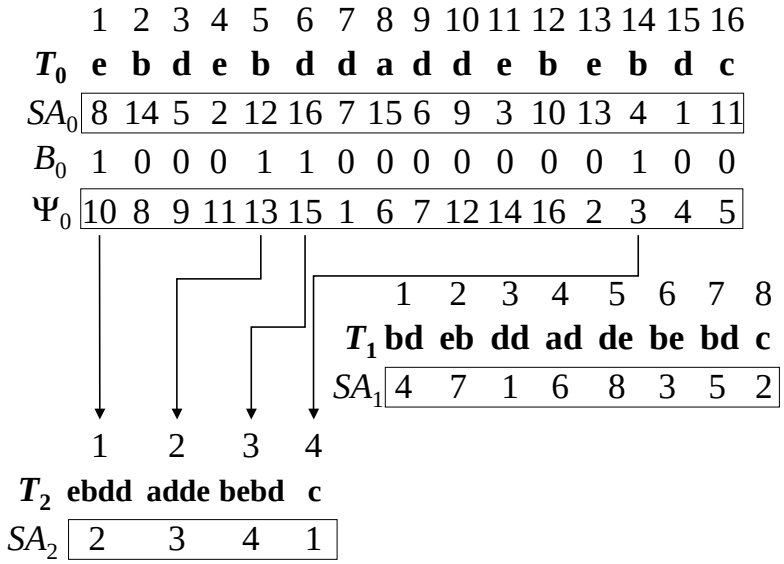

FIG. 1. An example of the compressed suffix array

Therefore $SA_k[i]$ is represented by

$$SA_k[i] = 2^e SA_{k+e}[rank(B_k, i)]$$

if $B_k[i] = 1$.

Figure 1 shows the compressed suffix array of a string $T = ebdebddaddebebdc$ where $n = 16$ and $l = \lceil \log \log n \rceil = 2$. We use $\epsilon = \frac{1}{2}$; therefore two levels 0 and 2 are stored. The arrows show the correspondence between two suffixes in different levels. In level 0, B_0 and Ψ_0 are stored, while in level 2 only SA_2 is stored. The vector B_0 shows whether $SA_0[i]$ is a multiple of $\log^\epsilon n = 4$ or not.

If $B_k[i] = 0$, $SA_k[i] = j2^e - v$ $(1 \leq v < 2^e)$ and it is represented by v and an index i' of SA_k where $SA_k[i'] = j2^e$. The function $\Psi_k[i]$ is defined as follows:

Definition 2.3.

$$\Psi_k[i] \equiv \begin{cases} i' \text{ such that } SA_k[i'] = SA_k[i] + 1 & (\text{if } SA_k[i] < n_k) \\ 0 & (\text{if } SA_k[i] = n_k) \end{cases}$$

Therefore $SA_k[i]$ is calculated by using a relation

$$SA_k[i] = SA_k[\Psi_k[i]] - 1$$

iteratively for v times while $B_k[i] = 0$.

The algorithm to calculate SA_k becomes as follows.

Algorithm $lookup_k(i)$
1. **if** $k = l$ **then return** $SA_l[i]$;
2. $v \leftarrow 0$;
3. **while** $B_k[i] = 0$
4. **do if** $i = pos_k^n$ **then return** $n_k - v$;
5. $i \leftarrow \Psi_k[i]$;
6. $v \leftarrow v + 1$;
7. **return** $2^e \cdot lookup_{k+e}[rank(B_k, i)] - v$.

The variable pos_k^n is necessary if n_k is not a multiple of $\log^\epsilon n$. It represents the lexicographic order of the last suffix $T[n_k]$, that is, $SA_k[pos_k^n] = n_k$.

The function to compute $SA[i]$ is obvious, $SA[i] = SA_0(i)$, and we have the following:

THEOREM 2.4. *[9] The compressed suffix array returns $SA[i]$ in $O(\log^\epsilon n)$ time for any fixed constant $1 \geq \epsilon > 0$.*

2.4. Entropy of Texts

Here we define the entropy of texts. First we define the entropy of a discrete random variable.

DEFINITION 2.4. Let X be a discrete random variable X with alphabet $\mathcal{X}$ and probability mass function $p(x) = \Pr\{X = x\}$ $(x \in \mathcal{X})$. Then the entropy $H(X)$ of X is defined by $H(X) = -\sum_{x\in\mathcal{X}} p(x) \log p(x)$.

We can also define the entropy of a text T on alphabet $\mathcal{A}$. We call this entropy order-0 entropy and denote by $H_0(T)$.

We need to express the entropy of a text T after *symbol expansion*, that is, the entropy of T_k where a character consists of 2^k consecutive characters in T. To do so, we use the following:

PROPOSITION 2.1. *For a stationary stochastic process, $H(X_1, X_2, \ldots, X_n) \leq nH(X_1)$.*

Proof.

$$H(X_1, X_2, \ldots, X_n) \;=\; \sum_{i=1}^{n} H(X_i | X_{i-1}, \ldots, X_1)$$

$$\leq \sum_{i=1}^{n} H(X_i) = nH(X_1)$$

where the first equality comes from the chain rule for entropy [3, Th. 2.5.1], the first inequality comes from the fact that conditioning recudes the entropy [3, Th. 2.6.5], and the second equality comes from that the process is stationary. ▮

Therefore we have the following Corollary for T_k:

COROLLARY 2.1. *If T is drawn from a stationary source, $H_0(T_k) \leq 2^k H_0(T)$*

Proof. From Proposition 2.1, $H_0(T_k) = H(X_1, \ldots, X_{2^k}) \leq 2^k H_0(X_1) = 2^k H_0(T)$ where X_i is a random variable with alphabet $\mathcal{A}$. ▮

For brevity we use H_0 instead of $H_0(T)$.

3. NEW INDEXING FUNCTIONALITIES OF COMPRESSED SUFFIX ARRAY

In this section we modify the original compressed suffix array in order to use it without the text. First we show that the text T is not necessary to perform a binary search to find a pattern P. Then we show that the text can be extracted from the compressed suffix array. We also propose a function to compute the inverse of the suffix array.

3.1. Pattern Matching using the Compressed Suffix Array without Text

An occurrence of a pattern $P[1..m]$ in a text $T[1..n]$ can be found in $O(m \log n)$ time by a binary search on the suffix array $SA[1..n]$ of the text T. In the binary search substrings $T[SA[i]..SA[i] + m - 1]$ are compared with the pattern. It will take $O((m + \log^{\epsilon} n) \log n)$ time if the compressed suffix array is used because it takes $O(\log^{\epsilon} n)$ time to calculate $SA[i]$. We show that the substring can be found without calculating the value of $SA[i]$ if the index i is given.

We use bit-vectors $D_k[1..n_k]$ defined as

$$D_k[i] \equiv \begin{cases} 1 & \text{if } i = 1 \text{ or } T_k[SA_k[i]] \neq T_k[SA_k[i-1]] \\ 0 & \text{otherwise} \end{cases} .$$

Then the bit-vector D_0 is used to obtain the head character $T[SA[i]]$ of a suffix if its lexicographic order i is known. We use an array $C[1..|\mathcal{A}|]$ in

which $C[j]$ stores the j-th smallest character appeared in T, and define a function $C^{-1}[i]$ as

$$C^{-1}[i] \equiv C[rank(D_0, i)],$$

then the following proposition holds:

PROPOSITION 3.1. *$T[SA[i]] = C^{-1}[i]$ and it is calculated in constant time for a given i.*

Proof. Since suffixes are lexicographically sorted in the suffix array, the head characters $T[SA[i]]$ of suffixes are alphabetically sorted. $D_0[i] = 1$ means that $T[SA[i]]$ is different from $T[SA[i-1]]$. Therefore the rank $r = rank(D_0, i)$ represents the number of different characters in $T[SA[1]]$, $T[SA[2]]$, ..., $T[SA[i]]$, and $C[r]$ becomes the character $T[SA[i]]$. Concerning the time complexity, the rank function takes constant time, and other operations also take constant time. ∎

This means that it is not necessary to calculate the exact value of $SA[i]$, which takes $O(\log^{\epsilon} n)$ time, to obtain the head character of the suffix $T[SA[i]..n]$. This is extended as follows. The function Ψ denotes Ψ_0, and Ψ^v denotes the iterated composition of Ψ for v times.

PROPOSITION 3.2.

$$T[SA[i] + v] = C^{-1}[\Psi^v[i]] \text{ for } 0 \leq v \leq n - SA[i]$$

Proof. From Definition 2.3,

$$SA[i] + v = SA[\Psi^v[i]].$$

By substituting the i in Proposition 3.1 by $\Psi^v[i]$,

$$T[SA[i] + v] = T[SA[\Psi^v[i]]] = C^{-1}[\Psi^v[i]]$$

holds. ∎

This proposition shows that a substring of length m, $T[SA[i]..SA[i] + m - 1]$ can be decoded in $O(m)$ time by using the Ψ and the C^{-1} functions m times. For this purpose, we modify the compressed suffix array to store values of $\Psi[i]$ for all i. The algorithm becomes as follows. It extracts a substring $T[SA[i]..SA[i] + m - 1]$ into $S[1..m]$ in $O(m)$ time.

Algorithm *substring*(i, m)

1. **for** $j \leftarrow 1$ **to** m
2. **do** $S[j] \leftarrow C^{-1}[i]$;
3. $i \leftarrow \Psi[i]$;
4. **return** S;

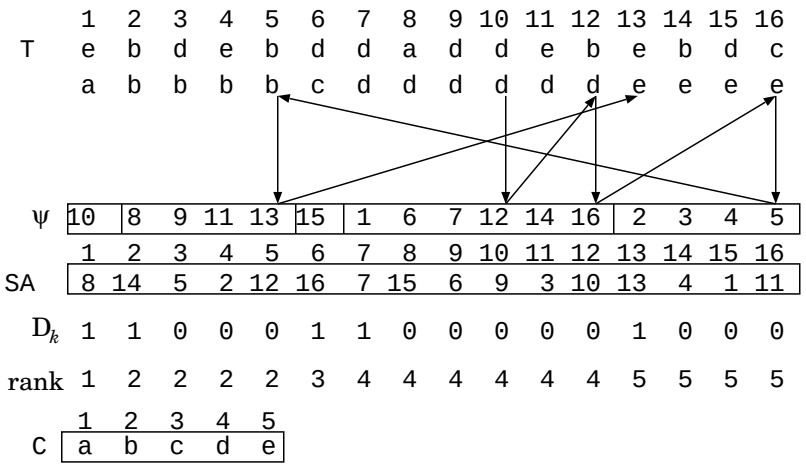

FIG. 2. Decoding a substring

Here we have the following theorem.

THEOREM 3.1. *An existential and a counting query for a pattern* $P[1..m]$ *from the compressed suffix array of a text* $T[1..n]$ *takes* $O(m \log n)$ *time.*

Proof. The search is performed by a binary search on the suffix array $SA[1..n]$. In each iteration of the binary search, a substring $T[SA[i]..SA[i]+m-1]$ is compared with the pattern $P[1..m]$. The substring $T[SA[i]..SA[i]+m-1]$ is decoded in $O(m)$ time from Proposition 3.2, which is the same time complexity as comparing two strings of length m. ∎

Recall that it takes $O((m + \log^{\epsilon} n) \log n)$ time if the original compressed suffix array is used.

3.2. Decoding the Text and the Inverse of the Suffix Array

Algorithm *substring* can decode a substring of the text. However, it is not possible to decode an arbitrary substring $T[s..e]$ if the lexicographic order i of the suffix $T[s..n]$ ($SA[i] = s$) is not known. Therefore we need to calculate the inverse of the suffix array, $i = SA^{-1}[s]$.

A basic idea of the algorithm is the following. Assume that we know $i = SA_{k+e}^{-1}[q]$. Then there exists $i' \in [1, n_k]$ such that $B_k[i'] = 1$ and $i = rank(B_k, i')$. Then i' can be computed by $i' = select(B_k, i)$. Because $i' = SA_k^{-1}[2^e q]$ holds, we can compute $SA_k^{-1}[2^e q + v]$ by $\Psi_k^v[i']$.

We traverse the hierarchy of the data structure in the opposite direction of computing $SA[i]$. We store the inverse array SA_l^{-1} explicitly in the l-th level. We also use the directory for the select function for B_k to move from a higher level to a lower level. In each level we store $pos_k^1 = i$ such that $SA_k[i] = 1$ because $SA_k^{-1}[0]$ does not exist. The algorithm becomes as follows.

Algorithm *inverse*(j)
1. $e \leftarrow \lceil \epsilon \log\log n \rceil$; $l \leftarrow \lceil \log\log n \rceil$;
2. $k \leftarrow 0$;
3. **while** $k < l$
4. **do** $r[k] \leftarrow j \bmod 2^e$;
5. $q[k] \leftarrow j/2^e$;
6. $j \leftarrow j/2^e$; $k \leftarrow k + e$;
7. $i \leftarrow SA_l^{-1}[j]$;
8. $k \leftarrow l - e$;
9. **while** $k \geq 0$
10. **do if** $q[k] = 0$
11. **then** $i \leftarrow pos_k^1$;
12. **for** $d \leftarrow 1$ **to** $r[k] - 1$
13. **do** $i \leftarrow \Psi_k[i]$;
14. **else** $i \leftarrow select(B_k, i)$;
15. **for** $d \leftarrow 0$ **to** $r[k] - 1$
16. **do** $i \leftarrow \Psi_k[i]$;
17. $k \leftarrow k - e$;
18. **return** i.

LEMMA 3.1. *Algorithm inverse computes* $SA^{-1}[j]$ *in* $O(\log^\epsilon n)$ *time.*

Proof. We prove the lemma by induction on the level k. If $k = l$, $SA_k^{-1}[j]$ is explicitly stored. Assume that $i = SA_{k+e}^{-1}[q]$ $(q \geq 1)$ is known. We show that $SA_k^{-1}[2^e q + r]$ $(0 \leq r < 2^e)$ can be computed by the algorithm. The suffix $T_k[2^e q..n_k]$ corresponds to the suffix $T_{k+e}[q..n_{k+e}]$. Let i' be the index such that $2^e q = SA_k[i']$, that is, $i' = SA_k^{-1}[2^e q]$. Then an equality $i = rank(B_k, i')$ holds. Therefore we can compute i' by $i' = select(B_k, i)$. We can also compute $SA_k^{-1}[2^e q + r] = \Psi_k^r[i']$.

For elements $SA_k^{-1}[r]$ $(0 \leq r < 2^e)$, we can calculate them by $\Psi_k^{r-1}[pos_k^1]$ because $pos_k^1 = SA_k^{-1}[1]$ is stored.

As for the time complexity, we use the Ψ_k function at most $2^e - 1 = \log^\epsilon n - 1$ times and the select function once for each level, and the number of the levels is a constant ($\frac{1}{\epsilon}$). Since both the Ψ_k function and the select function take constant time, the lemma holds. ∎

An arbitrary substring of a text can be decoded by using Algorithm *substring* and Algorithm *inverse*.

THEOREM 3.2. *A substring $T[s..s+m-1]$ of length m of a text of length n can be extracted from the proposed compressed suffix array of the text in $O(m + \log^\epsilon n)$ time.*

Proof. The lexicographic order i of the suffix $T[s..n]$ is computed in $O(\log^\epsilon n)$ time by Algorithm *inverse*. Then the substring is extracted in $O(m)$ time by Algorithm *substring*. ∎

4. THE DATA STRUCTURE OF CSA

In this section we analyze the size of the proposed compressed suffix arrays. The main result is the following:

THEOREM 4.1. *A compressed suffix array for a text of length n with alphabet size* polylog(n)*, which supports $O((|P| + occ) \log n)$ time enumerative query for a pattern P, $O(m + \log n)$ time decoding for a part of the text of length m, and $O(\log n)$ time lookup and inverse, can be stored in $n(H_0 + 3 + 2\log(1 + H_0)) + o(n)$ bits where H_0 is the order-0 entropy of the text*

Note that the size of the compressed suffix array can be expressed as $nH_0 + O(n \log\log|\mathcal{A}|)$ bits. Because the original text occupies $n \log|\mathcal{A}|$ bits, the CSA will become smaller than the text size.

We can reduce the time for lookup and inverse at the cost of increasing the size of the data structure by using the hierarchical data structure:

THEOREM 4.2. *A compressed suffix array supporting $O(\frac{1}{\epsilon\epsilon'} \log^\epsilon n)$ time lookup and inverse can be stored in*

$$n\left(\frac{1+\epsilon'}{\epsilon}H_0 + 2\log(1+H_0) + 3\right) + o(n)$$

bits for any $0 < \epsilon < 1$ and $\epsilon' > 0$.

The proposed compressed suffix array consists of the following components: For levels $0, e, 2e, \ldots, l-1$ where $e = \lceil \epsilon \log\log n \rceil$ $(1 \geq \epsilon > 0)$ and $l = \lceil \log\log n \rceil$, we store

- $B_k[1..n_k]$: representing $SA_k[i]$ is a multiple of $\log^\epsilon n$ or not
- S_k: the bit-stream storing $d_k[i] = \Psi_k[i] - \Psi_k[i-1]$
- $D_k[1..n_k]$: indicating $T_k[SA_k[i]] \neq T_k[SA_k[i-1]]$ or not
- $E_k[1..n_k]$: indicating $\Psi_k[i]$ is a representative or not
- R_k, P_k: values of representatives of $\Psi_k[i]$, and pointers to S_k which correspond to the representatives,
- pos_k^1, pos_k^n: $SA_k^{-1}[1]$ and $SA_k^{-1}[n]$.

In level 0, we also store the array $C[1..|\mathcal{A}|]$ to recover characters in the text. Its size is $|\mathcal{A}| \log |\mathcal{A}| = O(\mathrm{polylog}(n) \log\log n) = o(n)$ bits. In level l, we store SA_l and SA_l^{-1} explicitly in at most $2n$ bits. The data structure of Theorem 4.1 uses only levels 0 and l.

4.1. The Representation of Ψ Function

First we show the data structure for S_k and analyze its size z_k.

LEMMA 4.1. $z_k \leq n_k(1 + H_0(T_{2^k}) + 2\log(1 + H_0(T_{2^k})))$.

To achieve this size, we use the following property:

PROPOSITION 4.1. *$\Psi_k[i] < \Psi_k[j]$ if $i < j$ and $T_k[SA_k[i]] = T_k[SA_k[j]]$.*

Proof. If $T_k[SA_k[i]] = T_k[SA_k[j]]$, the lexicographic order of two suffixes $T_k[SA_k[i]..n_k]$ and $T_k[SA_k[j]..n_k]$ are determined by the lexicographic order of $T_k[SA_k[i]+1..n_k]$ and $T_k[SA_k[j]+1..n_k]$. Since $i < j$, $T_k[SA_k[i]+1..n_k] < T_k[SA_k[j]+1..n_k]$, or equivalently, $T_k[SA_k[\Psi_k[i]]..n_k] < T_k[SA_k[\Psi_k[j]]..n_k]$. This means that $\Psi_k[i] < \Psi_k[j]$, which concludes the proof. ∎

This proposition shows that values of the Ψ_k function are piecewise monotone increasing. This leads to a compact representation of the Ψ_k function. Instead of encoding $\Psi_k[i]$ as it is, we encode $d_k[i]$, defined as follows, by δ-code [5]:

$$d_k[i] = \begin{cases} \Psi_k[i] - \Psi_k[i-1] & \text{if } T_k[SA_k[i]] = T_k[SA_k[i-1]] \\ \Psi_k[i] & \text{otherwise.} \end{cases}$$

From Proposition 4.1 $d_k[i]$ is always positive except $SA_k[i] = n_k$. The δ-code encodes a natural number r $(r \geq 1)$ in $1 + \lfloor \log r \rfloor + 2\lfloor \log(1 + \lfloor \log r \rfloor) \rfloor$ bits. Table 1 shows examples of δ-code.

Now we prove Lemma 4.1:

TABLE 1.

	δ-code
r	$\delta(r)$
1	1
2	0 10 0
3	0 10 1
4	0 11 00
7	0 11 11
8	00 100 000
15	00 100 111

Proof. Let n_c be the number of $\Psi_k[i]$ values corresponding to $c = T_k[SA_k[i]] = T[SA_k[i]2^k..(SA_k[i]+1)2^k-1]$. Then the number of bits, say z_k^c, to encode the values for a character $c \in \mathcal{A}^{2^k}$ becomes

$$\begin{aligned}
z_k^c &= \sum_{i=1}^{n_c} (1 + \lfloor \log d_i \rfloor + 2 \lfloor \log (1 + \lfloor \log d_i \rfloor) \rfloor) \\
&\leq \sum_{i=1}^{n_c} (1 + \log d_i + 2(\log(1 + \log d_i))) \\
&= n_c \sum_{i=1}^{n_c} \frac{1}{n_c} (1 + (\log d_i) + 2(\log(1 + \log d_i))) \\
&\leq n_c \left(1 + \left(\log \sum_{i=1}^{n_c} \frac{d_i}{n_c}\right) + 2 \log \left(1 + \log \sum_{i=1}^{n_c} \frac{d_i}{n_c}\right)\right) \\
&\leq n_c \left(1 + \log \frac{n_k}{n_c} + 2 \log \left(1 + \log \frac{n_k}{n_c}\right)\right)
\end{aligned}$$

where the second inequality comes from Jensen's inequality [3] for a concave function log.

We calculate the total number of bits to encode all $\Psi_k[i]$ values for a level k. The probability p_c that a character c appears in the text T_k is expressed by $p_c = \frac{n_c}{n_k}$. Therefore we can express z_k by the order-0 entropy of the text T_{2^k}.

$$z_k \leq \sum_{c \in \mathcal{A}^{2^k}} z_k^c$$

$$
\begin{aligned}
&= \sum_{c\in\mathcal{A}^{2^k}} n_c \left(1+\log\frac{n_k}{n_c}+2\log\left(1+\log\frac{n_k}{n_c}\right)\right)\\
&= n_k \sum_{c\in\mathcal{A}^{2^k}} p_c \left(1+\log\frac{1}{p_c}+2\log\left(1+\log\frac{1}{p_c}\right)\right)\\
&\le n_k(1+H_0(T_{2^k})+2\log(1+H_0(T_{2^k})))
\end{aligned}
$$

∎

We express the size of the Ψ_k function by the order-0 entropy of T.

COROLLARY 4.1. *$z_k = nH_0 + o(n)$ if $k = d\log\log n$ for $d > 0$, and $z_0 \le n(1+H_0+2\log(1+H_0))$.*

Proof. If $k = 0$, it is easy to see $z_0 \le n(1+H_0+2\log(1+H_0))$. If $k > 0$,

$$
\begin{aligned}
z_k &\le n_k(1+H_0(T_{2^k})+2\log(1+H_0(T_{2^k})))\\
&\le n_k(1+2^kH_0+2\log(1+2^kH_0))\\
&\le n_k(1+2^kH_0+2(1+(k+1)\log H_0))\\
&= nH_0+n_k(3+2(k+1)\log H_0)
\end{aligned}
$$

where the second inequality comes from Corollary 2.1, and the third inequality comes from $\log(1+2^kH_0) \le \log(2\cdot\max\{1,2^kH_0\}) \le \log(2\cdot 1)+\log(2\cdot 2^kH_0)$. Furthermore, if $k = d\log\log n$ $(k > 0)$,

$$
\begin{aligned}
z_k &= nH_0 + n\left(\frac{3+2(d\log\log n+1)\log H_0}{\log^d n}\right)\\
&\le nH_0 + n\left(\frac{3+2(d\log\log n+1)\cdot c\log\log n}{\log^d n}\right)\\
&= nH_0+o(n)
\end{aligned}
$$

where the first inequality comes from $H_0 \le \log|\mathcal{A}| \le c\log\log n$. ∎

4.2. The Size of Directories

Next we consider the directory to compute $\Psi_k[i]$ in constant time. We can decode a sequence of δ-codes in constant time using a table of $o(n)$ bits if the total length of the encoding is $O(\log n)$ bits. Therefore we divide the sequence into segments of size $O(\log n)$ bits, and we explicitly store the Ψ_k values as *representatives* for the head element in each segment. The representatives are stored separately in R_k, and other values are encoded

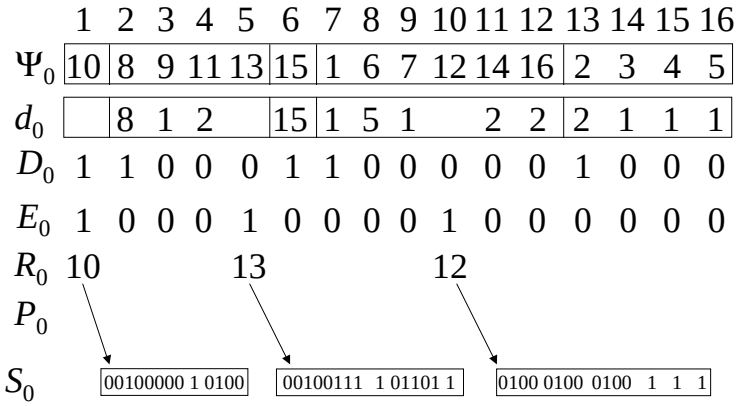

FIG. 3. The data structure for the Ψ_k function

as the δ-code of the difference $d_k[i]$. We use a bit-vector $E_k[1..n_k]$ which indicates whether $\Psi_k[i]$ is a representative ($E_k[i] = 1$) or not. We also need to store pointers P_k to the starting points of decoding in the bit-stream. Figure 3 shows the data structures for the Ψ_0 function. The bit-stream S_0 is divided into segments of length at most 16 bits.

To compute a value of $\Psi_k[i]$, we first obtain the closest representative to i and the pointer to the bit-stream, then sequentially decode the bit-stream. Because the total length to decode sequentially is $O(\log n)$, it is done in constant time using a table of $o(n)$ bits.

We also store $\Psi_k[i]$ as it is for i such that $D_k[i] = 1$ because such values may be smaller than the adjacent values. Therefore we need to discard the sequentially decoded values when we meet $d_k[i]$ with $D_k[i] = 1$.

Figure 4 explains how to decode Ψ values. The Ψ_k function is piecewise monotone function and each increasing sequence is depicted as a line. White circles show representatives. Each i such that $E_k[i] = 1$ corresponds to a representative. An easy case is to decode $\Psi_k[x]$. First we compute $s = \mathit{pred}(E_k, x)$ and obtain $\Psi_k[s]$. Then we decode $d_k[s+1], d_k[s+2], \ldots, d_k[x]$ in constant time by using a lookup table.

A slightly complicated case in Figure 4 is to decode $\Psi_k[y]$. Because $\Psi_k[s]$ ($s = \mathit{pred}(E_k, y)$) and $\Psi_k[y]$ belong to different increasing sequences, we cannot decode d_k by a table lookup. We have to know the head element in an increasing sequence to which $\Psi_k[y]$ belongs. The position of the head element can be computed by $t = \mathit{pred}(D_k, y)$. From that position we decode $d_k[t], d_k[t+1], \ldots, d_k[y]$ by using a lookup table. Because $d_k[t]$ is the head element of a increasing sequence it is equal to $\Psi_k[t]$. All other values

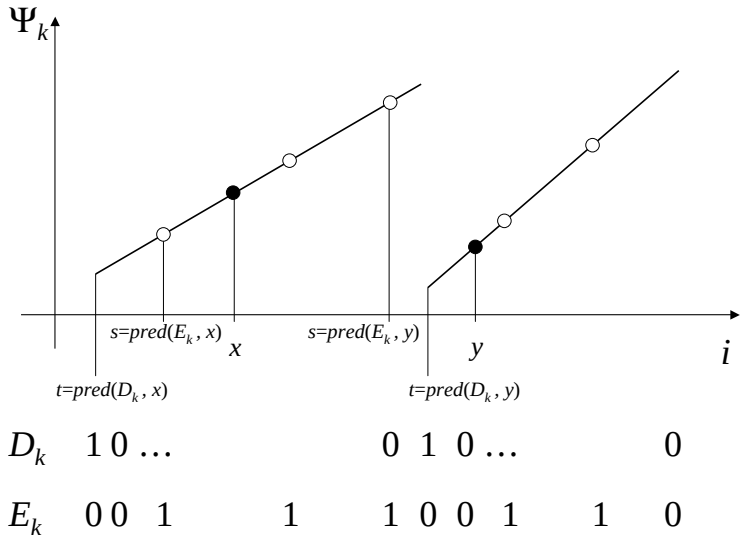

FIG. 4. How to decode $\Psi_k[i]$

$d_k[t+1], d_k[t+2], \ldots, d_k[y]$ represent the difference from the preceding one. Details of the algorithms are shown in Algorithm Ψ_k.

Algorithm $\Psi_k(i)$
1. $m \leftarrow rank(E_k, i)$; ($m$ is the number of representatives in $d_k[1..i]$)
2. $s \leftarrow pred(E_k, i)$; ($\Psi_k[s]$ is the nearest representative to $\Psi_k[i]$)
3. $\Psi \leftarrow select(R_k, m)$; ($\Psi = \Psi_k[s]$)
4. $p \leftarrow select(P_k, m)$; ($p$ is the pointer to $d_k[s]$ in S_k)
5. $t \leftarrow pred(D_k, i)$; ($t$ is the leftmost index with $T_k[SA_k[t]] = T_k[SA_k[i]]$)
6. **if** $(t > s)$ (if the representative belongs to a different increasing sequence)
7. **then** decode $t - s$ numbers from the position p in the bit-stream and discard them
8. $\Psi \leftarrow 0$;
9. $s \leftarrow t - 1$;
10. decode $i - s$ numbers in the bit-stream and add them to Ψ
11. **return** Ψ;

It is obvious that Algorithm Ψ_k runs in constant time. We analyze the size of the directories for Ψ_k function:

LEMMA 4.2. *For a text of length n with alphabet size* polylog(n)*, the value of $\Psi_k[i]$ can be computed in constant ($O(\frac{1}{\epsilon'})$) time by using data*

structures of size at most $n(1+H_0+2\log(1+H_0))+o(n)$ *bits for* $k=0$, *and* $(1+\epsilon')nH_0+o(n)$ *bits for* $k>0$ *where* $\epsilon'>0$ *is an arbitrary constant.*

Proof. The numbers of bits for the data structure for the components D_k, E_k, P_k, R_k, SA_l and SA_l^{-1} become as follows. For the bit-vector D_k, we have to support *rank* and *pred* operations for computing Ψ_k. For $k=0$ we use the directory of Theorem 2.2. Because the number of ones in D_k is at most $|\mathcal{A}|$, we cannot use the directory directly. To use it we add $O(\frac{n}{\log n})$ dummy ones at the end of D_k. Then the size is

$$\left(|\mathcal{A}|+O\left(\tfrac{n}{\log n}\right)\right)\log\frac{n+O\left(\frac{n}{\log n}\right)}{|\mathcal{A}|+O(\frac{n}{\log n})}+\left(|\mathcal{A}|+O\left(\tfrac{n}{\log n}\right)\right)(1+\log e)$$
$$+O\left(\frac{n\log\log n}{\log n}\right)=O\left(\frac{n\log\log n}{\log n}\right)$$

bits. For $k>0$ we use the directory of Theorem 2.1, whose size is $n_k+o(n_k)$ bits.

For the bit-vector E_k, we have to support *rank* and *pred* operations for computing Ψ_k. We use the directory of Theorem 2.2. Because there are at most $O(nH_0/\log n)$ ones in the range $[1,n_k]$, its size is at most

$$\begin{aligned}O\left(\frac{nH_0}{\log n}\log\frac{n_k\log n}{nH_0}+\frac{nH_0}{\log n}+o(n_k)\right) &= O\left(\frac{nc(\log\log n)^2}{\log n}+o(n_k)\right)\\ &= o(n).\end{aligned}$$

Note that $H_0\le\log|\mathcal{A}|\le c\log\log n$.

For the pointers P_k to the segments, we have to support the *select* operation. We use the directory of Theorem 2.3. Because the length of the bit-stream S_k is $O(nH_0)$, there are $O(nH_0/\log n)$ segments in the bit-stream. We store at most $\frac{O(nH_0)}{\log n}$ increasing numbers in the range $[1,O(nH_0)]$ and the difference between two adjacent numbers is $O(\log n)$. Therefore we can use the directory and the size is $O(\frac{nH_0}{\log n}\log\log n)=O(\frac{n(\log\log n)^2}{\log n})=o(n)$.

For the representatives R_k of Ψ_k we have to support the *select* operation. There are at most n_k numbers to encode but these are not completely aligned in increasing order. To form an increasing sequence we use the same technique as the original compressed suffix array [9]. Note that we use this technique to encode only the representatives and use the δ-code for other values, whereas the original method uses the former to encode all $d_k[i]$.

We convert each representative $\Psi_k[j]$ into $cn_k+\Psi_k[j]$ where $c=rank(D_k,j)$. Because $\Psi_k[j]$'s are increasing if the corresponding c are identical, the values form an increasing sequence after the conversion. Because there are at most $\min\{|\mathcal{A}|^{2^k},n_k\}$ different values of c, $cn_k+\Psi_k[j]$ is at most $n_k^2+n_k$

if $k > 0$, and at most $n|\mathcal{A}| + n$ if $k = 0$. To encode them we use the data structure of Lemma 2.1. The size is

$$\frac{O(nH_0)}{\log n}\left(2 + \log\frac{(n_k^2 + n_k)\log n}{O(nH_0)}\right) = O(nH_0)$$

if $k > 0$, and

$$\begin{aligned}&\frac{O(nH_0)}{\log n}\left(2 + \log\frac{(n|\mathcal{A}| + n)\log n}{O(nH_0)}\right)\\ =\ & O\left(\frac{nH_0\log|\mathcal{A}|}{\log n}\right) = O\left(\frac{n(\log\log n)^2}{\log n}\right) = o(n)\end{aligned}$$

if $k = 0$. For $k > 0$, we can reduce the size to less than $\epsilon' nH_0$ for an arbitrary constant $0 < \epsilon'$ by choosing the representatives for every $\frac{1}{\epsilon'}\log n$ bits of encoding. The time to extract a Ψ_k value takes $O(1/\epsilon')$ time, which is constant.

From Lemma 4.1, the total size is $n(1+H_0+2\log(1+H_0))+o(n)$ bits for $k = 0$, and $(1+\epsilon')nH_0 + o(n) + n_k + o(n_k) = (1+\epsilon')nH_0 + o(n)$ bits for $k > 0$. ∎

4.3. The Size of the CSA

Finally we analyze the size of the CSA.

Proof (of Theorem 4.1). We use only levels 0 and $l = \lceil\log\log n\rceil$. The level 0 has size $n(1+H_0+2\log(1+H_0))+o(n)$ bits and the last level has size $2n$ bits. By adding them we have the claim. ∎

Proof (of Theorem 4.2). We use levels $0,\ e,\ 2e, \ldots, l$ where $e = \epsilon\log\log n$ $(1 \geq \epsilon > 0)$ and $l = \lceil\log\log n\rceil$. Therefore we use $\frac{1}{\epsilon} + 1$ levels. For levels k $(0 \leq k < l)$, we store the bit-vector B_k, Ψ_k, and their directories. For the bit-vector B_k, we have to support *rank* and *select* operations for computing *lookup* and *inverse*. There are $n_{k+e} = n_k/\log^\epsilon n$ ones in $[1, n_k]$ where $e = \lceil\epsilon\log\log n\rceil$. Therefore we can use the directory of Theorem 2.2 and the size becomes $O(\frac{n_k\log\log n}{\log^\epsilon n})$.

The last level stores the suffix array SA_l and its inverse array SA_l^{-1} explicitly, each occupies n bits. The total size is at most

$$\begin{aligned}& n(1 + H_0 + 2\log(1 + H_0)) + \textstyle\sum_{k=1}^{\frac{1}{\epsilon}-1} n(1+\epsilon')H_0 + 2n + o(n)\\ =\ & n\left(\tfrac{1}{\epsilon}(1 + \epsilon' - \epsilon\epsilon')H_0 + 2\log(1 + H_0) + 3\right) + o(n)\\ \leq\ & n\left(\tfrac{1+\epsilon'}{\epsilon}H_0 + 2\log(1 + H_0) + 3\right) + o(n).\end{aligned}$$

We also require the array $C[1..|\mathcal{A}|]$ to store characters in the alphabet; however its size is negligible because $|\mathcal{A}|\log|\mathcal{A}| = o(n)$ bits. ∎

We can analyze the size of the original compressed suffix array in the same way. Our CSA is asymptotically only n bits larger than it because of the inverse array SA_l^{-1}, which occupies n bits, in the last level. By adding n bits we can eliminate a text whose size is $n \log |\mathcal{A}|$ bits.

4.4. How to construct CSA

Finally we show the algorithm to construct the CSA from a text T of length n. It is done in $O(n)$ time using $O(n \log n)$ bits space. The time complexity does not depend asymptotically on the parameter ϵ. We describe it briefly because there is little difference between the algorithm for the CSA and that for the original compressed suffix array.

We first construct the suffix tree of the text T and obtain the suffix array SA from it. For each level k ($0 \leq k < l = \lceil \log \log n \rceil$), the suffix array SA_{k+e} is computed in $O(n_k)$ time from SA_k by simply selecting values which are multiple of $\log^\epsilon n$ and divide them by $\log^\epsilon n$. Then we compute Ψ_k function from the suffix array SA_k.

Values of $\Psi_k[i]$ ($1 \leq i \leq n_k$) can be computed in linear time by a bucket sort. From the difinition of Ψ_k, $\Psi_k[i] = SA_k^{-1}[SA_k[i] + 1]$, which implies $T_k[SA_k[\Psi_k[i]] - 1] = T_k[SA_k[i]]$. Let $c = T_k[SA_k[j] - 1]$. Then there exists i such that $\Psi_k[i] = j$ and $T_k[SA_k[i]] = c$. Furthermore, from Proposition 4.1 $\Psi_k[i]$ is increasing if $T_k[SA[i]]$ is identical. Therefore i is also increasing for $j = 1, 2, \ldots, n_k$ such that $\Psi_k[i] = j$ and $c = T_k[SA_k[j] - 1]$. Such i belongs to the range $[L_c, R_c] \subset [1, n_k]$ such that $T_k[SA_k[i]] = c$ for any $i \in [L_c, R_c]$. The range can be computed in $O(n_k)$ time by simply counting the number of occurrences of each character c. From this observation we have a simple algorithm to compute $\Psi_k[i]$. We sort indices j of suffixes $T_k[SA_k[j]..n_k]$ for $j = 0, 1, \ldots, n_k$ such that $SA_k[j] \neq 1$ by bucket sort using $(T_k[SA_k[j]-1], j)$ as the key. This works correctly because $SA_k[0] = n_k + 1$.

The directories used in each level can be computed in $O(n_k)$ time. We can compute pos_k^1 and pos_k^n in $O(n_k)$ time by scanning SA_k. The bit-stream S_k can be created in $O(n_k)$ time although its size is $O(nH_0)$ because there are n_k elements, each of which can be output in constant time.

In level l we compute SA_l^{-1} from SA_l by bucket sort in $O(n_l)$ time. In level 0 the array C is computed in $O(n)$ time by counting the number of occurrences of each character. In the end, the whole time complexity is at most $O(n + n/2 + n/4 + \cdots) = O(n)$.

5. CONCLUDING REMARKS

We have proposed the CSA, a new representation of the compressed suffix arrays, and query algorithms on it. The original compressed suffix array of Grossi and Vitter is used with an uncompressed text, while ours aims at compressing both a text and its suffix array. We showed that a pattern

matching algorithm using the CSA has the same time complexity as that using the uncompressed suffix array. This is not achieved by using the original compressed suffix array. We also proposed algorithms to decode any substring of the text and to calculate the inverse of the suffix array. Therefore we can remove the text itself from the data structure. The size of the CSA is expressed by the order-0 entropy of the text. Therefore it will be smaller than the text on large alphabets. This means we can achieve fast queries from compressed texts.

ACKNOWLEDGMENT

The author would like to thank Prof. Takeshi Tokuyama of Tohoku university and anonymous referees for their valuable comments. This work is supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

REFERENCES

1. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.
2. M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithms. Technical Report 124, Digital SRC Research Report, 1994.
3. T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.
4. M. Crochemore and R. Vérin. Direct construction of Compact Directed Acyclic Word Graphs. In *Proc. of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97)*, LNCS 1264, pages 116–129, 1997.
5. P. Elias. Universal codeword sets and representation of the integers. *IEEE Trans. Inform. Theory*, IT-21(2):194–203, March 1975.
6. M. Farach and T. Thorup. String-matching in Lempel-Ziv Compressed Strings. In *27th ACM Symposium on Theory of Computing*, pages 703–713, 1995.
7. P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *41st IEEE Symp. on Foundations of Computer Science*, pages 390–398, 2000.
8. P. Ferragina and G. Manzini. An Experimental Study of an Opportunistic Index. In *Proc. ACM-SIAM SODA*, pages 269–278, 2001.
9. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000.
10. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching, 2001. submitted for publication.
11. D. A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer Academic Publishers, 1998.
12. G. Jacobson. Space-efficient Static Trees and Graphs. In *30th IEEE Symp. on Foundations of Computer Science*, pages 549–554, 1989.

13. J. Kärkkäinen and E. Sutinen. Lempel-Ziv Index for q-Grams. *Algorithmica*, 21(1):137–154, 1998.

14. T. Kasai, H. Arimura, R. Fujino, and S. Arikawa. Text data mining based on optimal pattern discovery – towards a scalable data mining system for large text databases –. In *Summer DB Workshop*, SIGDBS-116-20, pages 151–156. IPSJ, July 1998. (in Japanese).

15. T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A Unifying Framework for Compressed Pattern Matching. In *Proc. IEEE String Processing and Information Retrieval Symposium (SPIRE'99)*, pages 89–96, September 1999.

16. S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

17. U. Manber and G. Myers. Suffix arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

18. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(12):262–272, 1976.

19. E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In *Proc. of WSP'97*, pages 95–111. Carleton University Press, 1997.

20. J. I. Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Computer Science (FSTTCS '96)*, LNCS 1180, pages 37–42, 1996.

21. R. Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

22. S. Shimozono, H. Arimura, and S. Arikawa. Efficient Discovery of Optimal Word-Association Patterns in Large Text Databases. *New Generation Computing*, 18:49–60, 2000.

23. R. E. Tarjan and A. C.-C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.