アルゴリズム 74−10
(2000. 9. 21)

圧縮接尾辞配列を用いた文字列アルゴリズム

定兼 邦彦

東北大学大学院情報科学研究科システム情報科学専攻
〒 980-8577 仙台市青葉区片平 2-1-1
sada@dais.is.tohoku.ac.jp

全文検索のための索引である接尾辞配列は, 他の全文検索索引と比較すると省スペースであるが, 転置ファイルのような単語索引と比較するとサイズが大きい. この問題を解決するために圧縮接尾辞配列が提案されたが, 検索にはテキスト自身も必要であるため, 索引サイズはテキストよりも小さくならない.

本稿では圧縮接尾辞配列を用いた検索アルゴリズムを, テキスト自身が不要になるように変更する. また, テキスト全体やその一部を圧縮接尾辞配列から復元するアルゴリズムを提案する. これにより, テキストの圧縮と高速な検索の両立が可能となる.

Algorithms on Strings based on the Compressed Suffix Arrays

Kunihiko Sadakane

Department of System Information Sciences
Graduate School of Information Sciences, Tohoku University
Katahira 2-1-1, Aoba-ku, Sendai 980-8577, Japan
sada@dais.is.tohoku.ac.jp

A compressed text database based on the compressed suffix array is proposed. The compressed suffix array of Grossi and Vitter occupies only $O(n)$ bits for a text of length n; however it also uses the text itself that occupies $O(n \log |\Sigma|)$ bits for the alphabet Σ. On the other hand, our data structure does not use the text itself, and supports important operations for text databases: *inverse*, *search* and *decompress*. Our algorithms can find *occ* occurrences of any substring P of the text in $O(|P| \log n + occ \log^{\epsilon} n)$ time and decompress a part of the text of length l in $O(l + \log^{\epsilon} n)$ time for any fixed $1 > \epsilon > 0$. Our data structure occupies only $\frac{1}{\epsilon} nH_0 + n(6 + 3 \log H_0) \frac{\log^{\epsilon} n}{\log^{\epsilon} n - 1} + 2n + |\Sigma| \log |\Sigma| + o(n)$ bits where $H_0 \leq \log |\Sigma|$ is the order-0 entropy of the text.

1 Introduction

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 practical 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 [7] 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 [13] and suffix trees [14]. These indices enable finding any substring of a text. 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 [12, 6, 5, 9].

The compressed suffix array of Grossi and Vitter [6] reduces the size of the suffix array of a text of length n from $n \log n$ bits to $O(n)$ bits. We assume that the base of logarithm is two. We can find any pattern P in the text in $O((|P| + \log^{\epsilon}) \log n)$ time using the text and the compressed suffix array. 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 only considered binary alphabets, it can be generalized for texts with alphabet size $|\Sigma| > 2$. However, this representation also uses the text itself to search patterns. The text is also necessary to text databases because the purpose of a search is to obtain a part of the text containing a given pattern. The text occupies $n \log |\Sigma|$ bits. Indeed, the data size is always larger than that of the original text size.

Though some algorithms for finding words from a compressed text have been proposed [4, 11], 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 applicable to huge texts. Though a search index using the suffix array of a compressed text has also been proposed [15], it is 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.

The opportunistic data structure of Ferragina and Manzini [5] allows to enumerate any pattern P in a compressed text in $(|P|+occ\log^{\epsilon} n)$ time for any given $1 > \epsilon > 0$. The compression algorithm is the block sorting [1] based on a permutation of the text defined by the suffix array. It has good compression ratio and fast decompressing speed. Space occupancy of the opportunistic data structure of Ferragina and Manzini is $O(nH_k)+O(\frac{n}{\log n}(\log\log n+|\Sigma|\log|\Sigma|))$ bits where H_k is the order-k entropy of the text. Unfortunately, the second term is often too large in practice.

In this paper we propose compressed text databases and query algorithms using the compressed suffix array of Grossi and Vitter. We support three basic operations for text databases, *inverse*, *search* and *decompress*, without using the text itself. Since we do not need the original text, the size of our data structure can become smaller than the text size. 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 proximity search [10], which finds sets of keywords which appear in the neighborhood. Though the inverse of a suffix array can be computed easily from the suffix array, it also occupies $n\log n$ bits. On the other hand, our data structure can represent it by additional $n+o(n)$ bits.

The *search* returns the interval in the suffix array that corresponds to a pattern P from the text of length n in $O(|P|\log n)$ time. The *decompress* returns a substring of length l in the compressed database in $O(l+\log^{\epsilon} n)$ time. Space occupancy is only $\frac{1}{\epsilon}nH_0+n(6+3\log H_0)\frac{\log^{\epsilon} n}{\log^{\epsilon} n-1}+2n+|\Sigma|\log|\Sigma|+o(n)$ bits where H_0 is the order-0 entropy of the text and $1 > \epsilon > 0$ is a fixed constant. Assume that $n < 2^{32}$ and $H_0 = 3$, which is practical for English texts. If we use $\epsilon = 0.8$, $\log^{\epsilon} n \approx 16$. Then the size of the index is approximately $18n$ bits. On the other hand, the text itself and its suffix array occupies $8n + 32n = 40n$ bits. Therefore our search index reduces the space complexity by 55%.

The rest of this paper is organized as follows. In Section 2 we describe the original suffix array [6]. In Section 3 we describe algorithms for rank function [8] and select function [16], which are heavily used in the compressed suffix array. In Section 4 we propose modified version of the compressed suffix array. We also propose algorithms for compressed text databases. In Section 5, we describe algorithms and data structures for the Ψ_k function used in the compressed suffix arrays, and analyze the size of the data structure. In Section 6, we describe the sizes of the original and our compressed suffix arrays.

2 The original compressed suffix array

Let $T[1..n] = T[1]T[2]\cdots T[n]$ be a text of length n on an alphabet Σ. Assume that the alphabet size Σ is finite. For each symbol in the alphabet, a distinct number in $\{1, 2, \ldots, |\Sigma|\}$ is assigned. The order of symbols is defined by the numbers. We assume that $T[n] = \$$ is a unique terminator whose order is assigned to 0. We also assume that n is a power of two. 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.

The existential query, that is, whether a pattern $P[1..m]$ 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 $n\log|\Sigma|$ bits.

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 $1 > \epsilon > 0$ is a fixed constant. Therefore both an existential and a counting queries are done in $O((m+\log^{\epsilon} n)\log n)$ time. 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 $T[j2^k..(j+1)2^k-1]$.

Proposition 1 *The array SA_k coincides with the suffix array of the string T_k.*

Proof: The array SA_k is created by extracting indices $j2^k$ $(1 \le j \le n_k)$ from the SA, enumerating them in the same order as in the SA and dividing

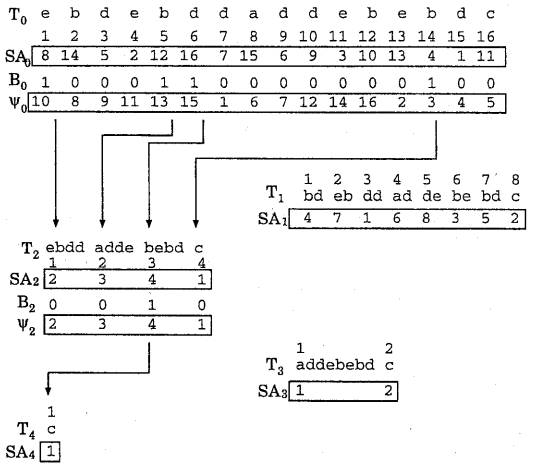

図 1: An example of the compressed suffix array

them by 2^k. Since a suffix $T_k[j..n_k]$ consists of the same character as the suffix $T[j2^k..n]$, lexicographic order of suffixes of T_k is equal to the lexicographic order of suffixes $T[j2^k..n]$. Thus the array SA_k forms the suffix array of T_k. Therefore we use the same technique of representation of SA_k recursively.

We use levels $0,\ e,\ 2e, \ldots, l$ where $e = \epsilon \log\log n$ $(1 > \epsilon > 0)$ 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 $SA_k[1..n_k]$. Figure 1 shows the compressed suffix array of a string $T = ebdebddaddebebdc$. We use $e = 2$; therefore three levels 0, 2 and 4 are stored. The arrows show the correspondence between two suffixes in different levels.

An element $B_k[i]$ of the bit-vector represents whether $SA_k[i]$ is a multiple of 2^e or not. If $B_k[i] = 1$, $SA_k[i] = j2^e$ and this index 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 $i' = rank(B_k, i)$ where $rank(B_k, i)$ returns the number of ones in $B_k[1..i]$. Therefore $SA_k[i]$ is represented by

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

if $B_k[i] = 1$.

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 1

$$\Psi_k[i] \equiv \begin{cases} i' \ s.t.\ SA_k[i'] = SA_k[i] + 1 & (\text{if } SA_k[i] < n_k) \\ i' \ s.t.\ SA_k[i'] = 1 & (\text{if } SA_k[i] = n_k) \end{cases}$$

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

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

iteratively while $B_k[i] = 0$.

The algorithm to calculate SA_k becomes as follows.

Algorithm $SA_k[i]$
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;
5. $i \leftarrow \Psi_k[i]$;
6. $v \leftarrow v + 1$;
7. **return** $2^e \cdot SA_{k+e}[rank(B_k, i)] - v$.

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, $SA_k[pos_k^n] = n_k$.

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

Proof: $SA[i]$ is calculated by the function $SA_0[i]$. The function recursively calls itself constant $(1/e)$ times. In each call the Ψ_k function is used at most $\log^\epsilon n - 1$ times because $SA_k[\Psi_k^v[i]] = SA_k[i] + v$ $(v = 0, 1, \ldots, \log^\epsilon n)$ always include an index which is a multiple of $\log^\epsilon$. Since each call of *rank* and Ψ_k functions takes constant time from Lemma 1 in Section 3 and Lemma 4 in Section refsec:psi, the theorem holds.

The original compressed suffix array is mainly used to compress indices in leaves of a suffix tree because there exists a technique to represent the shape of the suffix tree of a binary string in $O(n)$ bits [17]. This representation uses the text to find the length of edges in the suffix tree. In this paper we modify the compressed suffix array to be used without the suffix tree and the text. The size of search index may become smaller than the text size. Thus it becomes a compressed text database.

3 Rank and select functions

A function $rank(B, i)$ for a bit-vector $B[1..n]$ returns the number of ones in $B[1..i]$.

Lemma 1 *[8] The rank function can be computed in constant time by using a data structure of size $n+o(n)$ bits.*

A function $select(B, i)$ returns the position of i-th one in the bit-vector $B[1..n]$.

Lemma 2 *[16] The select function can be computed in constant time by using a data structure of size $n + o(n)$ bits.*

4 Modification of the compressed suffix array

In this section we modify the original compressed suffix array in order to use it for compressed text databases. 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 some functions for compressed text databases.

4.1 Pattern matching using the compressed suffix array without the 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 $C[1..|\Sigma|]$ 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 2 *$T[SA[i]] = C^{-1}[i]$ 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 $T[SA[1]], T[SA[2]], \ldots, T[SA[i]]$, and $C[r]$ becomes the character $T[SA[i]]$. Concerning the time complexity, the rank function takes constant time from Lemma 1, 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.

Proposition 3

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

Proof: From Definition 1,

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

By substituting the i in Proposition 2 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. The algorithm becomes as follows. It calculate a substring $T[SA[i]..SA[i]+m-1]$ in $S[1..m]$ and takes $O(m)$ 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;

For this purpose, we modify the compressed suffix array to store values of $\Psi[i]$ for all i.

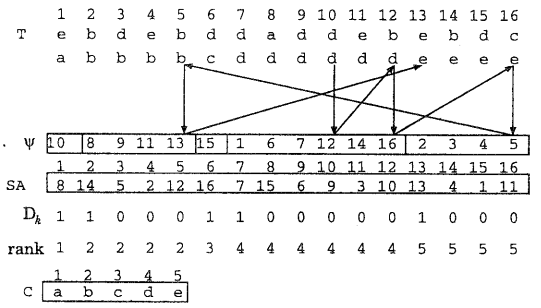

図 2: Decoding a substring

Here we have the following theorem.

Theorem 2 *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.*

Note that it takes $O((m + \log^{\epsilon} n) \log n)$ time if the original compresses suffix array is used. **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, which is the same time complexity as comparing two strings of length m.

4.2 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 want to calculate the inverse of the suffix array, $i = SA^{-1}[s]$. 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. In each level we store $pos_k^1 = i$ such that $SA_k[i] = 1$. The algorithm becomes as follows.

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

Lemma 3 *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 $T_{k+e}[q..n]$. Assume that $2^e q = SA_k[i']$. Then an equation $i = rank(D_k, i')$ holds. We can compute $i' = SA_k^{-1}[2^e q]$ by $i' = select(D_{k+e}, i)$. 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.

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

We can decode an arbitrary substring of a text by using Algorithm *substring* and Algorithm *inverse*.

Theorem 3 *A substring $T[s..s+l-1]$ of length l of a text of length n can be extracted from the proposed compressed suffix array of the text in $O(l + \log^\epsilon n)$ 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 $O(l)$ time by Algorithm *substring*.

5 The data structure of the Ψ_k function

In this section we show the following lemma:

Lemma 4 *The value of $\Psi_k[i]$ can be computed in constant time by using data structures of size at most*

$$nH_0 + n_k(5 + 3\log H_0) + o(n)$$

bits.

To achieve this size, we use the following property:

Proposition 4 *$\Psi_k[i] < \Psi_k[j]$ if $i < j$ and $T_k[SA_k[i]] = T_k[SA_k[j]]$.*

Note that $\Psi_k[i]$ and $\Psi_k[j]$ are always defined when the condition holds, otherwise $SA_k[j] = n_k$ and $T_k[SA_k[j]]$, which contains the unique character \$, is different with $T_k[SA_k[i]]$.

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. The values $\Psi_k[i]$ are grouped according to characters $T_k[SA_k[i]]$. The values in a group are encoded as a kind of lists. The value $\Psi_k[i]$ is encoded as $\Psi_k[i] - \Psi_k[i-1]$ if $T_k[SA_k[i-1]] = T_k[SA_k[i]]$, otherwise encoded as $\Psi_k[i]$, by a representation of natural numbers, for example the δ-code [3]. The values of $\Psi_k[i]$ are used only for indices with $B_k[i] = 0$. Therefore the original compressed suffix array stores only the necessary values. The number of stored values is $\frac{n(\log^\epsilon n - 1)}{\log^\epsilon n}$.

The data structure of the Ψ_k function consists of D_k, S_k, E_k, P_k^1, P_k^2, F_k^1 and F_k^2. D_k is a bit-vector of length n_k defined above with the directory of $o(n_k)$ bits for the rank function. It represents a group S_{kc} containing all $\Psi_k[i]$ such that $T_k[SA_k[i]] = c$. The S_k consists of S_{kc}'s. Other structures are used to decode $\Psi_k[i]$ values in constant time.

The S_{kc} stores values of $\Psi_k[i]$ corresponding to $c = T_k[SA_k[i]]$. From Proposition 4, the values are monotone increasing to i in a group S_{kc}. We encode $\Psi_k[i]$ as $d_i = \Psi_k[i] - \Psi_k[i-1]$ by δ-code if $D_k[i] = 0$. If $D_k[i] = 1$, we use $d_i = \Psi_k[i]$. The δ-code encodes a 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.

We calculate the size of S_k in bits.

Lemma 5 *$z_k = nH_0 + 3n_k(1 + \log H_0) + o(n)$ for $k = \epsilon \log\log n$.*

表 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

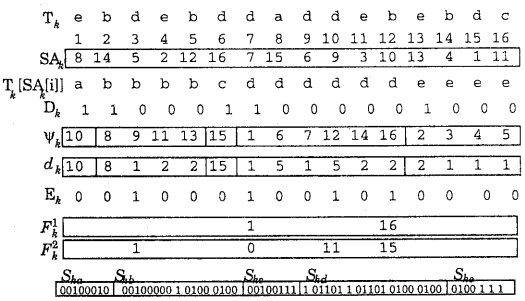

図 3: The data structure for the Ψ_k function

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 to encode the values for a $c \in \Sigma^{2^k}$ becomes

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

where the first inequality comes from Jensen's inequality for concave function log.

We calculate the total number of bits to encode all $\Psi_k[i]$ values for a level k. Let $p_c = \frac{n_c}{n_k}$, that is, the probability that a character c appears in the text T_k. The entropy of the text T_k is expressed by

$$H_{2^k} = \sum_{c \in \Sigma^{2^k}} p_c \log \frac{1}{p_c}.$$

The entropy has the following property [2]:

$$H_{2^k} \le 2^k H_0$$

where equality holds when characters in T appear independently.

Now we express z_k by the order-0 entropy H_0.

$$\begin{aligned} z_k &\le \sum_{c \in \Sigma^{2^k}} z_{kc} \\ &= \sum_{c \in \Sigma^{2^k}} n_c \left(1 + \log \frac{n_k}{n_c} + 2 \log \left(1 + \log \frac{n_k}{n_c}\right)\right) \\ &= n_k \sum_{c \in \Sigma^{2^k}} p_c \left(1 + \log \frac{1}{p_c} + 2 \log \left(1 + \log \frac{1}{p_c}\right)\right) \\ &\le n_k (1 + H_{2^k} + 2 \log(1 + H_{2^k})) \\ &\le n_k (1 + 2^k H_0 + 2 \log(1 + 2^k H_0)) \\ &\le n_k (1 + 2^k H_0 + 2 \log \max\{(2, 2^{k+1} H_0\})) \\ &= n \left(H_0 + \max\left\{\frac{3}{2^k}, \frac{2k+3}{2^k} \log H_0\right\}\right) \\ &\le n \left(H_0 + \frac{2k}{2^k} \log H_0\right) + 3n_k(1 + \log H_0) \end{aligned}$$

If $k = 0$, the first term becomes nH_0. If $k = \epsilon \log\log n$, the first term becomes $n\left(H_0 + \frac{2\epsilon \log\log n}{\log^\epsilon n} \log H_0\right) = nH_0 + o(n)$. Therefore the lemma holds.

We also use directories for the Ψ_k function to decode a $\Psi_k[i]$ value in constant time. We define a function $decode\delta(P_j, l)$ which decodes a bit-pattern of length $O(\log n)$ bits in constant time using table lookups. The $decode\delta(P_j, l)$ function returns the summation of the first l numbers encoded in P_j by δ-code. It takes constant time by using tables $\Delta_x[w, i]$, $\Delta_w[w, i]$ and $\Delta_n[w, i]$ where w is a bit-pattern of $W = \frac{\log n}{2}$ bits and $1 \le i \le W$. The element $\Delta_x[w, i]$ stores the summation of the first i numbers encoded by δ-code in a bit-pattern w and the element $\Delta_w[w, i]$ stores the total number of bits for the first i numbers. Elements $\Delta_x[w', i]$ and $\Delta_w[w', i]$ have the same value for all w' such that w' has the same prefix of length $\Delta_w[w, i]$ as the bit-pattern w. If the number of bits to encode i numbers is greater than W, We let $\Delta_n[w, i] = j$ where $j < i$ is the maximum number of numbers stored in the bit-pattern w. In this case we let $\Delta_x[w, i]$ and $\Delta_w[w, i]$ be the summation of the first j numbers and the number of bits for the first j numbers, respectively.

The size of the tables becomes as follows. Each table has $2^W \cdot W$ entries and each entry has size $\log\log n$ bits. Therefore the size of the three tables is

$$3 \cdot 2^W \cdot W \cdot \log\log n = 3\sqrt{n} \cdot \frac{\log n}{2} \cdot \log\log n = o(n)$$

bits. Note that these tables are common for all Ψ_k functions.

We can decode δ-code in constant time if the length of the encoding is $O(\log n)$ bits using a table of $o(n)$ bits. Therefore we encode $\Psi_k[i]$ explicitly for every $\log n$ bits of encoding. We use a two-level structure. The first level stores $\Psi_k[i]$ explicitly in an array F_k^1 and stores pointers to the bit-pattern of $\Psi_k[i]$ in an array P_k^1 for every $\log n \log z_k$ bits of encoding. The second level stores $\Psi_k[i]$ as the difference from values stored in the first level in an array F_k^2 and stores pointers which are relative to P_k^1 in an array P_k^2 for every $\log n$ bits are used for the encoding. We use

a bit-vector $E_k[1..n_k]$ to represent whether $\Psi_k[i]$ is encoded as it is or not. We also use directories for the rank and the select functions for E_k.

Assume that $r = rank(E_k, i)$. Then $\Psi_k[i]$ is expressed by the difference from $F_k^1[\lfloor \frac{r}{\log z_k} \rfloor] \log z_k + F_k^2[r]$. The difference is encoded in $O(\log n)$ bits. If a $\Psi_k[j]$ value, which corresponds to $F_k^2[r]$, belongs to a different $S_{kc'}$, we do not use $F_k^2[r]$. If $F_k^1[\lfloor \frac{r}{\log z_k} \rfloor \log z_k]$ and $F_k^2[r]$ belong to different S_{kc}, we do not use $F_k^1[\lfloor \frac{r}{\log z_k} \rfloor \log z_k]$ and let $F_k^2[r]$ be the difference from the first value of S_{kc}. On the other hand, the pointer to the bit-stream that encodes $\Psi_k[i]$

In Figure 3, values of $\Psi_k[i]$ are stored explicitly in F_k^1 for every 32 bits of S_k. For every 16 bits, the values are stored in F_k^2 as the difference from values in F_k^1. The first element of F_k^2 is 1 because it is the first element corresponding to S_{kb}.

The algorithm to decode the $\Psi_k[i]$ value becomes as follows:

Algorithm $\Psi_k(i)$

1. $c \leftarrow rank(D_k, i)$; $r \leftarrow rank(E_k, i)$;
2. $i_1 \leftarrow select(E_k, \lfloor \frac{r}{\log z_k} \rfloor \log z_k)$; $i_2 \leftarrow select(E_k, r)$;
3. $c_1 \leftarrow rank(D_k, i_1)$; $c_2 \leftarrow rank(D_k, i_2)$;
4. $P \leftarrow P + P_k^1[\lfloor \frac{r}{\log z_k} \rfloor] + P_k^2[r]$;
5. **if** $c = c_1$ **then**
6. $\Psi \leftarrow F_k^1[\lfloor \frac{r}{\log z_k} \rfloor] + F_k^2[r]$; $j \leftarrow i - i_2$;
7. **else if** $c = c_2$ **then**
8. $\Psi \leftarrow F_k^2[r]$; $j \leftarrow i - i_2$;
9. **else** $\Psi \leftarrow 0$;
10. $d \leftarrow select(D_k, c) - select(D_k, c_2)$;
11. $P \leftarrow P + \Delta_w[P, d]$;
12. $j \leftarrow i - select(D_k, c)$;
13. **if** $j > 0$ **then**
14. $\Psi \leftarrow \Psi + decode\delta(P, j)$;
15. **return** Ψ;

The number of bits for the directory becomes as follows. The bit-vector E_k occupies n_k bits and the directories for the rank and the select functions for E_k has size $o(n_k)$ bits each. The array P_k^1 occupies $\frac{z_k}{\log n \log z_k} \log z_k = \frac{z_k}{\log n}$ bits. The array F_k^1 occupies $\frac{z_k}{\log n \log z_k} \log n = \frac{z_{kc}}{\log z_k} < \frac{z_k}{\log n}$ bits. The arrays F_k^2 and P_k^2 occupy

$$\frac{z_k}{\log n} \log(\log n \log z_k) = \frac{z_k \log \log n}{\log n} + \frac{z_k \log \log z_k}{\log n} = o(n)$$

bits each because $H_0 \leq \log |\Sigma| = O(1)$.

The D_k is a bit-vector of length n_k. We use directories for the rank and the select functions for D_k. Their size is $n_k + o(n_k)$ bits each.

The proof of Lemma 4 is obvious. **Proof:** [of Lemma 4] From Lemma 5, the size of S_k is $z_k = nH_0 + 3n_k(1 + \log H_0) + o(n)$. The size of other structures is $2n_k + o(n_k)$ (see Table 2). By summing them, the size of Ψ_k is $z_k + 2n_k + o(n) = nH_0 + n_k(5 + 3 \log H_0) + o(n)$.

表 2: The sizes of data structures for Ψ_k function

	size (bits)		size (bits)
S_k	z_k		
E_k	$n_k + o(n_k)$	D_k	$n_k + o(n_k)$
P_k^1	$o(n_k)$	P_k^2	$o(n_k)$
F_k^1	$o(n_k)$	F_k^2	$o(n_k)$

6 The size of compressed suffix arrays

6.1 Proposed compressed suffix array

Our compressed suffix array uses levels $0, e, 2e, \ldots, l$ where $e = \epsilon \log \log n$ $(1 > \epsilon > 0)$ and $l = \lceil \log \log n \rceil$. Therefore we use $\frac{1}{\epsilon} + 1$ levels. The last level stores the suffix array SA_l and its inverse array SA_l^{-1} explicitly. They occupy n bits each. The k-th level stores a bit-vector $B_k[1..n_k]$, a function $\Psi_k[i]$ and their directories. Both the directories for the rank function and the select function for B_k have size $o(n_k)$ bits. The size of bit-vectors $B_k[1..n_k]$ for all levels is

$$n\left(1 + \frac{1}{\log^\epsilon n} + \frac{1}{\log^{2\epsilon} n} + \cdots\right) < \frac{n \log^\epsilon n}{\log^\epsilon n - 1}.$$

Now we have the main theorem.

Theorem 4 *The size of the proposed compressed suffix array is*

$$\frac{1}{\epsilon} nH_0 + n(6 + 3 \log H_0) \frac{\log^\epsilon n}{\log^\epsilon n - 1} + 2n + |\Sigma| \log |\Sigma| + o(n)$$

bits.

6.2 Comparison with the original compressed suffix array

The difference between the original compressed suffix array and ours is the following:

- Ours uses an array of alphabet size instead of the text itself.
- Ours stores $\Psi_k[i]$ values for all i, while the original one stores them for i such that $SA_k[i]$ is not a multiple of $\log^\epsilon n$.
- Ours stores the inverse suffix array SA_l^{-1} in the last level.
- Ours uses the directory for the select function in addition to the directory for the rank function for B_k.

Theorem 5 *The size of the original compressed suffix array with $O(\log^\epsilon n)$ access time $(1 > \epsilon > 0)$ is*

$$\frac{1}{\epsilon} n H_0 \frac{\log^\epsilon n - 1}{\log^\epsilon n} + n(6 + 3\log H_0)\frac{\log^\epsilon n}{\log^\epsilon n - 1} + n + n\log|\Sigma| + o(n)$$

bits.

Corollary 1 *The size of the proposed compressed suffix array is smaller than the original one by*

$$n\left(\log|\Sigma| - \frac{H_0}{\log^\epsilon n} - 1\right) - |\Sigma|\log|\Sigma| - o(n)$$

bits.

7 Concluding remarks

We have proposed algorithms and data structures for compressed text databases based on the compressed suffix array. The original compressed suffix array of Grossi and Vitter aims to compress leaves of a suffix tree, while ours aims to compress both a text and its suffix array. We showed that pattern matching by using our compressed suffix array has the same time complexity as the uncompressed suffix array. This is not achieved by the original compressed suffix array. We also proposed an algorithm to calculate the inverse of the suffix array. It is useful to text data mining.

References

[1] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithms. Technical Report 124, Digital SRC Research Report, 1994.

[2] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* Wiley-Interscience, 1991.

[3] P. Elias. Universal codeword sets and representation of the integers. *IEEE Trans. Inform. Theory*, IT-21(2):194–203, March 1975.

[4] M. Farach and T. Thorup. String-matching in Lempel-Ziv Compressed Strings. In *27th ACM Symposium on Theory of Computing*, pages 703–713, 1995.

[5] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. Technical Report TR00-03, Dipartimento di Informatica, Università di Pisa, March 2000.

[6] 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. `http://www.cs.duke.edu/~jsv/Papers/catalog/node68.html`.

[7] D. A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics.* Kluwer Academic Publishers, 1998.

[8] G. Jacobson. Space-efficient Static Trees and Graphs. In *30th IEEE Symp. on Foundations of Computer Science*, pages 549–554, 1989.

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

[10] 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).

[11] 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.

[12] S. Kurtz. Reducing the Space Requirement of Suffix Trees. Technical Report 98-03, Technische Fakultät der Universität Bielefeld, Abteilung Informationstechnik, 1998.

[13] 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.

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

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

[16] 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.

[17] J. I. Munro, V. Raman, and S. S. Rao. Space Efficient Suffix Trees. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1530, pages 186–195, 1998.