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 $\mathbf{rank}(x, S)$ (the number of elements in S which are no greater than x), and $\mathbf{select}(i, S)$ (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 four novel Rank/Select dictionaries, **esp**, **recrank**, **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:

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

These data structures are used in succinct representations for several data structures. A succinct representation is a method to represent an object from an universe with cardinality L by $(1 + \mathrm{o}(1)) \lg L$ 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 [1, 26, 5, 6, 13, 18, 23], strings [4, 9, 10, 22, 23, 24], functions [17], and labeled trees [1, 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, 13, 16, 19, 21] using $n + \mathrm{o}(n)$ bits or $nH_0(S) + \mathrm{o}(n)$ 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 large and the data structure size is large for real data. Although recently some practical implementations of Rank/Select dictionaries have been proposed using $n + \mathrm{o}(n)$ bits [8, 14], there is no practical implementation of those using $nH_0(S) + \mathrm{o}(n)$ bits. Recently *gap-based* compressed dictionaries have been proposed [11, 12]. They use another measure called $gap(S) = \sum_{i=1\ldots m} \lceil \lg (\mathbf{select}(i+1, S) - \mathbf{select}(i, S)) \rceil$ to define the minimum space to store S and propose the data structure using $gap + \mathrm{O}(m \log(n/m)/\log m) + \mathrm{O}(n \log\log m/n)$ bits where m is the number of elements in S. This measure would become much smaller than the entropy-based ones when adjacent elements in S have similar values, but it cannot support constant time rank and select queries.

We will introduce four novel Rank/Select dictionaries, **esp**, **recrank**, **vcode** and **sdarray**(**sarray** and **darray**), each of which is based on different ideas and thus has different advantages and disadvantages in terms of speed, size and simplicity. 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 $H_0(S) = \frac{m}{n} \lg \frac{n}{m} + \frac{n-m}{n} \lg \frac{n}{n-m}$ where m is the number of elements in S.

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

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

†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 $\lg n$ denote $\log_2 n$

Table 1: The space and time results for **esp**, **recrank**, **vcode**, **sarray** and **darray** for an ordered set $S \subset \{0, 1, \ldots, n-1\}$ with m elements. $H_0(S) \leq 1$ is the zero-th order *empirical entropy* of S.

method	size (bits)	**rank**	**select**
esp (Sec. 3)	$nH_0(S) + \mathrm{o}(n)$	$\mathrm{O}(1)$	$\mathrm{O}(1)$
recrank (Sec. 4)	$1.44m \lg \frac{n}{m} + m + \mathrm{o}(n)$	$\mathrm{O}(\log \frac{n}{m})$	$\mathrm{O}(\log \frac{n}{m})$
vcode (Sec. 5)	$m \lg(n/\lg n) + \mathrm{O}(m)$	$\mathrm{O}(\log^2 n)$	$\mathrm{O}(\log n)$
sarray (Sec. 6)	$m \lg \frac{n}{m} + 2m + \mathrm{o}(m)$	$\mathrm{O}(\log \frac{n}{m}) + \mathrm{O}(\log^4 m/\log n)$	$\mathrm{O}(\log^4 m/\log n)$
darray (Sec. 6)	$n + \mathrm{o}(n)$	$\mathrm{O}(1)$	$\mathrm{O}(\log^4 m/\log n)$

$\mathrm{O}(1)$ in almost cases.

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 $\mathrm{O}(\log n)$-bit integers in constant time, and we can also read/write consecutive $\mathrm{O}(\log n)$ 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 $\mathbf{rank}(x, S)$ is the number of ones in $B[0, x]$, and $\mathbf{select}(i, S)$ is the position of the i-th one from the left in B. These values are computed in constant time on word RAM using $\mathrm{O}(n \log\log n/\log n)$-bit auxiliary data structures [16].

The above representation of S using the bit vector of length n is worst-case optimal because there exist 2^n different sets in the universe and we need $\lg 2^n = n$ bits to distinguish different subsets. We call this representation *verbatim representation*. A lower-bound of the size of the representation of S with m elements is $\mathcal{B}(n, m) = \lceil \lg \binom{n}{m} \rceil$ bits. This value is approximately $nH_0(B)$, which is further approximated by $H_0(B) \simeq m \lg \frac{n}{m} + 1.44m$ bits if $m \ll n$. 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 $\mathcal{B}(n, m) + \mathrm{O}(n \log\log n/\log n)$, which matches the above lower-bound and the recent lower bounds [15, 7] 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 $\mathrm{O}(\log n)$ 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 blocks of length $\frac{1}{2} \lg n$ bits and in each block we logically mark one bit to construct a contracted tree with $\mathrm{O}(n/\log n)$ nodes. These logical marks are represented by a bit-vector of length $2n$ in which $4n/\lg n$ bits are 1. The ratio of ones is $2/\lg n$, that is, the vector is sparse. Such vectors can be encoded in $\mathcal{B}(2n, 4n/\log n) + \mathrm{O}(n \log\log n/\log n) = \mathrm{O}(n \log\log n/\log n) = \mathrm{o}(n)$ 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. In some applications like wavelet trees we also need a $\mathbf{select}_0$ function defined as $\mathbf{select}_0(i, S)$: the i-th smallest element not in S[2]. We did not discuss $\mathbf{select}_0$ in this paper because we usually assume *dense sets* in applications using $\mathbf{select}_0$.

2.1 Previous Implementation of Rank/Select Dictionaries

We first give a brief description of Rank/Select dictionary using $n + \mathrm{o}(n)$ bits, which is called **verbatim**. 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 blocks*. For the boundaries of *large blocks* we store rank-directory (results of **rank**) in $R_l[0 \ldots n/l]$ explicitly using $\mathrm{O}(n/\log^2 n \cdot \log n) = \mathrm{O}(n/\log n)$ bits. We also store rank-directory for each boundary of small blocks

[2] We do not discuss $\mathbf{rank}_0$ since it can be computed by **rank** as $\mathbf{rank}_0(i, S) = i + 1 - \mathbf{rank}(i, S)$.

in $R_s[0 \dots n/s]$, 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 $\mathbf{rank}(x, S) = R_l[\lfloor x/l \rfloor] + R_s[\lfloor x/s \rfloor] + popcount(\lfloor x/s \rfloor \cdot s, x \bmod s)$, where $popcount(i, j)$ is the number of ones between $B[i \dots i+j]$ 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 [8][3]. 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 [8]. Because of the lack of space we omit the detail of **select** in constant time and see the detail in [14].

We next introduce Rank/Select dictionary using $nH_0(S) + o(n)$ bits, which is called **ent**. The main difference between **verbatim** and **ent** is the representation of the bit-vector itself, where 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 $\sum_{i=1 \dots u} \binom{t}{t-p_i-1}$ where p_i is the position of i-th one in the block. This value is the unique number in $\binom{[0, \lceil t}{m \rceil - 1]}$ for each possible block of t length with u ones. This number can be represented by $\mathcal{B}(t, u) = \lceil \lg \binom{t}{m} \rceil$ bits and the size of all encoded blocks is less than $\mathcal{B}(n, m) \leq nH_0(S)$ [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 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 **esp** (stands for EStimating Pointer information), which does not require *pointer information* by estimating it from **rank** information. Although the size of *pointer information* is $O(n \log\log n / \log n) = o(n)$, this size is actually as large as $\Theta(nH_0(S))$ terms for real-size data.

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

[3]In this paper let $a \bmod b$ denote $a - \lfloor a/b \rfloor$.

PROPOSITION 1. $L(B) \leq nH_0(B)$.

Because $nH_0(B)$ is the size of a representation of block that uses $\lg(n/m)$ bits for each 1's and $\lg(n/(n-m))$ bits for each 0's, and the $L(B) = \mathcal{B}(n, m) := \lceil \lg \binom{n}{m} \rceil$ is the smallest length of the code to represent the bit vector.

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

PROPOSITION 2. $\sum_{i=1}^{\lceil \frac{n}{u} \rceil} L(B_i) \leq \sum_{i=1}^{\lceil \frac{n}{u} \rceil} uH_0(B_i) \leq nH_0(B)$.

The second inequality holds because $H_0(B)$ is the concave function.

Let $B' := B[0 \dots t]$ $(t \leq n)$ 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 gap exists between the estimated position and the actual position, we use estimated position to store all code words at encoding. This means that we insert gap bits so that we always estimate the correct pointer information at reading time.

Figure 1 shows the example of coding in **esp**. We estimate pointer information using **rank** information and do not store the actual sizes of each encoded block.

We will explain the details of **esp**. Basically, **esp** is based on **ent** except for the existence of *super-large blocks* (SLB) used to regularly reset estimation errors. We conceptually partition B into subsequences of length $k := \log^3 n$ each, called *super large block* (SLB). Each SLB is partitioned into *large block* (LB) of length $l := \log^2 n$, each LB is partitioned again into *small block* (SB) of length $s := \log n/2$. We then encode each SB by its *enumerative code* (Section 2.1) independently. The code word for the i-th SB: SB_i is stored in the position which is determined as follows. Let l_r and s_r be results of **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 ,

$$\begin{aligned} lp &= H_0(LB'_{x_l}) \\ &= l_r \cdot \lg \frac{l \cdot x_l}{l_r} + (l \cdot x_l - l_r) \cdot \lg \frac{l \cdot x_l}{l \cdot x_l - l_r} \\ sp &= H_0(SB'_{x_s}), \\ &= s_r \cdot \lg \frac{s \cdot x_s}{s_r} + (s \cdot x_s - s_r) \cdot \lg \frac{s \cdot x_s}{s \cdot x_s - s_r}, \end{aligned}$$

where LB'_{x_l} denotes the preceding LBs from the boundary of SLB up to LB_i and SB'_i 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

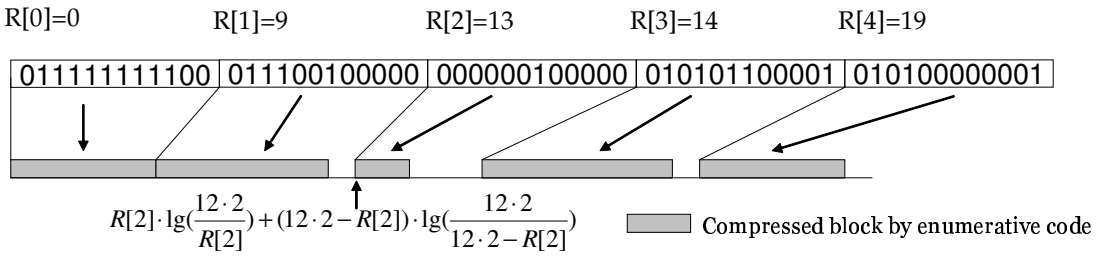

Figure 1: Example of **esp**. Estimate pointer information by **rank** information, $R[0\ldots 4]$ and compressed blocks are placed at estimated positions.

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 *pointer information* for SLB. All of them are stored in $o(n)$ bits.

For **rank**(x, S), we lookup the 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 *pointer information* for LB and SB as in encoding. We then read the compressed bit representation of SB from that position, decode it in constant time using pre-computed table and apply *popcount* as in **verbatim**.

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 a fixed-point integer representation. We require two integer multipliers and each integer addition for estimating one value of the entropy.

4 RecRank

The second data structure **recrank** employs 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.

```
int rank_rr(int pos, int bit){
    // z[i]: i-th contracted bit array
    // m[i]: i-th block size
    int p = pos;
    for (int i = 0; i < rec; i++){
        int blockPos = p / m[i];
        int b = z[i][blockPos];
       int r = rank(blockPos,z[i]);
        if (b == 0) {
            if (r == 0) return 0;
            else p = ((r * m[i]) - 1);
        } else
            p = ((r-1) * m[i]) + p % m[i];
    }
    return rank_1(p,B_e);
}
```

Figure 2: An example code of **rank** in **recrank**

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

$$(4.1)\ \mathbf{rank}(x, B) = \mathbf{rank}(\mathbf{rank}(\lfloor x/t \rfloor, B_c)\cdot t + (x \bmod t)\cdot B_c[\lfloor x/t \rfloor], B_e).$$

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, that is the probability of one in a bit-array is larger than 1/4. We denote the extracted bit array and the contracted bit array at i-th process as B_e^i and B_c^i. After u times of the reduction, we have t contracted bit arrays $B_c^1, B_c^2, .., B_c^t$ and the final *extracted* bit array B_e^t.

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 $t = \frac{1}{-\lg(1-p)}$ so that the $p(B_c)$ would be 0.5. This is because the probability of t bits being all zero is $(1-p)^t$

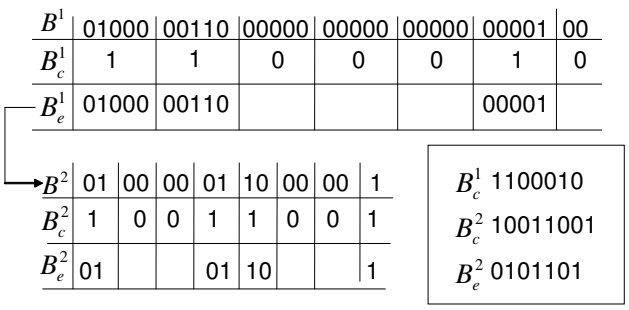

Figure 3: Example of coding in **recrank**. Input bit array B^1 is converted into dense bit arrays: B^1_c, B^2_c and B^2_e.

```
int select_vc(int i){ // return select(i,S)
    // b is the block number and q is the offset
    int b = i/p; int q = i%p;
    int x = S[b] + q;
    // count the number of ones
    //   in first q bits in each digit
    int mask = (1U << q) - 1;
    for (int j = 0; j < T[b]; j++)
        x += popcount[V[b][j] & mask]<< j;
    return x;
}
```

Figure 4: An example code of **select** in **vcode** written in C++. $V[p][j]$ contains $V_p[j]$ and $popcount[k]$ returns the number of set bit in binary sequence of k. Other variables correspond to the definition in the paper

and the half of the elements in the *contracted* bit array is one when $(1-p)^t = 1/2$. Then we can expect that the length of B_c is $-n\lg(1-p)$ 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 $u = -\lg p$ times so that the probability of ones in the final *extracted* bit array is larger than $1/4$.

Figure 3 shows the example of coding in **recrank**. In the first stage the block size is $5 = \lfloor \frac{1}{-\lg(1-4/32)} \rfloor$ and in the second stage the block size is $2 = \lfloor \frac{1}{-\lg(1-4/15)} \rfloor$.

Let T be the size of **recrank** and $p = 2^{-u}$. We can analysis T as follows,

$$\begin{aligned} T &= n \cdot \sum_{i=0\ldots u-2} \left(-\frac{\lg(1-2^i p)}{2^i} \right) + 2m \\ (4.2)\leq\ & n \cdot \frac{1}{\log_e(2)} \sum_{i=0\ldots u-2} \left(\frac{2^i p + 2\cdot(2^i p)^2/3}{2^i} \right) + 2m \\ &= 2m + n \cdot \frac{1}{\log_e(2)} \left(-p\lg p - 2p/3 - 2p^2/3 \right) \\ &= \frac{1}{\log_e(2)} \left(-m\lg p - 2m/3 - 2mp/3 \right) + 2m \end{aligned}$$

In (4.2), we use $\lg(1-x) \leq x + \frac{2}{3}x^2$ for $0 \leq x \leq \frac{1}{4}$. In short, T is bounded by $1.44m\lg n/m + m$ bits. We note that this is the expected size and the actual size depends on the bit array.

For **rank**, we apply (4.1) at each stage. Since the number of reduction is $-\lg p = \lg n/m$ 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 Vertical Code

A *Vertical Code* (**vcode**) supports fast **select** using small space in practice because of its byte-based operations and a novel orientation of data. This is a kind of opportunistic data structure, that is, although it is not an entropy-compressed Rank/Select dictionary in the worst case, in most case its size is close to the zero-th order *empirical entropy*.

Given a bit-arrays $B[0\ldots n-1]$ with m ones, we first convert it into the *gap* sequence $d[0\ldots m-1]$, $d[i] = \mathbf{select}(B, i+1) - \mathbf{select}(B, i) - 1$ $(i = 0\ldots m-1)$[4].

We then partition d into blocks $B_1, \ldots, B_{m/p}$ of size $p = O(\log n)$. Let $T[0\ldots m/p-1]$ be the arrays such that $T[i] = \lg\lfloor \max_{j=0\ldots p-1} d[ip+j] \rfloor$, $V_i[j]$ be the bit arrays of length p consisting of the set of the $j+1$-th bit of d in the block B_i, and $S[0\ldots m/p-1]$ be the arrays such that $S[i] = \mathbf{select}(B, ip)$. We note that all d in a block B_i can be represented in $T[i]$ bits each. Figure 5 shows the example of coding in **vcode**.

We describe how to get $\mathbf{select}(S, i)$ by using T, V and S. Let $b = i/p$ and $q = i \bmod p$. Since $\mathbf{select}(S, i) = S[b] + q + \sum_{i=bp}^{bp+q} d[i]$, we count the number of ones in the first q bits of each $V_b[0], \ldots, V_b[T_i]$, then sums them up with shift.

Figure 5 shows the example code of **select** in **vcode**.

The characteristic of **vcode** is that all operations are byte-aligned if we set p is a multiple of eight for any bit arrays. And the cost of $\sum_{i=bp}^{bp+q} d[i]$ is $O(T[b])$, which would be small if $T[b]$ is small. This idea is similar to *gap-based* compressed dictionary [11, 12]. In **vcode**we encode *gap* information directly and we can expect the time of **select** is small if *gap* is small. For example, the *gap* of ψ in compressed suffix arrays [24] is very small.

For **select**, we need to do $T[i]$ operations each of which is done in constant time. Since $T[i]$ would become $O(\log n)$ in the worst case, the total time for **select** is

[4] $d[0] = \mathbf{select}(B, 1)$

$O(\log n)$. For **rank** and **select**$_0$, we need to do the binary search from m elements using **select** which is done in $O(\log n \cdot \log m)$ time in the worst case.

The size of S is $O(\log n \cdot m/\log n) = O(m)$. Since $d[i] < n$, the size of T_i is bounded by $\lg n$ and T is bounded by $O(\log n \cdot m/\log n) = O(m)$ and the size of V is bounded by $m \lg(n/\lg n)$ bits, which occurs when $d[ip] = n/\lg n$ $(0 \leq i < n/p)$ and others $d[i]$ are all 0. we note that we can expect the size of V is close to $m \lg n/m$ $(\simeq nH_0(B))$ bits and the time of **select** is close to $O(1)$ when adjacent elements in d have similar values.

6 SDarrays

The idea of **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).

First we will introduce **sarray**. Given a bit-arrays $B[0 \ldots n-1]$ with m ones ($m \ll n$), we define $x[0 \ldots m-1]$ such that $x[i] = \textbf{select}(i+1, B)$. Each x is then divided into upper $z = \lfloor \lg m \rfloor$ bits and lower $w = \lceil \lg n/m \rceil$ bits. Lower bits are stored explicitly in $L[0 \ldots m-1]$ using $m \cdot \lceil \lg n/m \rceil$ bits. Upper bits are represented by a bit array $H[0 \ldots 2m-1]$ such that $H[x_i/2^w + i] = 1$ and others are 0. This can be seen as a unary coding of gaps between values of upper bits. By using H and L, we can calculate **select** in **sarray** by $\textbf{select}(i, B) = (\textbf{select}(i, H) - i) \cdot 2^w + L[i]$. Here we can assume that H is dense because there are m ones and m zeros in H.

Figure 7 shows the example of coding in **sarray**. Since $\lfloor \lg 8 \rfloor = 3$, we divide each x into upper 3 bits and lower 2 bits. We store L directly and convert upper bits into

We then explain **darray** for *dense sets*, $B[0 \ldots n-1]$ with $m \simeq n/2$ ones[5]. We first partition B into the blocks such that each block contains L ones respectively. Let $P_l[0 \ldots n/L - 1]$ be the arrays such that $P_l[i] := \textbf{select}(iL, B)$. We classify these blocks into two groups. If the length of a block $(P_l[i] - P_l[i-1])$ is larger than L_2, we store all the positions of ones explicitly in S_l. If the length of a block is smaller than L_2, we store the positions of each L_3-th ones in S_s. We can store these values in $\lg L_2$ bits by storing only relative values to ones stored for P_l.

For $\textbf{select}(i, B)$ in **darray**, we lookup $P_l[\lceil i/L \rceil]$ and see whether the block is larger than L_2 or not. If it is, we lookup the value in S_l which is stored explicitly.

[5]The size of H in **sarray** is $2m$ not n. Here we explain **darray** in general case.

```
int rank_sarray(int i){
    int y = select_0(i>>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 6: An example code of **rank** in **sarray**. Variables correspond to the definition in the paper

If not, we lookup the correspondent L_3-th value in S_s and then perform a sequential search in the block in $O(L_2/\log n)$ time because we can read $O(\log n)$ bits in RAM model. We note that if we can assume that ones are distributed in B uniformly, this sequential search is done in $O(1)$ time. Although this data structure concerns only **select**, we can use the same data for **select**$_0$ by reversing bits in H at reading time with corresponding auxiliary data structure as for **select**.

For **rank** in **darray**, we use the same method as in **verbatim**. For $\textbf{rank}(i, B)$ in **sarray**(see the example code in figure 6), we first calculate $y = \textbf{select}_0(i/2^w, H) + 1$ to find the smallest element which is greater than $\lceil i/2^w \rceil \cdot 2^w$. Then we count the number of elements which are equal to or smaller than i by sequentially searching over H and L in $O(n/m)$ time because the number of 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.

The size of P_l is $O(\frac{n}{L} \cdot \log n)$, that of S_l is at most $\frac{n}{L_2} \cdot L \lg n$ bits, and that of S_s is at most $\frac{n}{L_3} \lg L_2$ 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_l and S_l 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 $m \cdot \lceil \lg n/m \rceil$ bits for L. For H of length $2m$, we use the data structure of **darray**, which is $2m + o(m)$ bits. Therefore the total size of **sarray** is $m \cdot \lceil \lg n/m \rceil + 2m + o(m)$ bits.

7 Experimental Results

We conducted experiments using **esp** (*esp*), **recrank** (*rr*), **vcode** (*vc*), **sarray** (*sa*) and **darray** (*da*). We also compared them with byte-based implementation in [14] (*Kim*), and its re-implementation by us (*Kim2*) and the implementation in [8] (*navarro*). For each data structure, we used the following parameters. For *esp*, we used $k = 2^{12}$, $l = 2^8$, $s = 2^5$. For *vc*, we used $p = 8$.

B = 00110011100100001000001010011

i	0	1	2	3	4	5	6	7	8	9	10
D	2	0	2	0	0	2	4	5	1	3	0
T	2				3				2		
V[0]	0	0	0	0	0	0	0	1	1	1	0
V[1]	1	0	1	0	0	1	0	0	0	1	0
V[2]					0	0	1	1			

Figure 5: Example of coding in **vcode**. D represents the gap between ones.

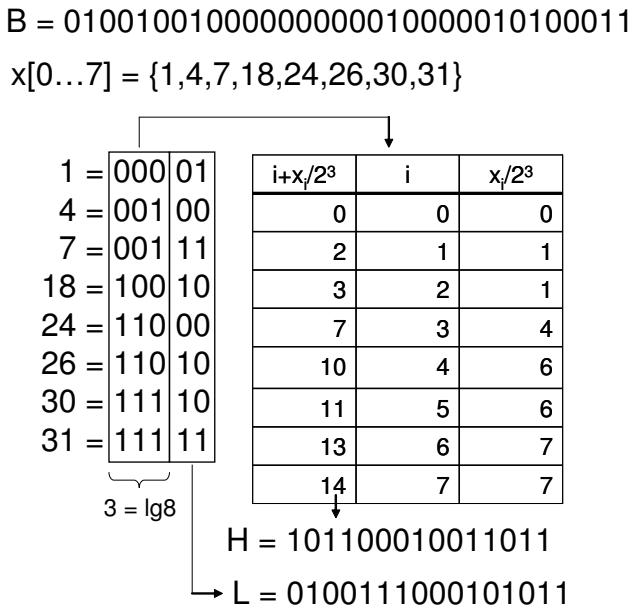

Figure 7: Example of coding in **sarray**. x is the positions of ones in B. Since the number of ones is 8, we divide each element in x into upper $3 = \lfloor \lg 8 \rfloor$ and lower $2 = \lceil \lg 32/8 \rceil$ bits.

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 for *select* would become large. For **rank** and **select** in *sa* and *da*, we used sequential search in H instead of the $O(1)$ solution because the sequential search is fast in practice. The source code in this experiment is found at "http://www-tsujii.is.s.u-tokyo.ac.jp/~hillbig/practical_ent_rs.zip".

We used GNU C 3.4.3 -O6 -m64. We measured time using the *ftime* function on a 3.4GHz Xeon with 8GB of main memory.

All experiments are done using bit arrays in which the positions of ones are determined randomly.

Figure 8 shows the measured size of several data structure and Figure 9 shows the zoom-in on the $1-10$ ratio of 1. We also show the result of $H_0(B)$ which is the lower bound on the size of a data structure if we only know the ratio of ones. The size of *esp* is close to zero-th order *empirical entropy* in all conditions. and the sizes of *rec*, *sa* and *vc* are very close to zero-th order *empirical entropy* 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 because the size of its auxially data structure is $o(m)$ not $o(n)$ and is small if m is small.

Figure 10, 11 and 12 are the results of 10^8 **rank** operations on the bit vector of 10^7, 10^8 and 10^9 length. We can see that *Kim2*, *Navarro* and *da* are the fastest, which is the same as for **rank** in **verbatim**. On the other hand *vc* is the slowest for **rank** because it needs binary search using **select** functions. Only *rec* becomes slower when the ratio of ones is small because its computation cost is $O(\log n/m)$, depending on the inverse number of m.

Figure 13, 14 and 15 are the results of 10^8 **select** operations on the bit vector of 10^7, 10^8 and 10^9 length. Among several methods, *sa* is the fastest in all conditions. As in the result of **rank**, *rec* is slower in the small ratio of 1. We also find that *da* shows different behavior in the small ratio of 1 because it switches data structures in function of the ratio of 1. We note that the result of *esp* for **rank** and **select** is fast if the ratio of 1 is small or large because *esp* employs decoding table for *enumerative code* which is only prepared for compressible block, that is the ratio of 1 is small or large.

We also note that *Navarro*, *rec*, *esp* *vc* which use binary search in **select** becomes slower when the size of bit arrays becomes large because the cache effects becomes large as discussed in [8].

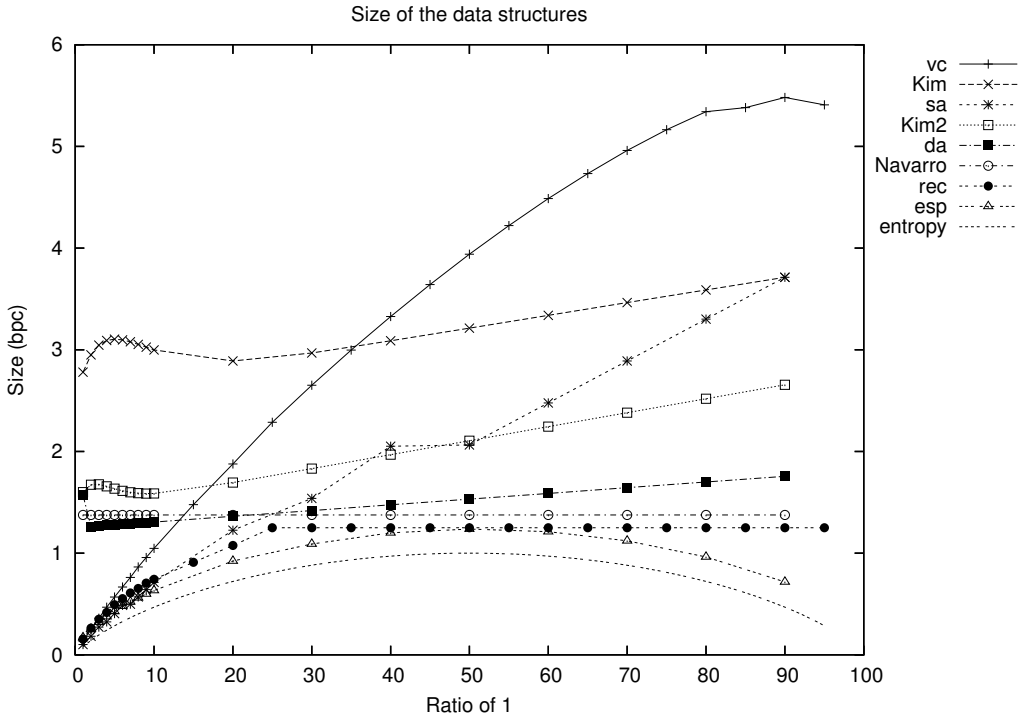

Figure 8: Size of the data structures with $1 - 100$ ratio of ones.

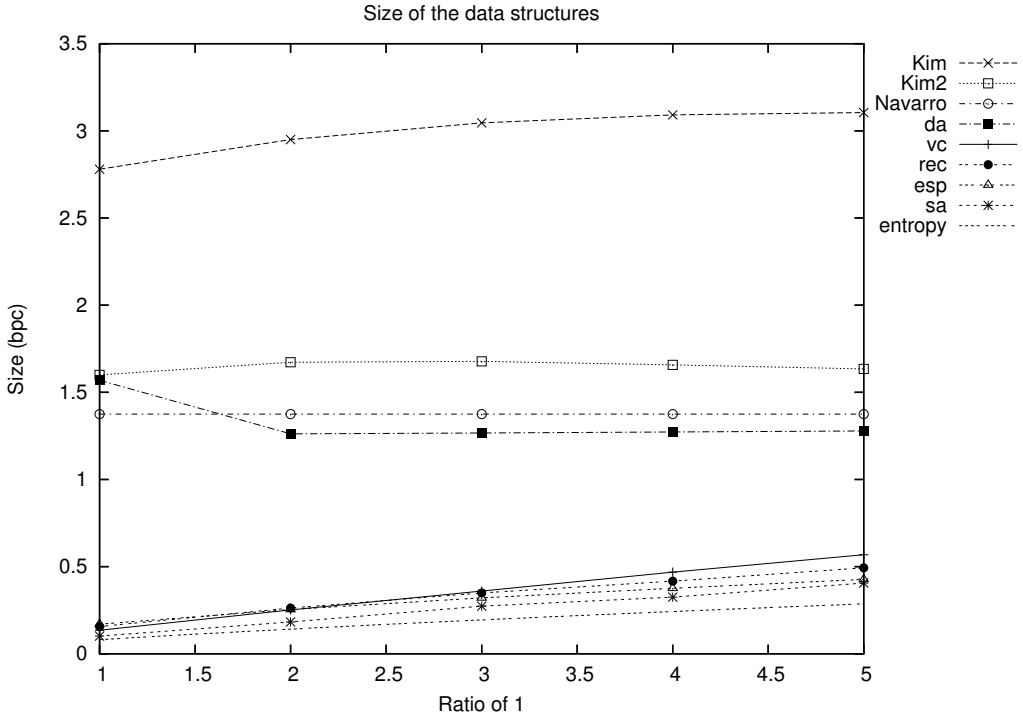

Figure 9: Size of the data structures of bit arrays with $1 - 5$ ratio of ones.

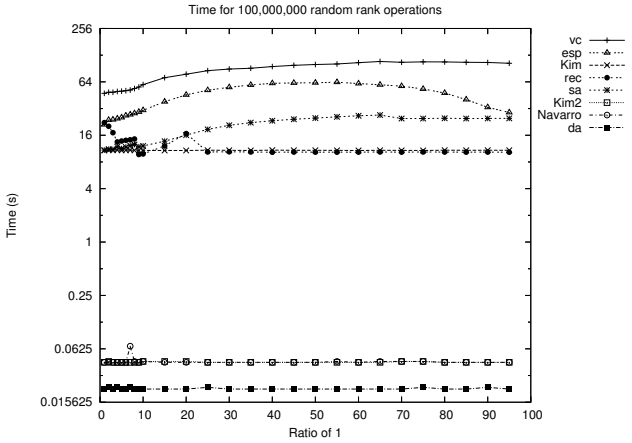

Figure 10: Time for 100,000,000 random rank operations on a bit vector of 10^7 length.

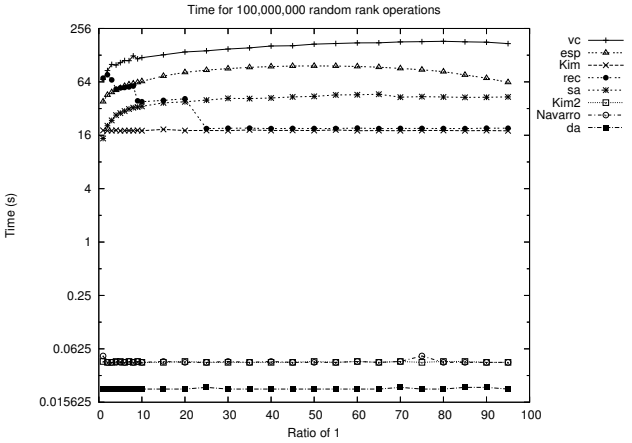

Figure 11: Time for 100,000,000 random rank operations on a bit vector of 10^8 length.

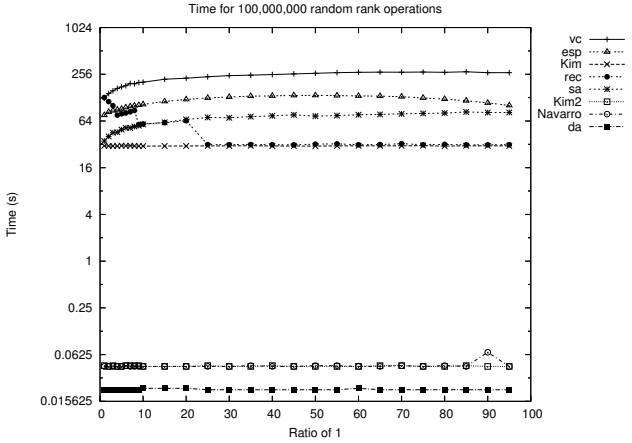

Figure 12: Time for 100,000,000 random rank operations on a bit vector of 10^9 length.

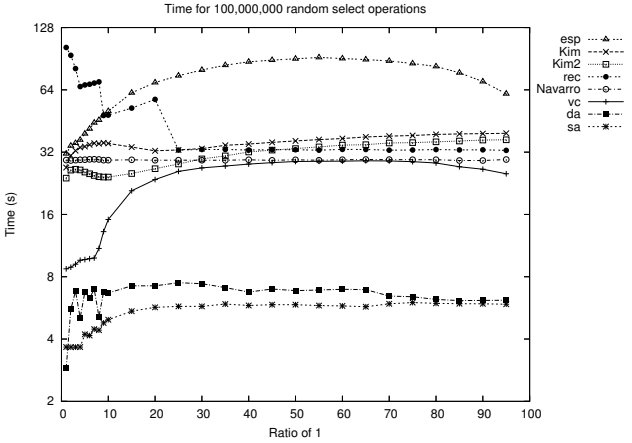

Figure 13: Time for 100,000,000 random select operations on a bit vector of 10^7 length.

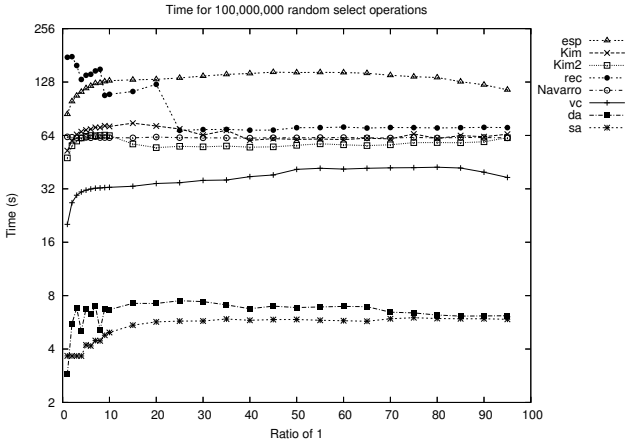

Figure 14: Time for 100,000,000 random select operations on a bit vector of 10^8 length.

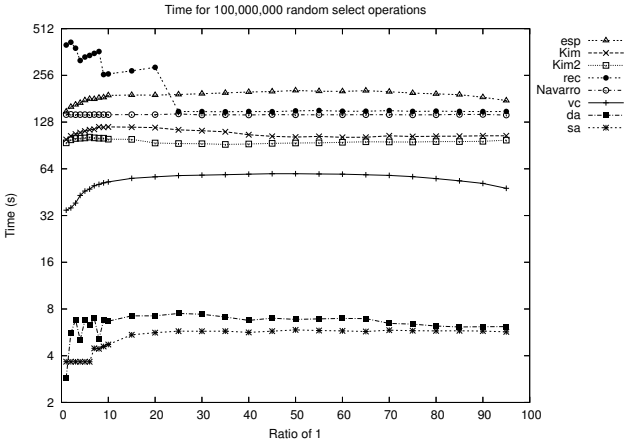

Figure 15: Time for 100,000,000 random select operations on a bit vector of 10^9 length.

Table 2: The space results for **esp**, **recrank**, **vcode**, **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**	**recrank**	**vcode**	**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

From the results of these experiments, we can choose data structures as follows. For *sparse sets*, **sarray**is the smallest among other data structure. For *dense sets* **darray**is small and this is the fastest in **select** queries. We also note that the size of **esp**is small for all condition.

8 Concluding Remarks

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

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

The remaining problem is as follows. Can we design a entropy-compressed Rank/Select dictionary which supports not only fast **select** operation but also fast **rank** operation ?

In the next stage of our research, we will extend our result to more complex data structures, such as sequences from large alphabets as [7]. We also consider applications which employ appropriate data structures and also apply them to data compression as well.

Acknowledgements. The authors would like to thank Prof. G. Navarro, who provided [8] and Prof. D. K. Kim, who provided [14]. The authors would like to thank Jérémy Barbay for his helpful comments. The work of the second author is supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

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 represengtation 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. Golynski. 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. Rman, 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 suffi 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.