Practical Entropy-Compressed Rank/Select Dictionary

Daisuke Okanohara∗ Kunihiko Sadakane†

Abstract

Rank/Select dictionaries are data structures for an ordered set \( S \subset \{0, 1, \ldots, n - 1\} \) to compute \( \text{rank}(x, S) \) (the number of elements in \( S \) which are no greater than \( x \)), and \( \text{select}(i, S) \) (the \( i \)-th smallest element in \( S \)), which are the fundamental components of succinct data structures of strings, trees, graphs, etc. In those data structures, however, only asymptotic behavior has been considered and their performance for real data is not satisfactory. In this paper, we propose novel four Rank/Select dictionaries, esp, reocrank, vcode and sdarray, each of which is small if the number of elements in \( S \) is small, and indeed close to \( nH_0(S) \) (\( H_0(S) \leq 1 \) is the zero-th order empirical entropy of \( S \)) in practice, and its query time is superior to the previous ones. Experimental results reveal the characteristics of our data structures and also show that these data structures are superior to existing implementations in both size and query time.

1 Introduction

Rank/Select dictionaries are data structures for an ordered set \( S \subset \{0, 1, \ldots, n - 1\} \) to support the following queries:

- \( \text{rank}(x, S) \): the number of elements in \( S \) which are no greater than \( x \),
- \( \text{select}(i, S) \): the position of \( i \)-th smallest element in \( S \).

These data structures are used in succinct representations of several data structures. A succinct representation is a method to represent an object from an universe with cardinality \( L \) by \((1 + o(1))\log L\) bits\(^1\). While this idea is very similar to the idea of data compression, the difference is that succinct representations support fast queries on the object such as enumerations or navigations. Various succinct representation techniques have been developed to represent data structures such as ordered sets \([25, 19, 20, 21]\), ordinal trees \([11, 26, 5, 13, 18, 23]\), strings \([11, 9, 10, 22, 24, 24]\), functions \([17]\), and labeled trees \([11, 3]\). All these data structures are based on a succinct representation of Rank/Select dictionaries.

Many data structures have been proposed for Rank/Select dictionaries, most of which support the queries in constant time on word RAM \([7, 18, 16, 19, 21]\) using \( n + o(n) \) bits or \( nH_0(S) + o(n) \) bits (\( H_0(S) \leq 1 \) is the zero-th order empirical entropy of \( S \)). In most of these data structures, however, their asymptotic behavior is only considered, and their performance is not optimal for real-size data. As a result, the query time is slow and the data structure size is large for real data. Although recently some practical implementation of Rank/Select dictionaries have been proposed

\( \ast \)Department of Computer Science, University of Tokyo. Hongo 7-3-1, Bunkyo-ku, Tokyo 113-0013, Japan. (hillbig@is.s.u-tokyo.ac.jp).

\( \dagger \)Department of Computer Science and Communication Engineering, Kyushu University. Motooka 744, Nishi-ku, Fukuoka 819-0395, Japan. (sada@csce.kyushu-u.ac.jp). Work supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

\(^1\)Let \( \log n \) denote \( \log_2 n \).
using \( n + o(n) \) bits \[8\] \[14\], there is no practical implementation of those using \( nH_0(S) + o(n) \) bits. Recently gap-based compressed dictionaries have been proposed \[11\] \[12\]. They use another measure called \( \text{gap}(S) := \sum_{i=1}^{n} \lfloor \log(\text{select}(i+1, S) - \text{select}(i, S)) \rfloor \) to define the minimum space to store \( S \) and propose the data structure using \( \text{gap} + O(m \log(n/m)/\log m) + O(n \log \log m/n) \) bits, which is much smaller than the entropy-based ones if \( m \ll n \), but it cannot not support constant time rank and select queries because of the lower bound \[15\] \[7\].

We will introduce novel four Rank/Select dictionaries, esp, re crank, vcode and sdarray(sarray and darray), each of which is based on different ideas and thus has different advantage and disadvantage in terms of speed, size and simpleness. These sizes are small if the number of elements in \( S \) is small, and even close to the zero-th order empirical entropy of \( S \). \( H_0(S) \leq 1 \), which is defined as \( nH_0(S) = m \log \frac{n}{m} + (n - m) \log \frac{n}{n - m} \) where \( m \) is the number of elements in \( S \).

Table 1 summarizes the properties of proposed data structures for an ordered set \( S \subset \{0, 1, \ldots, n-1\} \) with \( m \) elements in terms of size, time for rank and select. We note that these bounds are in the worst case and we can expect faster in practice. For example, the \( O(\log^4 m/\log n) \) term in sarray and darray and \( O(\log n) \) term in vcode are \( O(1) \) in almost the case.

| method     | size (bits)                   | rank               | select               |
|------------|-------------------------------|--------------------|----------------------|
| esp (Sec. \[3\]) | \( nH_0(S) + o(n) \)         | \( O(1) \)        | \( O(1) \)          |
| re crank (Sec. \[4\]) | \( 1.44m \log \frac{n}{m} + m + o(n) \) | \( O(\log \frac{n}{m}) \) | \( O(\log \frac{n}{m}) \) |
| vcode (Sec. \[5\]) | \( m \log(n/\log^2 n) + o(n) \) | \( O(\log^2 n) \) | \( O(\log n) \)     |
| sarray (Sec. \[6\]) | \( m \log \frac{n}{m} + 2m + o(m) \) | \( O(\log \frac{n}{m}) + O(\log^4 m/\log n) \) | \( O(\log^4 m/\log n) \) |
| darray (Sec. \[6\]) | \( n + o(n) \)                | \( O(1) \)        | \( O(\log^4 m/\log n) \) |

We conducted experiments using proposed methods and previous methods and show that our data structures are fast and small compared to the previous ones.

2 Preliminaries

In this paper we assume the word RAM model. Under the word RAM model we can perform logical and arithmetic operations for two \( O(\log n) \)-bit integers in constant time, and we can also read/write consecutive \( O(\log n) \) bits of memory for any address in constant time.

An ordered set \( S \), which is a subset of the universe \( U = \{0, 1, \ldots, n - 1\} \), can be represented by a bit-vector \( B[0, \ldots, n - 1] \) such that \( B[i] = 1 \) if \( i \in S \) and \( B[i] = 0 \) otherwise. We denote \( m \) as the number of ones in \( B \). Then \( \text{rank}(x, S) \) is the number of ones in \( B[0, x] \), and \( \text{select}(i, S) \) is the position of the \( i \)-th one from the left in \( B \). These values are computed in constant time on word RAM using \( O(n \log \log n / \log n) \)-bit auxiliary data structures \[16\].

The above representation of \( S \) using the bit vector of length \( n \)-bit is the worst-case optimal because there exist \( 2^n \) different sets in the universe and we need \( \log 2^n = n \) bits to distinguish different subsets. We call this representation verbatim representation. Similarly, a lower-bound of the size of the representation of \( S \) with \( m \) elements is \( \mathcal{B}(n, m) = \lceil \log \binom{n}{m} \rceil \) bits. This value is approximately \( nH_0(B) \), which is further approximated by \( H_0(B) \leq m \log \frac{n}{m} + 1.44m \) bits . Therefore the size of the verbatim representation is far from this lower-bound if \( m \ll n \). Raman et al. \[21\] proposed a constant-time Rank/Select data structure whose size is \( \mathcal{B}(n, m) + O(n \log \log n / \log n) \), which matches the above lower-bound asymptotically.
The applications of Rank/Select dictionaries can be divided into two groups. One is for sets with \( m \simeq n/2 \) and the other is for sets with \( m \ll n \). In this paper we call the former dense sets and the latter sparse sets. Typical applications of dense sets are for the wavelet trees \([9]\) that are used for indexing strings, and for ordinal trees. On the other hand sparse sets are used in many succinct data structures in order to compress pointers to blocks, each of which stores a part of the data. Because in the word RAM model any consecutive \( O(\log n) \) bits of data are accessed in constant time, we often divide the data into blocks of \( \Theta(\log n) \) bits each. For example, an ordinal tree with \( n \) nodes is encoded in a bit-vector of length \( 2n \), and to support tree navigating operations, the bit-vector is divided into block of length \( \frac{1}{2} \lg n \) bits and in each block we logically mark one bit to construct a contracted tree with \( O(n/\log n) \) nodes. These logical marks are represented by a bit-vector of length \( 2n \) in which \( 4n/\lg n \) bits are one. The ratio of one is \( 2/\lg n \), that is, the vector is sparse. Such vectors can be encoded in \( B(2n, 4n/\log n) + O(n \log \log n/\log n) = O(n \log \log n/\log n) = o(n) \) bits. Therefore for storing a sparse vector in a compressed form is important for succinct data structures.

In this paper we will mainly focus on sparse sets to support rank and select functions. Although in some applications like wavelet trees we also need a select function\(^2\), we usually assume dense sets in such applications and well-developed Rank/Select dictionaries for dense sets can be applied.

### 2.1 Previous Implementation of Rank/Select Dictionaries

We first give a brief description of Rank/Select dictionary using \( n + o(n) \) bits, which is called verbative. We conceptually partition \( B \) into subsequences of length \( l := \log^2 n \) each, called large block. Then each large block is partitioned into subsequences of length \( s := \log n/2 \) each, called small block. For the boundaries of large blocks we store rank-directory (results of rank) in \( R_l[0 \ldots n/l] \) explicitly using \( O(n/\log^2 n \cdot \log n) = O(n/\log n) \) bits. We also store rank-directory for each boundary of small blocks in \( R_s[0 \ldots n/s] \), but here we store only relative values to ones stored for the large blocks, which are stored in \( O(n \log \log n/\log n) \) bits.

Then rank is computed by
\[
\text{rank}(x, S) = R_l[\lfloor x/l \rfloor] + R_s[\lfloor x/s \rfloor] + \text{popcount}(\lfloor x/s \rfloor \cdot s, x \mod s),
\]
where \( \text{popcount}(i, j) \) is the number of ones between \( B[i \ldots i+j] \) which can be calculated in constant time using a pre-computed table of size \( O(\sqrt{n} \log^2 n) \) bits or the popcount function \([5]\). For select we have two options; the first is a constant time solution using \( o(n) \) auxiliary data structures \([14]\) and the second is a \( O(\log n) \) solution which is a binary search using rank functions without any auxiliary data structures \([5]\). Because of the luck of space we omit the detail of select in constant time \([14]\).

We next introduce Rank/Select dictionary using \( nH_0(S) + o(n) \) bits, which is called ent. The main difference between verbative and ent is the representation of bit-vector itself, that is each small block is encoded by the enumerative code \([2]\) as follows. Given \( t \), the length of the block, and \( u \), the number of ones in the block, we calculate \( \sum_{i=1}^{u} \left( t - p_{i-1} \right) \) where \( p_u \) is the position of \( i \)-th one in the block. This value is the unique number in \( [0, \lfloor \log \binom{t}{u} \rfloor - 1] \) for each possible block of \( t \) length with \( u \) ones. This number can be represented by \( B(t, u) = \lfloor \log \binom{t}{u} \rfloor \) bits and the size of all encoded blocks is less than \( B(n, m) \leq nH_0(S) \)[19]. We represent each small block as the result of enumerative code, and the total size is less than \( nH_0(S) \). Since they have different sizes, we also need to store pointers to compressed small blocks, which is \( O(n \log \log n/\log n) = o(n) \) bits. These encoding and decoding are performed by using pre-computed table of \( O(\sqrt{n} \log^2 n) \)-bits.

We note that although the size of ent is \( nH_0(S) + o(n) \) bits, we cannot ignore the \( o(n) \) term

\(^2\)We do not discuss rank since it can be computed by rank as \( \text{rank}(i, S) = i + 1 - \text{rank}(i, S) \).

\(^3\)In this paper let \( a \mod b \) denote \( a - \lfloor a/b \rfloor \).
because \( nH_0(S) \) term is small compared to \( n \) if \( m \ll n \) and \( o(n) \) is as much as \( \Theta(nH_0(S)) \).

### 3 Estimating Pointer Information

We first propose \( \text{esp} \) (stands for EStimating Pointer information), which does not require \textit{pointer information} by estimating them from \textit{rank} information. Although the size of \textit{pointer information} is \( O(n \log \log n / \log n) = o(n) \), this size is actually large as much as \( \Theta(nH_0(S)) \) terms for real-size data.

First we show the propositions which are needed to bound the size of compressed bit vector in terms of \textit{rank} information. Given a-bit vector \( B[0 \ldots n-1] \) with \( m \) ones, let \( L(B) \) be the length of code word for \( B \) using enumerative code [2] (See Section 2.1). Then,

**Proposition 1** \( L(B) \leq H_0(B) \).

Because \( H_0(B) \) is the size of a representation of block that uses \( \log(n/m) \) bits for each 1’s and \( \log(n/(n-m)) \) bits for each 0’s, and the \( L(B) = B(n, m) := \lceil \log \binom{n}{m} \rceil \) is the smallest length of the code to represent the bit vector.

Let \( B_i \) \((i = 1 \ldots \lceil \frac{n}{s} \rceil)\) be the partition of \( B \), and \( u \) be the size of each block. Then,

**Proposition 2** \( \sum_{i=1}^{\lceil \frac{n}{s} \rceil} L(B_i) \leq \sum_{i=1}^{\lceil \frac{n}{s} \rceil} uH_0(B_i) \leq nH_0(B) \).

The second inequality holds because \( nH_0(B) \) is the concave function.

Let \( B' := B[0 \ldots t] \) \((t \leq n)\) be the prefix of bit-vector \( B \). Since \( L(B') \leq H_0(B') \) (use Prop. [1] and Prop. [2]), we can store all code words of \( B' \) within \( H_0(B') \) bits. However since the inequality not equality holds we still have an estimation error of pointers. We therefore need to insert gap bits so that we always estimate the correct pointer information.

We will explain the details of \( \text{esp} \). Basically, \( \text{esp} \) is based on \textit{ent} except the existence of super-large blocks (SLB) since we need to reset estimation errors in each SLB. We conceptually partition \( B \) into subsequences of length \( k := \log^3 n \) each, called \textit{super large block} (SLB). Then each SLB is partitioned into \textit{large block} (LB) of length \( l := \log^2 n \). Then each LB is partitioned again into \textit{small block} (SB) of length \( s := \log n / 2 \). We then encode each SB by enumerative code (Section 2.1) independently. The code word for \( i \)-th SB: \( SB_i \) is stored in the position which is determined as follows. Let \( l_r \) and \( s_r \) be results of \textit{rank} for LB and SB as \( l_r = R_l[x_l] \), and \( s_r = R_s[x_s] \) where \( x_l = \lfloor x/l \rfloor \) and \( x_s = \lfloor x/s \rfloor \). Then we estimate the starting positions of LB and SB as

\[
lp = H_0(LB_{x_l}) = l_r \cdot \frac{l \cdot x_l}{l_r} + (l \cdot x_l - l_r) \cdot \frac{l \cdot x_l}{l \cdot x_l - l_r} \tag{1}
\]

\[
sp = H_0(SB_{x_s}) = s_r \cdot \frac{s \cdot x_s}{s_r} + (s \cdot x_s - s_r) \cdot \frac{s \cdot x_s}{s \cdot x_s - s_r} \tag{2}
\]

where \( LB_{x_l} \) denotes the preceding LBs from the boundary of SLB up to \( LB_i \) and \( SB_{x_s} \) denotes the preceding SBs from the boundary of LB up to \( SB_i \). Then the position for compressed \( SB_i \) is \( slp + lp + sp \) where \( slp \) is the pointer information of SLB which is stored explicitly. We note that all code words are not overlapped (use Prop. [2]) and gap-bits are automatically inserted.

We store rank-directory for LB, SB and \textit{pointer information} for SLB. All of them are stored in \( o(n) \) bits.

For \textit{rank}(\( x, S \)), we lookup correspondent rank-directory for LB, SB as \( l_r = R_l[x_l] \), and \( s_r = R_s[x_s] \) where \( x_l = \lfloor x/l \rfloor \) and \( x_s = \lfloor x/s \rfloor \). Then we estimate the \textit{pointer information} for LB and SB as in \( \text{1} \) and \( \text{2} \). We then read the compressed bit representation of SB from that position and decode it in constant time and do \textit{popcount} as in \textit{verbative}. 


For select, we use the same approach as in [14] which is done in constant time with the $o(n)$-bits auxiliary data structures.

In practice, since it is very slow to compute the logarithm of a floating-point number for the estimating the entropy, we use a pre-computed table lookup and also use fixed-point integer representation. We require two integer multipliers and one integer addition for estimating one value of the entropy.

4 RecRank

The second data structure re crank uses the reduction of a sparse bit-array into a contracted bit-array and a denser extracted bit-array which was originally used for Algorithm I in [14]. Here we use the reduction recursively.

Given a bit-arrays $B[0\ldots n-1]$ with $m$ ones, we conceptually partition $B$ into the blocks $B_0,\ldots,B_{n/t}$ of length $t$. We call zero block (ZB) a block where all elements are 0 and non-zero block (NZ) a block where there is at least one 1. The contracted bit-array of $B$, $B_c[0\ldots,n/t-1]$ is defined as a bit-string such that $B_c[i] = 0$ if $B_i$ is ZB, and $B_c[i] = 1$ if $B_i$ is NZ, and the extracted bit-array $B_e$ is defined as a bit-array which is formed by concatenating all NZ blocks of $B$ in order.

We can calculate rank of $B$ using $B_c$ and $B_e$ as

$$
\text{rank}(x, B) = \text{rank}(\text{rank}([x/t], B_c) \cdot t + (x \mod t) \cdot B_c[\lfloor x/t \rfloor], B_e).
$$

We then recursively apply this reduction by considering the extracted bit array as a new input bit array. We continue this process until the extracted bit-array is dense enough (the probability of one in a bit-array is larger than 1/4). After $u$ times of the reduction, we have $t$ contracted bit arrays $B_{c1}, B_{c2},\ldots,B_{cu}$ and the final extracted bit array $B_e$.

Here we take the strategy that contracted bit arrays would be dense (the probability of ones in the bit array would be 1/2). Let $p(B) = m/n$ be the probability of ones in the bit array $B$. We choose the block size $t = \frac{1}{-\log(1-p)}$ so that the $p(B_c)$ would be 0.5. This is because the probability of $t$ bits being all zero is $(1-p)^t$ and the half of the elements in contracted bit array is one when $(1-p)^t = 1/2$. Then the length of $B_c$ is $-n \log(1-p)$ and the length of $B_e$ is $n/2$. We note that $B_e$ contains $m$ ones and $p(B_e) = 2p$. This reduction is applied $u = -\log p$ times so that the probability of ones in the final extracted bit array is larger than 1/4.

Let $T$ be the size of re crank and $p = 2^{-u}$. We can calculate $T$ as follows,

$$
T = n \cdot \sum_{i=0\ldots u-2} \left( -\frac{\log(1-2^i p)}{2^i} \right) + 2m
$$

$$
\leq n \cdot \frac{1}{\log_e(2)} \sum_{i=0\ldots u-2} \left( \frac{2^i p + 2 \cdot (2^i p)^2 / 3}{2^i} \right) + 2m
$$

$$
= \frac{1}{\log_e(2)} (-m \log p - 2m/3 - 2mp/3) + 2m
$$

In [5], we use $\log(1-x) \leq x + \frac{2}{3} x^2$ for $0 \leq x \leq \frac{1}{4}$. In short, $T$ is bounded by $1.44m \log n/m + m$ bits.

For rank, we apply [3] at each stage. Since the number of reduction is $-\log p = \log n/m$ and each stage is done in constant time, the total time is $O(\log n/m)$. For select, we apply select in each stage, each of which is done in constant time [14], so the total time is in $O(\log n/m)$. 

5
The idea of SDarrays (sdarray) is to use two different techniques for sparse sets and dense sets each, which enables us to design the data structure simply. We call the former sarray and the latter darray (sarray uses darray as a part of data structure).
int rank_sarray(int i) { // return rank(i,B) in sarray
    int y = select_0(i/2^w,H)+1; int x = y-i/2^w;
    for (int j = i%2^w; H[y] == 1; x++,y++)
        if (L[x] >= j) { //L is lower-bit of B
            if (L[x] = j) x++;
            break;
        }
    return x;
}

Figure 2: An example code of rank in sarray. Variables correspond to the definition in the paper.

First we will introduce sarray for sparse sets. Given a bit-arrays B[0...n-1] with m ones (m \ll n), we define x[0...m-1] such that x[i] = select(i+1,B). Each x is then divided into upper z = [lg m] bits and lower w = [lg n/m] bits. Lower bits are stored explicitly in L[0...m-1] using m \cdot [lg n/m] bits. Upper bits are represented by a bit array H[0...2m-1] such that H[x[i/2^w + i] = 1 and others are 0. By using H and L, we can calculate select in sarray by select(i,B) = (select(i,H) - i) \cdot 2^w + L[i]. We need select for H. Here we can assume that H is dense because there are m ones and m zeros in H.

We then explain darray for dense sets, B[0...n-1] with m \approx n/2 ones4. We first partition H into the blocks such that each block contains L ones respectively. Let P[i][0...n/L-1] be the bit arrays such that P[i][j] is the position of (iL+1)-th one. We classify these blocks into two groups. If the length of block size (P[i][j] - P[i-1][j]) is larger than L_2, we store all the positions of ones explicitly in S_i. If the length of block size is smaller than L_2, we store the each L_3-th positions of ones in S_s. We can store these values in lg L_2 bits.

For select(i,B) in darray, we lookup P[i][i/L] and see whether the block is larger than L_2 or not. If it is, we lookup the value in S_i which is stored explicitly. If not, we lookup correspondent L_3-th value in S_s and then do sequential search in the block which would take O(L_2/log n) time because we can read O(log n) bits in RAM model. We note that if we can assume that ones is distributed in B uniformly, this sequential search is done in O(1) time. Although this data structure concerns only select, we can use same data for select_0 by reversing bits in H at reading time.

For rank in darray, we use the same method as in verbatime. For rank(i,B) in sarray(see the example code in figure 2), we first calculate y = select_0(i/2^w,H) + 1 to find the smallest element which is greater than [i/2^w] \cdot 2^w. Then we count the number of elements which equals to or smaller than i by sequentially searching over H and L in time O(n/m) because the possible bit pattern of length lg n/m is n/m. If we use binary search, we can do it in O(log n/m) time but this is slower than sequential search in practice and we use sequential search.

The size of P_i is O(n^2/2 \cdot log n), that of S_i is at most n \cdot L \cdot lg n bits, and that of S_s is at most \frac{n}{2^w} \cdot lg L_2 bits. When we choose L := O(log^2 n), L_2 := O(log^4 n), and L_3 := O(lg n), all the sizes of P_i and S_i and S_s are o(n). In summary, the size of darray is n + o(n) bits.

We then analyze the size of sarray. We use m \cdot [lg n/m] bits for L. For H of length 2m, we use the data structure of darray, which is m + o(m) bits. Therefore the total size of sarray is m \cdot [lg n/m] + 2m + o(m) bits.

4The size of H in sarray is 2m not n. Here we explain darray in general case.
7 Experimental Results

We conducted experiments using esp (esp), recrank (rr), vcode (vc), sarray (sa) and darray (da). We also compare them with byte-based implementation in [14] (Kim), and its re-implementation by us (Kim2) and [8] (Navarro). For esp, we used $k = 2^{12}$, $l = 2^{8}$, $s = 2^{5}$. 
Table 2: The space results for esp, re crank, v code, sarray and Navarro for the bit arrays of n-bit length with 1% and 5% ones. The values is the percentage of the size of each data structures over an original bit-array.

| Ratio of 1’s | esp   | re crank | v code | sarray | nH_0  | Navarro |
|-------------|-------|----------|--------|--------|-------|---------|
| 1%          | 17.02 | 15.83    | 15.05  | 10.13  | 8.08  | 137.5   |
| 5%          | 42.67 | 49.32    | 62.25  | 40.59  | 28.64 | 137.5   |

For vc, we used p = 8. For sa, we used L = 2^{10}, L_2 = 2^{16} and L_3 = 2^5.

For select in rr, we used O(log n) solutions because o(n) auxiliary data would become large. For rank and select in sa and da, we used sequential search in H because it is faster in practice.

We used GNC C 3.4.3 -O6 -m64. We measured time using ftime functions on the 3.4GHz Xeon with 8GB main memory.

All experiments are done using the bit arrays of length 10M(10 \cdot 2^{20}) bits.

Figure 3 shows the result of the size of several data structure. We also show the result of nH_0(B) which is the lower bound of data structure if we only know the ratio of ones. From here we can see that the size of esp is very close to nH_0(B) in all conditions. We also find that the size of rec, sa and vc are very close to nH_0(B) when the ratio of 1 is very small.

Table 2 shows the sizes of each data structures for the bit-arrays with 1% and 5% ones. We find that the sizes of proposed data structures are indeed close to nH_0. We note that sarray is the smallest in both case.

Figure 4 is the result of 10^8 rank operations. We can see that Kim2, Navarro and da is the fastest which is the same as in rank in verbative. On the other hand vc is the slowest for rank because it needs binary search using select functions. Only rec is slower in the small ratio of 1 because its computation cost is O(log n/m) depending on the inverse number of m.

Figure 5 is the result of 10^8 select operations. Among several methods, sa is the fastest in all
conditions. As in the result of \textbf{rank}, \textit{rec} is slower in the small ratio of 1. We also find that \textit{da} shows different behavior in the small ratio of 1 because it switches data structures depends on the ratio of 1. We note that the result of \textit{esp} for \textbf{rank} and \textbf{select} is fast in the ratio of 1 is small or large since \textit{esp} employ decode table for \textit{enumerative code} which is only prepared for compressible block. Therefore it becomes slower when the block could not be compressible.

We did not show the results in bit arrays of different length because of the lack of space. We note that except \textit{Navarro}, \textit{rec} and \textit{vc} which use binary search in \textbf{select}, all methods have the similar result with bit arrays of different length.

8 Concluding Remarks

In this paper, we have proposed novel four Rank/Select dictionaries, \textit{esp}, \textit{recrank}, \textit{vcode} and \textit{sdarray}. Experimental results show that the sizes of these data structures are indeed close to the zero-th order \textit{empirical entropy} and achieves fast queries.

We also note that they are easy to implement (except \textit{esp}) because \textit{recrank} uses reduction which can employ well-developed \textit{dense sets} techniques and \textit{vcode} converts the problem into the \textit{popcount} in bytes and \textit{sdarray} separates the problem for \textit{dense sets} and \textit{sparse sets}, which simplify the problem.

In the next stage of our research, we will extend our result to more complex data structures, such as sequences from large alphabets. We also consider applications which employ appropriate data structures and also apply them to data compression as well.

References

[1] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. Algorithmica, 43(4):275–292, 2005.
[2] T. Cover. Enumerative source encoding. IEEE Trans. on Information, 19(1):73–77, 1973.
[3] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In FOCS, 2005.
[4] P. Ferragina and G. manzini. Indexing compressed texts. Journal of the ACM, 52(4):552–581, 2005.
[5] R. Geary., N. Rahman., R. Raman., and V. Raman. A simple optimal representaiong for balanced parentheses. In Proc. of CPM, pages 159–172, 2004.
[6] R. Geary., N. Rahman., and V. Raman. Succinct ordinal trees with level-ancestor queries. In ACM-SIAM SODA, pages 1–10, 2004.
[7] A. Golyinski. Optimal lower bounds for rank and select indexes. In Proc. of ICALP, 2006.
[8] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA '05), pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.
[9] R. Grossi., A. Gupta., and J. Vitter. High-order entropy-compressed text indexes. In Proc. of SODA, pages 841–850, 2003.
[10] R. Grossi., A. Gupta., and J. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In Proc. of SODA, pages 636–645, 2004.
[11] A. Gupta, W. Hon, R. Shar, and J. Vitter. Compressed data structures: Dictionaries and data-aware measures. In Proc. of DCC, pages 213–222. IEEE, 2006.
[12] A. Gupta, W. Hon, R. Shar, and J. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In Proc. of WEA, 2006. To appear.
[13] G. Jacobson. Space-efficient static trees and graphs. In Proc. of FOCS, pages 549–554, 1989.
[14] D. K. Kim., J.C. Na., J.E. Kim., and K. Park. Efficient implementation of rank and select functions for succinct representation. In *Proc. of WEA*, 2005.

[15] P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. of SODA*, pages 11–12, 2005.

[16] J. I. Munro. Tables. In *Proc. of FSTTCS*, pages 37–42, 1996.

[17] J. I. Munro and S. S. Rao. Succinct representations of functions. In *Proc. of ICALP*, pages 1006–1015, 2004.

[18] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[19] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Computation*, 31(2):353–363, 2001.

[20] C. K. Poon and W. K. Yiu. Opportunistic data structures for range queries. In *Proc. of COCOON*, pages 560–569, 2005.

[21] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of SODA*, pages 232–242, 2002.

[22] S. S. Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, 2002.

[23] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *ACM-SIAM SODA*, pages 225–232, 2002.

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

[25] K. Sadakane W. K. Hon and W.K. Sung. Succinct data structures for searchable partial sums. In *Proc. of ISAAC*, pages 505–516, 2003.

[26] C. C Lin Y. T. Chiang and H. I. Lu. Orderly spanning trees with applications. *SIAM Journal on Computing*, 34(4):924–945, 2005.