# Succinct Text Indexes on Large Alphabet[⋆]

Meng Zhang[1], Jijun Tang[2], Dong Guo[1], Liang Hu[1⋆⋆⋆], and Qiang Li[1]

[1] College of Computer Science and Technology, Jilin University,
Changchun 130012, China
zm@mail.edu.cn, {guodg, li_qiang}@jlu.edu.cn, hul@mail.jlu.edu.cn,
[2] Department of Computer Science and Engineering,
University of South Carolina, USA
jtang@cse.sc.edu

**Abstract.** In this paper, we first consider some properties of strings who have the same suffix array. Next, we design a data structure to support *rank* and *select* operations on an alphabet $\Sigma$ using $n\log|\Sigma| + o(n\log|\Sigma|)$ bits in $O(\log|\Sigma|)$ time for a text of length $n$. It also supports an extended *rank*, namely $rank^{\leq}$, such that $rank^{\leq}_{\alpha}(T, i)$ returns the number of letters which are not smaller than $\alpha$ in string $T$, plus the number of $\alpha$s up to position $i$. Also, it runs in $O(\log|\Sigma|)$ time. By this structure, we implement the DAWG succinctly. The main structure only takes $n\log|\Sigma| + o(n\log|\Sigma|)$ bits and supports basic operations of DAWG efficiently.

## 1 Introduction

Given a text string, full-text indexes are data structures that can be used to find any substring of the text quickly. Many full-text indexes have been proposed, such as suffix trees [8, 16], DAWGs [3] and suffix arrays [7, 12]. However, the major drawback that limits the applicability of full-text indexes is their space complexity–the size of full-text indices are quite larger than the original text. Standard representations of suffix tree require $4n\log n$ bits space, where log denotes the logarithm base 2.

Suffix array was proposed to reduce the space cost of suffix trees. It consists of the values of the leaves of the suffix tree in in-order, but without the tree structure information, hence takes only $n\log n$ bits. Recent researches are focused on reducing the sizes of full-text indices [6, 9, 10, 15]. The compressed suffix array structure [9] proposed by Grossi and Vitter is the first method that reduces the size of the suffix array from $O(n\log n)$ bits to $O(n)$ bits and supports access to any entry of the original suffix array in $O(\log^{\epsilon}_{|\Sigma|} n)$ time, for any fixed constant $0 < \epsilon < 1$ (without computing the entire original suffix array). FM-index proposed by Ferragina and Manzini [6] is a self-index data structure

⋆Supported by NSF of China No.60473099 and Foundation of Young Scientist of Jilin Province No.20040119.

⋆⋆⋆Corresponding author.

with good compression ratio and fast decompressing speed. The FM-index occupies at most $5nH_k(T) + o(n)$ bits of storage and allows the search for the *occ* occurrences of a pattern $P[1..p]$ within $T$ in $O(p + occ\log^{1+\varepsilon} n)$ time.

He *et al.* [10] present a succinct representation of suffix arrays of binary strings that uses $n + o(n)$ bits. For the case of large alphabet, they suggested an approach which conceptually sets a bit vector for each alphabet symbol to support operations *rank* and *select*, and uses a wavelet tree in the actual implementation to save space. They also proved a categorization theorem by which one can determine whether a given permutation is the suffix array for a binary string.

In this paper, we study the same problem over large alphabet, and develop a space-economical method to solve the problem. We first describe some properties of strings whose suffix array is a given permutation, such as at least how many different letters must occur in such strings. We then present a data structure that supports *rank* and *select* operations on large alphabet using at most $n\log|\Sigma| + o(n\log|\Sigma|)$ bits in $O(\log|\Sigma|)$ time. Although these operations can be implemented to run in constant time [6, 10], the additional space occupation will be unacceptable if $\log|\Sigma|$ can not be neglected. Thus, it is reasonable to make the operations run in $O(\log|\Sigma|)$ time to save space. The data structure also supports an extended *rank*, namely $rank^{\leq}$, which also runs in $O(\log|\Sigma|)$ time without using any additional space. More precisely, $rank^{\leq}_{\alpha}(T, i)$ returns the number of letters which are not smaller than $\alpha$ in string $T$, plus the number of $\alpha$s up to position $i$. Function $rank^{\leq}$ plays a crucial role in succinct index for large alphabet. In [6, 10], the same function of $rank^{\leq}$ is performed via a table of $|\Sigma|\log n$ bits which for each symbol stores the number of characters in the text that lexicographically precede it. Based on this index, we implement the DAWG [3, 5] in a succinct way, not storing the states and edges explicitly. The main structure only takes $n\log|\Sigma| + o(n\log|\Sigma|)$ bits for a text of length $n$ on an alphabet $\Sigma$ and supports basic operations of DAWG without loss of speed.

## 2 Basic Definitions

Let $\Sigma$ be a nonempty alphabet and $|\Sigma|$ be the number of symbols in $\Sigma$. Let $T = t_1 t_2 \ldots t_n$ be a word over $\Sigma$, $|T|$ denotes its length, $T[i]$ or $t_i$ its $i^{th}$ letter, and $T[i..]$ its suffix that begins at position $i$, $T[i..j]$ its substring begins at $i$ ends at $j$, $1 \leq i \leq j \leq n$. Let $T^R$ be the reverse string of $T$ and $Suff(T)$ the set of all suffixes of $T$ and $Fact(T)$ the set of its factors.

**Definition 1** *For a string $T$ of length $n$ over an ordered alphabet $\Sigma$. Denote the set of different letters occur in $T$ by $A(T)$. $\forall a \in A(T)$, function $Order_T(a)$ returns the numbers of letters in $A(T)$ that is not greater than $a$. For termination character \$, $Order_T(\$) = 0$. $T^*$ denotes the string of length $n$ over integer alphabet, such that $T^*[i] = Order_T(T[i])$.*

The *rank* and *select* operation play important roles in succinct data structures. Function $rank_1(B, i)$ and $rank_0(B, i)$ return the number of 1s and 0s in the bit vector $B[1..n]$ up to position $i$, respectively.

**Lemma 1.** *[11]The rank function can be computed in constant time by using a data structure of size $n + o(n)$ bits.*

Function $select_1(B, i)$ and $select_0(B, i)$ return the positions of $i^{th}$ 1 and 0, respectively.

**Lemma 2.** *[14]The select function can be computed in constant time by using a data structure of size $n + o(n)$ bits.*

For convenience, we use $rank_b(B)$, $b \in \{0, 1\}$, to denote $rank_b(B, n)$. We will also use $rank_b(B[s..i])$, $1 \le s \le i \le n$, to denote $rank_1(B, i) - rank_1(B, s - 1)$. Because *rank* runs in constant time, $rank_b(B[s..i])$ also runs in constant time.

## 3 Permutations and Suffix Arrays

Permutation $P$ can be treated as a string over alphabet $\{1, 2, \ldots, n\}$. Because $P[P^{-1}[1]] = 1 < P[P^{-1}[2]] = 2 < \ldots < P[P^{-1}[n]] = n$, where $P^{-1}$ denotes the inverse permutation of $P$, it is apparent that the suffix array of this string is $P^{-1}$ and vice versa. Among the strings who have the same suffix array, the number of different letters occur in each string can be different. The question is how to compute the minimal number of different letters occur in such strings.

The core of our solution is a simple fact, that is, for a string $T$ and its suffix array $P$, if $T[P[i-1]] = T[P[i]]$, then $T_{P[i-1]+1} < T_{P[i]+1}$, because $T_{P[i-1]} < T_{P[i]}$. Therefore, if $T_{P[i-1]+1} < T_{P[i]+1}$ is true then $T[P[i]]$ can be any letter not less than $T[P[i-1]]$, including $T[P[i-1]]$, such that the suffix array of $T$ is still $P$; otherwise it must be greater than $T[P[i-1]]$. According to this fact, we define the special positions in $P$ that increase the number of symbols must occur in $T$.

**Definition 2** *Given a permutation $P$ of $\{1, 2, \ldots, n\}$. For $1 \le i \le n$, we call $i$ an **increasing position** of $P$, if $i = 1$ or $P^{-1}[P[i-1] + 1] > P^{-1}[P[i] + 1]$.*

Since \$ is the minimal letter, therefore 1 is an increasing position of $P$. To achieve this, we assume that for any permutation $P$ of $\{1, 2, \ldots, n\}$, $P[0] = n+1$, $P[n+1] = n+2$. Thus for any string $T$ of length $n$, $T[n+2]$ should be greater than any characters in $T$. Denote the set of increasing positions of $P$ by $IP(P)$ and the number of increasing positions in $P$ by $ic(P)$. Let $I$ be an increasing position of $P$, denote the minimum increasing position greater than $I$ by $\nu(I)$. Denote the maximal non-increasing position greater than $I$ such that there is no increasing position between $I$ and this position by $\kappa(I)$.

**Definition 3** *Given a permutation $P$ of $\{1, 2, \ldots, n\}$. Let $T$ be a string over an ordered alphabet. For any increasing position of $P$, say $I$, if $t_{P[I]} \le t_{P[I+1]} \le \ldots \le t_{P[\kappa(I)]}$ and $t_{P[\kappa(I)]} < t_{P[\nu(I)]}$ if $\nu(I)$ exists, then $T$ is called a generating string of $P$. Denote the set of generating string of $P$ by $G(P)$.*

The following theorem summarizes the property of strings who have the same suffix array. The proof can be found in [17].

**Theorem 1.** *The suffix array of $T$ is $P$ if and only if $T^* \in G(P)$.*

By theorem 1, the following is immediate.

**Theorem 2.** *Given a permutation $P$ of $\{1, 2, \dots, n\}$. For any string $T$ whose suffix array is $P$, the number of different letters that occur in $T$ is at least $ic(P)$.*

By theorem 2, one can determine at least how many different letters must occur in a string whose suffix array is a given permutation. The same result was first revealed by Bannai *et al.* [2].

To generate the strings whose suffix arrays are $P$. We can set each letter on position $P[i]$ of $T$ from $P[1]$ to $P[n]$. First, the letter $T[P[1]]$ must be the minimal letter of the alphabet. Because $P$ is treated as suffix array, thus $T_{P[i-1]} < T_{P[i]}$ and $T[P[i-1]] \leq T[P[i]]$. If $i$ is not an increasing position then $T[P[i-1]]$ and $T[P[i]]$ can be set to the same letter, otherwise $T[P[i-1]] < T[P[i]]$.

Bannai *et al.* [2] presented an algorithm to generate the string consisting $ic(P)$ different letters whose suffix array is $P$. The input of the algorithm is $P$ and a string $w$ whose suffix array is $P$. If $w$ is not available, $P^{-1}$ can take this role. Then the algorithm is the same as ours.

In [10], He *et al.* gave an algorithm that checks whether a permutation is the suffix array for a given binary string. According to the theorem 1, the check over large alphabet can be done by testing whether $T^* \in G(P)$. Precisely, for all $i = 1, \dots, n-1$, if there exits $i$, such that $T[P[i]] < T[P[i-1]]$ or $T[P[i-1]] = T[P[i]]$ and $P^{-1}[P[i]+1] < P^{-1}[P[i-1]+1]$, then $P$ is not the suffix array for $T$. Otherwise, we have $T_{P[1]} < T_{P[2]} < \dots < T_{P[n]}$; therefore, $P$ is the suffix array for $T$. This simple algorithm, of course, can be used to check whether a permutation is the suffix array of a given binary string.

## 4 Succinct Indexes on Large Alphabet

For an internal node $\overline{u}$ of the suffix tree, where $u$ is the longest string in the node, all the occurrences of $u$ are grouped consecutively in suffix array of $T$, say $SA$. Therefore, $\overline{u}$ can be represented by an interval $[s, e]$ over $SA$ where all suffixes with prefix $u$ are included and $SA[s]$ is the lexically smallest one, $SA[e]$ is the lexically greatest one [1, 10]. In this paper, we use interval to represent the states of DAWG and give an implementation of DAWG which is succinct and fast.

First, recall the definition of DAWG. For any string $u \in \Sigma^*$, let $u^{-1}S = \{x | ux \in S\}$. The syntactic congruence associated with $Suff(w)$ is denoted by $\equiv_{Suff(w)}$ [3] and is defined, for $x, y, w \in \Sigma^*$, by

$$x \equiv_{Suff(w)} y \Longleftrightarrow x^{-1}Suff(w) = y^{-1}Suff(w).$$

We call classes of factors the congruence classes of the relation $\equiv_{Suff(w)}$. Let $[u]_w$ denote the congruence class of $u \in \Sigma^*$ under $\equiv_{Suff(w)}$. The longest element in the equivalence class $[u]_w$ is called its *representative*, denoted by $rp([u]_w)$.

**Definition 4** *The* ***DAWG*** *of $w$ is a directed acyclic graph with set of states $\{[u]_w | u \in Fact(w)\}$ and set of edges $\{([u]_w, a, [ua]_w) | u, ua \in Fact(w), a \in \Sigma\}$. Denoted by $DAWG(w)$.The state $[\varepsilon]_w$ is called the* ***root*** *of $DAWG(w)$.*

The **suffix link** of a state $p$ is the state whose representative $v$ is the longest suffix of $u$ such that $v$ not $\equiv_{Suff(w)} u$. The suffix link is useful for many string applications.

In a state $r$ of $DAWG(T)$, any string is a suffix of $rp(r)$. And if a substring in $r$ ends at a position $i$ of $T$ then other substrings in $r$ also end at $i$. Therefore, the nodes and suffix links of $DAWG(T)$ form the suffix tree of reverse string of $T$ [3]. Thus a state of $DAWG(T)$, which corresponds to a node in suffix tree of $T^R$, can be represented by an interval of suffix array of $T^R$. Denote the suffix array of $T^R$ by $SA'$, for a state $r$, if $r = [s, e]$, then for any suffix of $T^R$, say $T_i^R$, $T^R_{SA'[s]} \leq T_i^R \leq T^R_{SA'[e]}$ if $rp(r)^R$ is a prefix of $T_i^R$ where $T^R_{SA'[s]}$ is the lexically smallest suffix of $T^R$ of which $rp(r)^R$ is a prefix, and $T^R_{SA'[e]}$ is the greatest one of which $rp(r)^R$ is a prefix. Fig. 1 shows such an example.

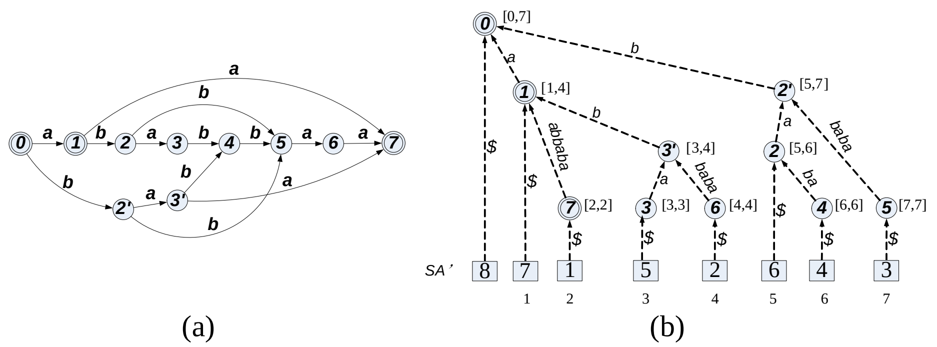

**Fig. 1.** (a) The DAWG for *ababbaa*. (b) The suffix tree for $ababbaa^R$. Each node of the tree corresponds to a DAWG state marked with same number. Each edge corresponds to a suffix link of DAWG. The interval for each node is shown on the right.

The edges of DAWG also need not stored explicitly. The state transaction, which occurs in the process of scanning an input pattern to find its occurrences in $T$ by DAWG, can be mapped to the changing of interval. For current state $s$ and input letter $a$, the state transaction is to find the state which $rp(s)a$ is in, denoted by $goto(s, a)$. For interval $[b, e]$ and input letter $a$, we need to find the interval corresponding to $goto([b, e], a)$.

We define a new succinct data structure that performs *goto* function. The data structure extends the index in [10] to the case of large alphabet. Let $T[0] = \$$ and $T[n + 1] = \$$. Define an array $\hat{T}$ of size $n + 1$ as follows:

$$\hat{T}[i+1] = \begin{cases} T[1], & \text{if} \quad i = 0, \\ T^R[SA'[i] - 1] = T[n + 2 - SA'[i]], & \text{if} \quad 1 \leq i \leq n. \end{cases}$$

Each entry of this array stores the character after each prefix of $T$ in the lexical order of reverse strings of prefixes. For example, for $T = ababbaa$, $\hat{T} =$

$ab\$baaba$. $\hat{T}$ is the result of Burrows-Wheeler transform(BWT) [4] on $T^R$. The BWT result on $T$ is first used in FM-index [6].

To deal with the situation of large alphabet, we extend operation *rank* to large alphabet. Operation $rank_\alpha(\hat{T}, i)$ returns the number of $\alpha$s in $\hat{T}$ up to position $i$, where $\alpha \in \Sigma$, $|\Sigma| \geq 2$. We also define another useful operation $rank^{\leq}$. $rank^{\leq}_\alpha(\hat{T}, i)$ returns the number of letters which are not smaller than $\alpha$ in string $T$, plus the number of $\alpha$s up to position $i$. For bit vectors, the *rank* operation can be done in constant time [11]. For arrays over alphabet $\Sigma$, we develop a method by which the *rank* and $rank^{\leq}$ operation can run in $O(\log|\Sigma|)$ time, which will be described in the next section. The following algorithm determines whether a pattern $w$ occurs in a given string $T$. It is similar to the reverse of *BW_count* algorithm of FM-index[6].

***Scan(w)***

```
1  $s \leftarrow 1; e \leftarrow n + 1$
2  for $i \leftarrow 1$ to $|w|$ do
3      $a \leftarrow w[i]$
4      $s \leftarrow rank^{\leq}_a(\hat{T}, s - 1) + 1$
5      $e \leftarrow rank^{\leq}_a(\hat{T}, e)$
6      if $s > e$ then
7          report $w$ is not a substring of $T$
8      end if
9  end for
10 report $w$ is a substring of $T$
```

This function implements the DAWG existential query. The DAWG state $[w[1..i]]_T$ corresponding to interval $[s-1, e-1]$ is computed in each step $i$ of *Scan*. Changing of interval in each step is corresponding to the state changing of DAWG existential query. In the end, all the suffixes of $T^R$ of which $w^R$ is the prefix are in $[s-1, e-1]$ of $SA'$. By this procedure, the number of occurrences of pattern can also be computed, which is $e-s+1$, the number of suffixes in interval $[s-1, e-1]$. The running time of these queries are $O(|w|\log|\Sigma|)$, because the running time of $rank^{\leq}$ is $O(\log|\Sigma|)$. The speed is not slowed down comparing to other implementations [5] of DAWG.

## 5 Implementing *Rank* and *Select* on Large Alphabet in $O(\log|\Sigma|)$ Time with $n\log|\Sigma| + o(n\log|\Sigma|)$ Bits

### 5.1 The New Index Structure

In this section, we use $E$ to refer to $\hat{T}^*$ and denote $\log|\Sigma|$ by $N$. For a bit vector $V$, denote the $i^{th}$ bit of $V$ by $V_i$, the bit segment of $V$ from $i^{th}$ bit to $j^{th}$ bit by $V[i..j]$.

Our index consists a series of bit vectors of length $n$, $E^N, E^{N-1}, \ldots, E^1$, computed from $E$. First, $E^N$ is defined as follows:

$$E^N_i = E[i]_N,\ 1 \leq i \leq n.$$

Bit vector $E^{N-1}$ is defined as follows:

$$E_i^{N-1} = \begin{cases} E[select_0(E^N, i)]_{N-1}, & \text{if} \quad 1 \le i \le rank_0(E^N) \text{ and } rank_0(E^N) \ne 0 \\ E[select_1(E^N, i)]_{N-1}, & \text{if} \quad rank_0(E^N) < i \le n \text{ and } rank_1(E^N) \ne 0. \end{cases}$$

Generally speaking, the bit vector $E^k$, $1 \le k < N$, can be constructed by the following procedures: First, order the positions in $E$ by the most significant $N-k$ bits of the integers on them. For positions on which the integers have the same first $N-k$ bits, keep their order in $E$. This procedure gives us a series of positions: $pos_1$, $pos_2$, …, $pos_n$. Second, set $E_i^k$, $1 \le i \le n$, to the $k^{th}$ significant bit of the integer on position $pos_i$ of $E$.

Precisely, $pos_1$, $pos_2$, …, $pos_n$ are divided into $2^{N-k}$ groups noted by $G_0^k$, …, $G_{2^{N-k}-1}^k$; items of $E$ on positions in $G_i^k$ have the same first $N-k$ bits, which equal to the binary representation of $i$. Accordingly, the vector $E^k$ is composed of $2^{N-k}$ non-overlapping segments $S_0^k$,…, $S_{2^{N-k}-1}^k$. The bits in $S_i^k$ are the $k^{th}$ most significant bits of items of $E$ on positions in $G_j^i$; the order of these bits is in accordance with the order of positions in $E$. Denote the start position of $S_i^k$ in $E^k$ by $F(S_i^k)$ and the end position of $S_i^k$ in $E^k$ by $L(S_i^k)$. $E^N$ has only one segment $S_\star^N = E^N$ and $F(S_\star^N) = 1, L(S_\star^N) = n$, where $\star$ denotes the empty letter. Segments of $E^k$, $1 \le k < N$, is defined recursively as follows:

For $t$ from 0 to $2^{N-k}-1$

$$S_{2t}^k = \begin{cases} E^k[F(S_t^{k+1}) \;..\; F(S_t^{k+1}) + rank_0(S_t^{k+1}) - 1], & \text{if} \quad rank_0(S_t^{k+1}) \ne \varnothing, \\ \varnothing, & \text{otherwise;} \end{cases}$$

$$S_{2t+1}^k = \begin{cases} E^k[F(S_t^{k+1}) + rank_0(S_t^{k+1}) \;..\; L(S_t^{k+1})], & \text{if} \quad rank_1(S_t^{k+1}) \ne \varnothing \\ \varnothing, & \text{otherwise.} \end{cases}$$

Let $E^{(k)}$ denote the array of length $n$, such that $E^{(k)}[i] = E[i][N..k]$, $1 \le i \le n$, where $E[i]$ is treated as a bit vector. Then $E^k$, $1 \le k < N$, is defined recursively as follows:

For $t$ from 0 to $2^{N-k}-1$

$$E_i^k = \begin{cases} E[select_{2t}(E^{(k)}, i - F(S_t^{k+1}) + 1)]_k, & \text{if } F(S_t^{k+1}) \le i \le F(S_t^{k+1}) + rank_0(S_t^{k+1}) \\ & \text{and } S_t^{k+1} \ne \varnothing, \\ E[select_{2t+1}(E^{(k)}, i - F(S_t^{k+1}) - \\ \quad rank_0(S_t^{k+1}))]_k, & \text{if } F(S_t^{k+1}) + rank_0(S_t^{k+1}) < k \le L(S_t^{k+1}) \\ & \text{and } S_t^{k+1} \ne \varnothing; \end{cases}$$

Fig. 2 gives an example of this structure. Conceptually, the bit vector $E^0$ is divided into $|\Sigma|$ segments and all the elements are set to $\star$ which denotes the empty letter. The number of elements in segment $S_a^0$ is equal to $rank_a(E)$. In practice, multi-key *rank* and *select* can be computed without $E^0$.

## 5.2 *Rank* and *Select* on Large Alphabet

We store the bit vectors $E^N$,$E^{N-1}$,…,$E^1$ in continuous memory, and take them as one bit vector, named $\mathfrak{E}$. That is $E^k = \mathfrak{E}[(k-1)n + 1 \;..\; kn]$. We build *rank* structures over $\mathfrak{E}$ using the structure of [11]. The operations on any $E^k$, say

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $i$ | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* |
| $E$ | 3 | 6 | 1 | 2 | 0 | 4 | 3 | 4 | 5 | 1 | 7 |
| $b_3$ | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| $b_2$ | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| $b_1$ | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |

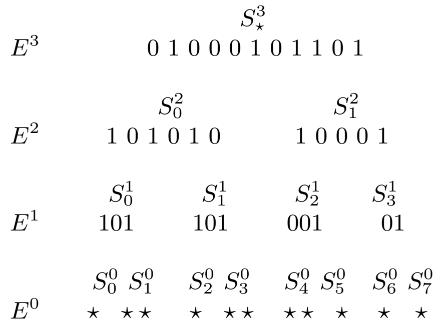
$S^3_\star$
$E^3$
0 1 0 0 0 1 0 1 1 0 1
$S^2_0$
$S^2_1$
$E^2$
1 0 1 0 1 0
1 0 0 0 1
$S^1_0$ $S^1_1$ $S^1_2$ $S^1_3$
$E^1$
101 101 001 01
$S^0_0$ $S^0_1$ $S^0_2$ $S^0_3$ $S^0_4$ $S^0_5$ $S^0_6$ $S^0_7$
$E^0$
$\star$ $\star\star$ $\star$ $\star\star$ $\star\star$ $\star$ $\star$ $\star$

**Fig. 2.** Example of the bit vectors for $E = 36120434517$. $\star$ denotes the empty letter.

$rank_\alpha(E^k, i)$, can be computed by $rank_\alpha(\mathfrak{E}, (k-1)n+i) - rank_\alpha(\mathfrak{E}, (k-1)n)$. $\mathfrak{E}$ with corresponding *rank* structure are our main indexing data structure, which together use $n\log|\Sigma| + o(n\log|\Sigma|)$ bits. The operation *select* defined on large alphabet can also be implemented using this data structure. It is the reverse computing of *rank*. $select_\alpha(E, i)$ returns the index of $i^{th}$ $\alpha$ character of $E$. The algorithm of *rank* and *select* is given below. To be clear, we do not use $\mathfrak{E}$ in explaining the algorithm but use separated bit vectors.

```
$rank_b(E, end)$
$s \leftarrow 1; e \leftarrow n; c \leftarrow end$
for $k \leftarrow \log|\Sigma|$ downto 1 do
    $c \leftarrow rank_{b_k}(E^k[s..c])$
    if $b_k = 1$ then
        $s \leftarrow s + rank_0(E^k[s..e])$
    else
        $e \leftarrow e - rank_1(E^k[s..e])$
    end if
end for
return $c$
```

```
$select_b(E, count)$
$s_{N+1} \leftarrow 1; e_{N+1} \leftarrow n; c \leftarrow count$
for $i \leftarrow \log|\Sigma|$ downto 1 do
    if $b_i = 1$ then
        $s_i \leftarrow s_{i+1} + rank_0(E^i[s_{i+1}..e_{i+1}])$
    else
        $e_i \leftarrow e_{i+1} - rank_1(E^i[s_{i+1}..e_{i+1}])$
    end if
end for
for $i \leftarrow 1$ to $\log|\Sigma|$ do
    $c \leftarrow select_{b_i}(E^i[s_i..c])$
end for
return $c$
```

In algorithm $rank_b(E, end)$, at the end of each step $k$, the start position and end position ($s$ and $e$), of segment $S^{k-1}_{b_N b_{N-1} \cdots b_k}$ are computed, where the number of symbols up to position *end*, whose first $N-k+1$ most significant bits are $b_N \cdots b_k$, say $c$, is also available. In step $k+1$, these values can be computed from the values in step $k$. Therefore in the end, the number of $b$s in $E$ can be computed. Thus the *rank* algorithm is correct.

By Lemma 1, each iteration of function *rank* runs in constant time. Therefore, by this algorithm, the number of $b$s in array $E$ of length $n$ over an arbitrary alphabet $\Sigma$ can be calculated in $O(\log|\Sigma|)$ time using $n\log|\Sigma| + o(n\log|\Sigma|)$ bits.

Thus we can find the interval corresponding to $goto(s,a)$ in $O(\log|\Sigma|)$ time. The total space for succinct DAWG is $n\log|\Sigma| + 4n + o(n\log|\Sigma|)$ bits.

Because when $rank_\alpha(E, end)$ finished, the $s-1$ equals to the number of letters which are not smaller than $\alpha$ in $E$. Therefore, by replacing line 10 of the algorithm for *rank* with the following statement, algorithm $rank^{\leq}$ is available.

**return** $s + c - 1$.

We next consider the correctness of *select* algorithm. Denote the $s$, $e$ and $c$ of each iteration in the running of the *for* loop of $rank_b(E, end)$ by $s_i$, $e_i$ and $c_i$, where $i$ is the value of loop variable and $c_{N+1} = end$. The sequence of the computing of $s_i$, $e_i$ and $c_i$ in *rank* is as follows:

$$\begin{aligned} &c_N = rank_{b_N}(E^N[s_N..e_N], c_{N+1}) \\ &c_{N-1} = rank_{b_{N-1}}(E^{N-1}[s_{N-1}..e_{N-1}], c_N) \\ &\quad\vdots \\ &c_1 = rank_{b_1}(E^1[s_1..e_1], c_2) \end{aligned}$$

Denote the value of $c$ in each *for* loop in lines 9-11 of *select* by $c'_k$ ($k$ is the loop variable) and denote the initial value of $c$ by $c'_1 = count$, then $select_b(E, count) = c'_{N+1}$. Since the first *for* loop of function $select_b(E,i)$ is to compute $rank_b(E)$, then $s'_i = s_i$ and $e'_i = e_i$. According to the *select* algorithm, the sequence of computing of $c'_i$ is as follows (we replace $s'_i$, $e'_i$ with $s_i$, $e_i$):

$$\begin{aligned} &c'_2 = select_{b_1}(E^1[s_1..e_1], c'_1) \\ &c'_3 = select_{b_2}(E^2[s_2..e_2], c'_2) \\ &\quad\vdots \\ &c'_{N+1} = select_{b_N}(E^N[s_N..e_N], c'_N) \end{aligned}$$

Immediately, we get $c'_N = c_N$ and $c_i = c'_i$ for $1 \leq i \leq N$. If $E[end] = b$, $c'_{N+1} = c_{N+1}$. Therefore $select_b(E, count) = end$ and the *select* algorithm is correct.

## 6 Conclusions

We study the properties of strings whose suffix array is a given permutation, and also give a succinct index structure for large alphabet. The core is a data structure that supports *rank* and *select* on large alphabet using $n\log|\Sigma| + o(n\log|\Sigma|)$ bits in $O(\log|\Sigma|)$ time. There are still many interesting issues, such as bidirectional index based on this structure, the relation between strings with inverse suffix arrays. There are works to be done to reveal these structures.

## References

1. M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE02), LNCS 2476, pages 31-43. Springer-Verlag, 2002.

2. Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In Proc. 28th International Symposium on Mathematical Foundations of Computer Science (MFCS 2003), LNCS 2747, pages 208-217. Springer Verlag, 2003.
3. A.Blumer, J.Blumer, D.Haussler, A.Ehrenfeucht, M.T.Chen, and J.Seiferas. The smallest automation recognizing the subwords of a text. Theoretical Computer Science, 40:31-55, 1985.
4. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. DEC SRC Research Report 124, 1994.
5. M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2, Linear Modeling: Background and Application, chapter 9, pages 399-462. Springer-Verlag, 1997.
6. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In Proceedings of the 4lst Annual IEEE Symposium on Foundations of Computer Science (FOCS), pages 390-398, 2000.
7. G. Gonnet, R. Baeza-Yates, T. Snider, New indices for text: PAT trees and PAT arrays, in: W. Frakes, R.A. Baeza-Yates (Eds.), Information Retrieval: Algorithms and Data Structures, Prentice-Hall, Englewood Cliffs, NJ, 1992, pp. 66-82.
8. D. Gusfield. Algorithms on Strings Trees and Sequences. Cambridge University-Press, New York, New York, 1997.
9. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC), 2000.
10. Meng He, J. Ian Munro, S. Srinivasa Rao. A categorization theorem on suffix arrays with applications to space efficient text indexes In SIAM Symposium on Discrete Algorithms (SODA), 2005. Pages: 23-32.
11. G. Jacobson. Succinct static data structures. Technical Report CMU-CS-89-112, Dept. of Computer Science, Carnegie-Mellon University, Jan. 1989.
12. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing, 22:935–948, Oct 1993.
13. J.I. Munro and V. Raman, Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs, Proc. 38th Annual IEEE Symp. on Foundations of Computer Science, October 1997, pages 118-126.
14. J. I. Munro. Tables. In Proceedings of the 16th ray Conference on Foundations of Software Technology and Computer Science (FSTTCS '96), LNCS 1180, pages 37–42, 1996.
15. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix arrays. In Proc. 11th International Symposium on Algorithms and Computation, pages 410-421. Springer-Verlag LNCS 1969, 2000.
16. P. Weiner. Linear pattern matching algorithm. In Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, pages 1-11, 1973.
17. Meng Zhang. Succinct Text Indexes on Large Alphabet. Technical Report, Jilin University, 2005.