

Practical Entropy-Compressed Rank/Select Dictionary

Daisuke Okanohara*

Kunihiko Sadakane†

Abstract

Rank/Select dictionaries are data structures for an ordered set $S \subset \{0, 1, \dots, n-1\}$ to compute $\mathbf{rank}(x, S)$ (the number of elements in S that 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 these 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 **sarray**, 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. Furthermore, their query times are superior to those of existing structures. Experimental results reveal the characteristics of our data structures and also show that these data structures are superior to existing implementations, both in terms of size and query time.

1 Introduction

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

- **rank**(x, S): the number of elements in S which are no greater than x ,
- **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 of representing an object from a universe with cardinality L by $(1 + o(1)) \lg L$ bits¹. 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 enu-

merations or navigations. Various succinct representation techniques have been developed to represent data structures such as ordered sets [15, 22, 23, 24], ordinal trees [1, 2, 6, 7, 16, 21, 26], strings [5, 10, 11, 25, 26, 27], functions [20], and labeled trees [1, 4]. 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 [8, 16, 19, 22, 24] 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, only their asymptotic behavior is considered, and their performance is not optimal for practical data sizes. As a result, both query times and the data structure sizes are large for real data. Although recently some practical implementations of Rank/Select dictionaries have been proposed using $n + o(n)$ bits [9, 17], there is no practical implementation of those using $nH_0(S) + o(n)$ bits. Recently *gap-based* compressed dictionaries have also been proposed [14, 13]. They use another measure called $gap(S) = \sum_{i=1 \dots m} [\lg(\mathbf{select}(i+1, S) - \mathbf{select}(i, S))]$ to define the minimum space to store S and propose a data structure using $gap(S) + O(m \log \frac{n}{m} / \log m) + O(n \log \log m/n)$ bits where m is the number of elements in S . This measure becomes much smaller than entropy-based measures if 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 **sarray** (**sarray** and **darray**), each of which is based on different ideas, and so 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 can even approach 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 $S \subset \{0, 1, \dots, n-1\}$ with m elements in terms of size and time. We note that these bounds represent the worst case, and that these quantities are smaller in practice. For example, the $O(\log^4 m / \log n)$ term in **sarray** and **darray** and

*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.

¹Let $\lg n$ denote $\log_2 n$

Table 1: The space and time results for **esp**, **recrank**, **vcode**, **sarray** and **darray** for an ordered set $S \subset \{0, 1, \dots, n-1\}$ 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)	$nH_0(S) + o(n)$	$O(1)$	$O(1)$
recrank (Sec. 4)	$1.44m \lg \frac{n}{m} + m + o(n)$	$O(\log \frac{n}{m})$	$O(\log \frac{n}{m})$
vcode (Sec. 5)	$m \lg(n/\lg n) + O(m)$	$O(\log^2 n)$	$O(\log n)$
sarray (Sec. 6)	$m \lg \frac{n}{m} + 1.92m + o(m)$	$O(\log \frac{n}{m}) + O(\log^4 m / \log n)$	$O(\log^4 m / \log n)$
darray (Sec. 6)	$n + o(n)$	$O(1)$	$O(\log^4 m / \log n)$

$O(\log n)$ term in **vcode** are $O(1)$ in almost all cases.

We conducted experiments using the proposed methods as well as existing methods, and found that our data structures are faster and smaller than the existing methods.

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 from/to any address in constant time.

An ordered set S , which is a subset of the universe $U = \{0, 1, \dots, n-1\}$, can be represented by a bit-vector $B[0, \dots, n-1]$ 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 **rank** (x, S) is the number of ones in $B[0, x]$, and **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 [19].

The above representation of S using a 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 $nH_0(B) \simeq m \lg \frac{n}{m} + 1.44m$ bits if $m \ll n$. Therefore the size of the *verbatim representation* is much larger than this lower-bound if $m \ll n$. Raman et al. [24] proposed a constant-time Rank/Select data structure whose size is $\mathcal{B}(n, m) + O(n \log \log n / \log n)$, which tends to be the above lower-bound and the recent lower bounds [18, 8] in the limit as $n \rightarrow \infty$.

The applications of Rank/Select dictionaries can be divided into two groups: One is for sets with $m \simeq n/2$ while the other is for sets with $m \ll n$. In this pa-

per we call the former *dense sets* and the latter *sparse sets*. Typical applications of *dense sets* are for the wavelet trees [10] that are used for indexing strings. 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. Since 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. For example, an ordinal tree with n nodes is encoded in a bit-vector of length $2n$, and to support tree navigation operations, the bit-vector is divided into blocks of length $\frac{1}{2} \lg n$ bits, where 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 1. The ratio of ones is $2/\lg n$, that is, the vector is sparse. Such vectors can be encoded in $\mathcal{B}(2n, 4n/\lg n) + O(n \log \log n / \log n) = O(n \log \log n / \log n) = o(n)$ bits. The storage of such sparse vectors in a compressed form is therefore important for the use of succinct data structures.

In this paper we will mainly focus on *sparse sets* to support **rank** and **select** functions. In some applications, for example wavelet trees, we also need a **select** $_0$ function defined as **select** $_0(i, S)$: the i -th smallest element not in S^2 . We do not discuss **select** $_0$ in this paper because we will assume we have *dense sets* in applications using **select** $_0$.

2.1 Existing Implementations of Rank/Select

Dictionaries We first give a brief description of a Rank/Select dictionary using $n + o(n)$ bits, which is called **verbatim**. We conceptually partition B into subsequences of length $l := \log^2 n$, calling these subsequences *large blocks*. Then, each *large block* is partitioned into subsequences of length $s := \log n/2$, called *small blocks*. For the boundaries of *large blocks* we store rank-directory (results of **rank**) in $R_l[0 \dots n/l]$ explic-

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

itly using $O(n/\log^2 n \cdot \log n) = O(n/\log n)$ bits. We also store a rank-directory for each small block boundary in $R_s[0 \dots n/s]$, but here we store only values relative to the values stored for the large blocks. These small-block rank-directories are stored in $O(n \log \log n / \log n)$ bits.

Next, **rank** is computed by $\mathbf{rank}(x, S) = R_l[\lfloor x/l \rfloor] + R_s[\lfloor x/s \rfloor] + \mathit{popcount}(\lfloor x/s \rfloor \cdot s, x \bmod s)$, where $\mathit{popcount}(i, j)$ 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 $\mathit{popcount}$ function [9]³. For **select** we have two options; the first is a constant time solution using $o(n)$ auxiliary data structures [17], while the second is an $O(\log n)$ solution given by a binary search using **rank** functions without any auxiliary data structures [9]. Due to space limitations, we omit discussion of the detail of **select** in constant time, the interested reader may refer to [17] for further details.

We next introduce a Rank/Select dictionary using $nH_0(S) + o(n)$ bits 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* [3]. Figure 1 shows an example of enumerative coding. Given a bit vector of length t with u ones, enumerative code return the unique number in $[0, \binom{t}{u} - 1]$. This number can be represented by $\mathcal{B}(t, u) = \lceil \lg \binom{t}{u} \rceil$ bits. We represent each *small block* as the result of *enumerative code*, and the total size is less than $nH_0(S)$ [22]. Since they have different sizes, we also need to store pointers to compressed small blocks, which requires $O(n \log \log n / \log n) = o(n)$ bits. This encoding and decoding is performed using a 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 large 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 practical data sizes.

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* [3] (See Section 2.1). We then have the following proposition.

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

```
int enumCode(bool* B, int u, int t){
// B[i] : i-th bit
// u    : the number of ones in B
// t    : the length of B
// choose(a,b) = (a \choose b) if a >= b
//              0           otherwise
int x = 0;
for (int i = 0; i < t; i++){
    if (B[i] == 1){
        x += choose(t-i-1, u);
        u--;
    }
}
return x;
}
```

Figure 1: An example code of enumerative code. Return the unique number in $[0, \binom{n}{m} - 1]$ for each possible block of length n with m ones.

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

Proof. Since $nH_0(B)$ is the size of a representation of a block that uses $\lg \frac{n}{m}$ bits for each 1 and $\lg(n/(n-m))$ bits for each 0, and the $L(B) = \mathcal{B}(n, m) := \lceil \lg \binom{n}{m} \rceil$ is the rounded up value of the smallest length of the code to represent the bit vector, we have the claim.

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.

$$\begin{aligned} \sum_{i=1}^{\lceil \frac{n}{u} \rceil} L(B_i) &\leq \sum_{i=1}^{\lceil \frac{n}{u} \rceil} (uH_0(B_i) + 1) \\ (3.1) \qquad \qquad &\leq nH_0(B) + n/u + 1. \end{aligned}$$

Proof. The second inequality holds because $H_0(B)$ is a 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 $tH_0(B') + o(t)$ bits. However, since a gap exists between the estimated position and the actual position, we use the 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. Since $H_0(B')$ can be calculated by *rank* information, we do not require *pointer information*. The total code size is $nH_0(B) + n/u + 1 = nH_0(B) + o(n)$ bits because $u = \log n/2$ and $n/u + 1 = o(n)$.

Figure 2 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 now explain the details of **esp**. The structure of **esp** is essentially based on **ent** but differs in that it

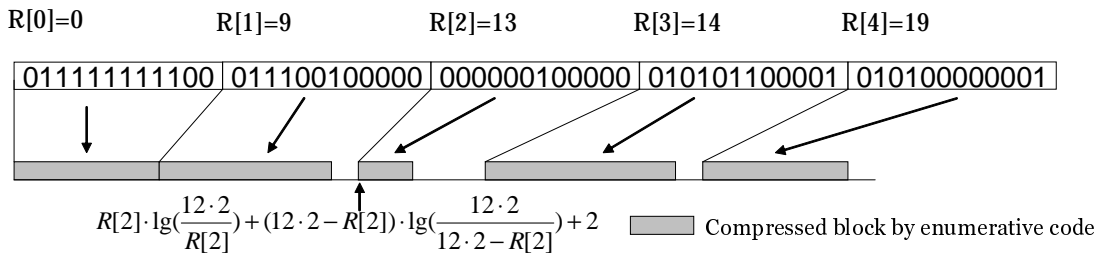


Figure 2: Example of **esp**. Estimate pointer information by **rank** information, $R[0 \dots 4]$ and compressed blocks are placed at estimated positions.

makes use of *super-large blocks* (SLB) to regularly reset estimation errors. These SLB's are the subsequences that arise from conceptually partitioning B into subsequences of length $k := \log^3 n$. Each SLB is in turn partitioned into *large blocks* (LB) of length $l := \log^2 n$, and each LB is partitioned again into *small blocks* (SB) of length $s := \log n/2$. We then independently encode each SB using its *enumerative code* (Section 2.1). The code word for the i -th SB: SB_i is stored in the position determined as follows: Let l_r and s_r be results of **rank** for LB and SB given 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_{x_l} , and SB'_{x_s} denotes the preceding SBs from the boundary of LB up to SB_{x_s} . Then the position for compressed SB_i is $slp + lp + x_l + sp + x_k^4$, where slp is the pointer information of SLB which is stored explicitly. We note that none of the code words overlap (use Prop.(2)) and that gap-bits are automatically inserted.

We store rank-directory for LB, SB and *pointer information* for SLB. All of these quantities 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 for the encoding. We then read the compressed bit representation of SB from that position, decode it in constant time

⁴We need x_l and x_k for rounding up.

using a pre-computed table and apply *popcount* as in **verbatim**.

For **select**, we use the same approach as in [17], which can be accomplished in constant time with the $o(n)$ -bit auxiliary data structures.

In practice, since it is very slow to compute the logarithm of a floating-point number for the purposes of estimating the entropy, we use a pre-computed table lookup approach and also use a fixed-point integer representation. We require two integer multiplications and an integer addition to estimate 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 originally used for Algorithm I in [17]. Here we use the reduction recursively.

Given a bit-array $B[0 \dots n-1]$ with m ones, we conceptually partition B into the blocks $B_0, \dots, B_{n/s}$ of length s . We call a zero block (ZB) a block where all elements are 0, and a non-zero block (NZ) a block where there is at least one 1. The *contracted* bit-array of B , $B_c[0, \dots, n/s-1]$ 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 the 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

$$\begin{aligned}
 (4.2) \text{rank}(x, B) &= \text{rank}(\text{rank}(\lfloor x/s \rfloor, B_c) \cdot s \\
 &\quad + (x \bmod t) \cdot B_e[\lfloor x/s \rfloor], B_e).
 \end{aligned}$$

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 of the i -th process by B_e^i and B_c^i respectively. After applying the reduction t times, we have t contracted bit arrays $B_c^1, B_c^2, \dots, B_c^t$ and

```

int rank_recrank(int pos, int bit){
  // c[i]: i-th contracted bit array
  // s[i]: i-th block size
  // t : the number of reduction
  int p = pos;
  for (int i = 0; i < t; i++){
    int blockPos = p / s[i];
    int b = c[i][blockPos];
    int r = rank(blockPos,c[i]);
    if (b == 0) {
      if (r == 0) return 0;
      else p = ((r * s[i]) - 1);
    } else
      p = ((r-1) * s[i]) + p % m[i];
  }
  return rank_1(p,B_e);
}

```

Figure 3: An example code of **rank** in **recrank**

the final *extracted* bit array B_e^t .

Here we adopt the strategy that *contracted* bit arrays should be *dense* (the probability of ones in the bit array should be $1/2$) by choosing an appropriate t in each process. Let $p(B) = m/n$ be the probability of ones in the bit array B . We choose the block size $s = \frac{1}{-\lg(1-p)}$ so that the $p(B_c)$ is 0.5. This is because the probability of s bits being all zero is $(1-p)^s$ and half of the elements in the *contracted* bit array are one when $(1-p)^s = 1/2$. The expected value of the length of B_c is $-n\lg(1-p)$, while the expected value of 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 $t = -\lg p$ times so that the probability of ones in the final *extracted* bit array is larger than $1/4$.

Figure 4 shows an 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 $p = 2^{-t}$. We can analyze T as follows,

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

In inequality (4.3), we use $\lg(1-x) \leq x + \frac{2}{3}x^2$ for

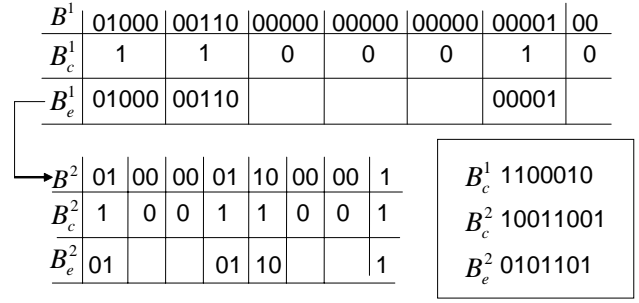


Figure 4: Example of coding in **recrank**. Input bit array B^1 is converted into dense bit arrays: B_c^1 , B_c^2 and B_e^2 .

$0 \leq x \leq \frac{1}{4}$. In short, T is bounded by $1.44m \lg \frac{n}{m} + m$ bits. We note that this is the expected size; the actual size depends on the bit array.

For **rank**, we apply equation (4.2) at each stage. Since the number of reductions is $-\lg p = \lg \frac{n}{m}$ and each stage is done in constant time, the total time is $O(\log \frac{n}{m})$. For **select**, we apply **select** in each stage, each of which is done in constant time [17], so the total time is of $O(\log \frac{n}{m})$.

5 Vertical Code

A *Vertical Code* (**vcode**) supports a fast **select** using a 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 cases its size is close to the zero-th order *empirical entropy*.

Given a bit-array $B[0 \dots n-1]$ with m ones, we first convert it into the *gap* sequence $d[0 \dots m-1]$, $d[i] = \mathbf{select}(B, i+1) - \mathbf{select}(B, i) - 1$ ($i = 0 \dots m-1$)⁵.

We then partition d into blocks $B_1, \dots, B_{m/p}$ of size $p = O(\log n)$. Let $T[0 \dots m/p-1]$ be the arrays such that $T[i] = \lg[\max_{j=0..p-1} d[ip+j]]$, $V_i[j]$ 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 $S[0 \dots m/p-1]$ be the arrays such that $S[i] = \mathbf{select}(B, ip)$. We note that all the d in a block B_i can be represented by $T[i]$ bits. Figure 6 shows an example of coding in **vcode**.

We now 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 $V_b[0], \dots, V_b[T_i]$, then sum them up using a shift.

⁵ $\overline{d[0]} = \mathbf{select}(B, 1)$

```

int select_vcode(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 5: 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

Figure 5 shows an example coding of **select** in **vcode**.

The characteristic of **vcode** is that all operations are byte-aligned for any bit array if we set p to be a multiple of eight. Furthermore, 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 a *gap-based* compressed dictionary [14, 13]. In **vcode** we encode *gap* information directly and we can expect the time of **select** to be small if *gap* is small. For example, the *gap* of ψ in compressed suffix arrays [27] is very small.

For **select**, we need to do $T[i]$ operations, each of which is completed in constant time. Since $T[i]$ would become $O(\log n)$ in the worst case, the total time for **select** is $O(\log n)$. For **rank** and **select₀**, we need to do the binary search from m elements using **select**. This can be achieved 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 $O(\log n \cdot m / \log n) = O(m)$. 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 all the other $d[i]$ are all 0. It may be noted that we can expect the size of V to be close to $m \lg \frac{n}{m}$ ($\simeq nH_0(B)$) bits and the time of **select** is close to $O(\frac{m}{m})$ when adjacent elements in d have similar values.

6 SDarrays

The idea of **sdarray** is to use two different techniques to treat *sparse sets* and *dense sets* separately, 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**. The encoding idea of **sarray** is the same as that described in [12].

We introduce **rank** operation on this data structure. Moreover **sarray** is smaller than the previous one.

Given a bit-array $B[0 \dots n-1]$ with m ones ($m \ll n$), we define $x[0 \dots m-1]$ such that $x[i] = \mathbf{select}(i+1, B)$. Given a parameter t , each x is divided into upper $z = \lceil \lg t \rceil$ bits and lower $w = \lfloor \lg \frac{n}{t} \rfloor$ bits. Lower bits are stored explicitly in $L[0 \dots m-1]$ using $m \cdot w$ bits. Upper bits are represented by a bit array $H[0 \dots m+t-1]$ such that $H[x_i/2^w + i] = 1$ and the other values are 0. This can be seen as a unary coding of gaps between values of upper bits and there are m ones and $2^z = t$ zeros in H . By using H and L , we can calculate **select** in **sarray** by $\mathbf{select}(i, B) = (\mathbf{select}(i, H) - i) \cdot 2^w + L[i]$. The total size of H and L is $m + t + m(\lg \frac{n}{t})$ bits, which is the convex function and is minimized when $t = m \lg e \simeq 1.44m$. The size is therefore $1.92m + m \lg \frac{n}{m}$ bits when $t = 1.44m$. Here we can assume that H is dense because there are m ones and $t = 1.44m$ zeros in H .

Figure 8 shows an example of coding in **sarray**. Since $t = \lceil \lg(1.44 \cdot 8) \rceil = 4$, we divide each x into an upper 4 bits and a lower 1 bits.

We now explain **darray** for *dense sets*, $B[0 \dots n-1]$ with $m \simeq n/2$ ones⁶. We first partition B into blocks such that each block contains L ones. Let $P_l[0 \dots n/L-1]$ be arrays such that $P_l[i] := \mathbf{select}(iL, B)$. 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 iL_3 -th ones ($i = 0 \dots L_2/L_3$) in S_s . We can store these values in $\lg L_2$ bits by storing only values relative to those stored for P_l .

For **select**(i, B) in **darray**, we lookup $P_l[\lceil i/L \rceil]$ and see whether or not the block is larger than L_2 . If it is, we lookup the value in S_l which is stored explicitly. If not, we lookup the corresponding L_3 -th value in S_s and then perform a sequential search in the block which can be achieved in $O(L_2 / \log n)$ time because we can read $O(\log n)$ bits in the RAM model. We note that if we can assume that ones are distributed uniformly in B , this sequential search can be carried out in $O(1)$ time. Although this data structure concerns only **select₁**, we can use the same data for **select₀** by reversing bits in H at reading time with the corresponding auxiliary data structure for **select₀** as for **select₁**.

For **rank** in **darray**, we use the same method as in **verbatim**. For **rank**(i, B) in **sarray** (see the example code in figure 7), we first calculate $y = \mathbf{select}_0(i/2^w, H) + 1$ to find the smallest element which

⁶The size of H in **sarray** is $2.44m$ not n . Here we explain **darray** in general case.

B = 001100111001000010000010100011

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 6: Example of coding in **vcodes**. D represents the gap between ones.

```
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 7: An example code of **rank** in **sarray**. Variables correspond to the definition in the paper

is greater than $\lceil i/2^w \rceil \cdot 2^w$. We then 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 patterns of length $\lg \frac{n}{m}$ is n/m . If we use a binary search, this can be accomplished in $O(\log \frac{n}{m})$ time, but this is slower than a 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. If we choose $L := O(\log^2 n)$, $L_2 := O(\log^4 n)$, and $L_3 := O(\log n)$, 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 now analyze the size of **sarray**. The total size of L and H is $1.92m + m \lg \frac{n}{m}$ bits and the size of auxiliary data structure for H is $o(m)$ bits. The total size of **sarray** is therefore $1.92m + m \lg \frac{n}{m} + o(m)$ bits.

7 Experimental Results

We conducted experiments using **esp** (*esp*), **recrank** (*rr*), **vcodes** (*vc*), **sarray** (*sa*) and **darray** (*da*). We also compared the results with byte-based implementation in [17] (*Kim*), and its re-implementation by the present authors (*Kim2*), and the implementation in [9]

B = 01001001000000000010000010100011

$x[0...7] = \{1,4,7,18,24,26,30,31\}$

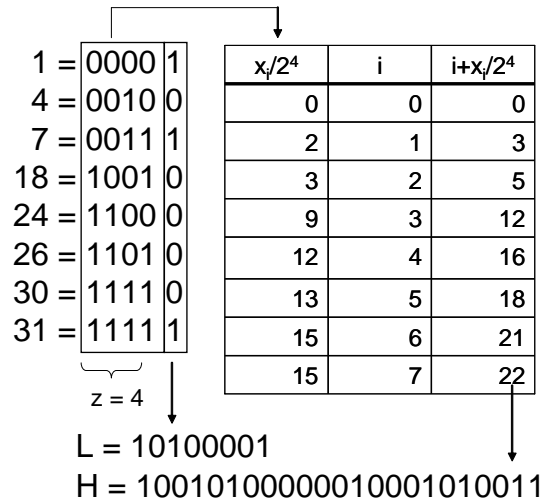


Figure 8: Example of coding in **sarray**. x is the positions of ones in B . Since the number of ones is 8, we divide each element in x into upper 4 = $\lceil \lg(1.44 \cdot 8) \rceil$ and lower 1 bit.

(*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 $p = 8$. 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 a sequential search in H instead of the $O(1)$ solution, because a sequential search is faster 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

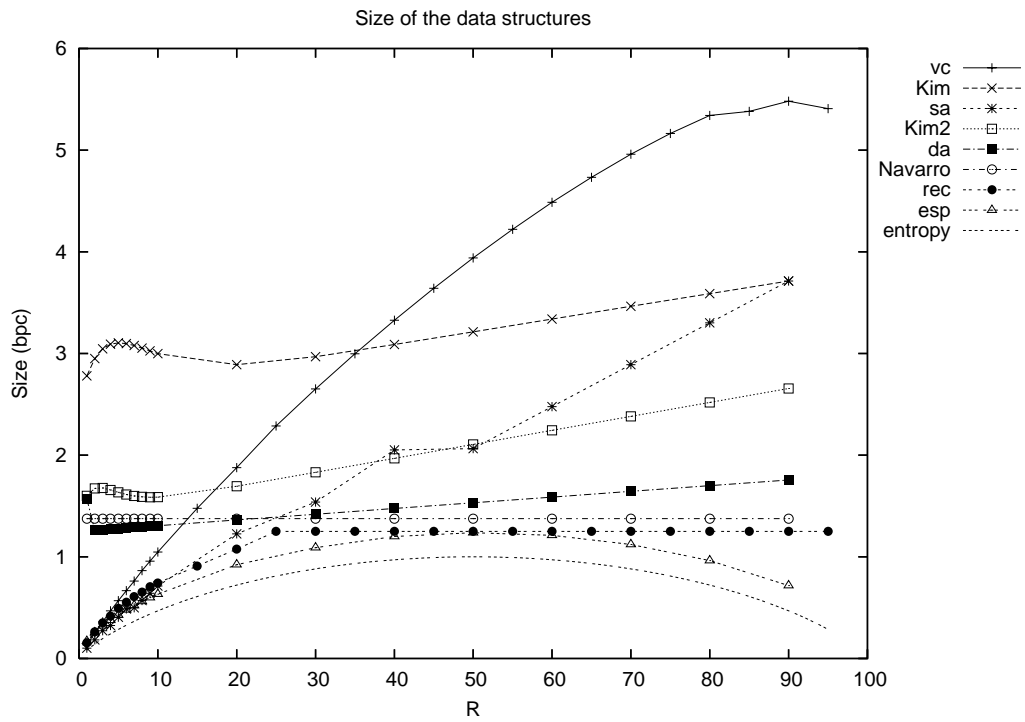


Figure 9: Size of the data structures with $1 \leq R \leq 100$.

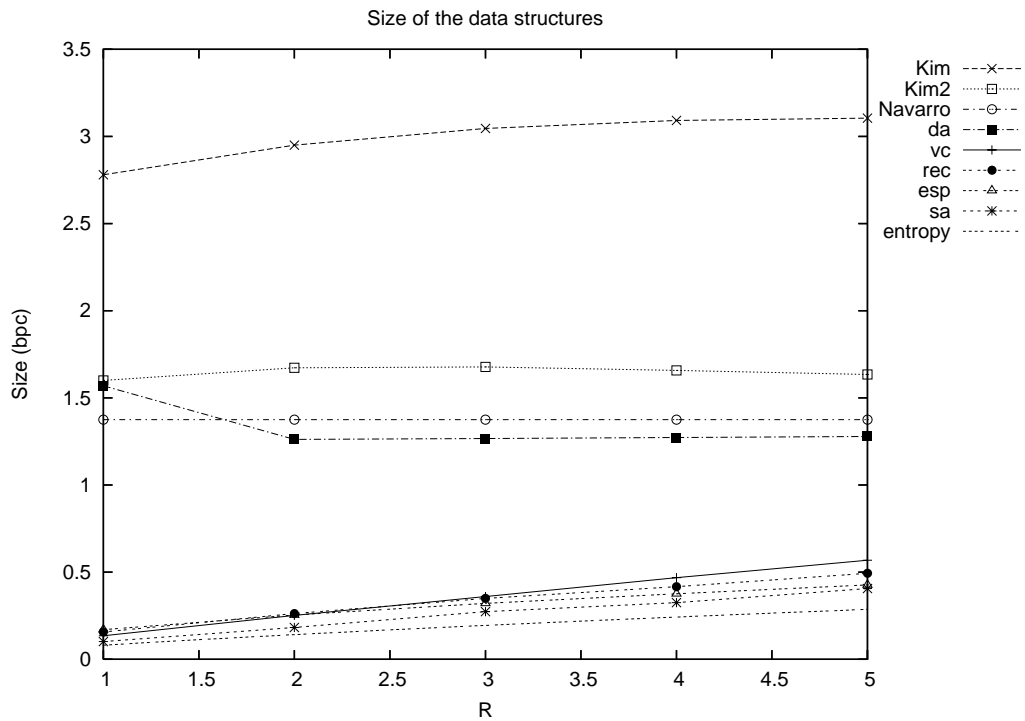


Figure 10: Size of the data structures of bit arrays with $1 \leq R \leq 5$.

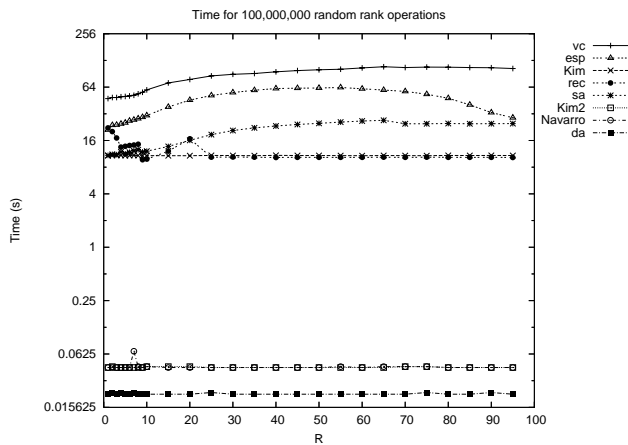


Figure 11: Time for 100,000,000 random rank operations on a bit vector of 10^7 length.

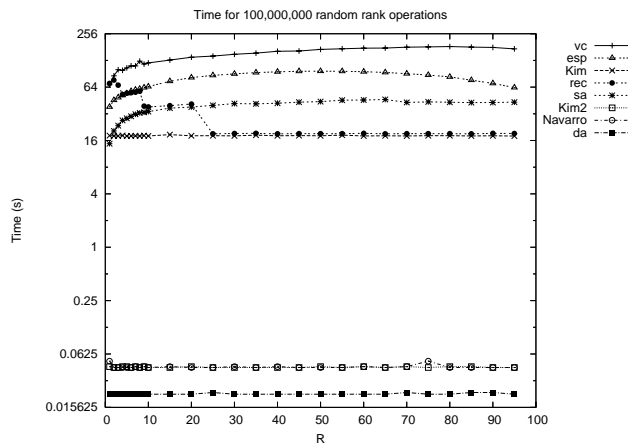


Figure 12: Time for 100,000,000 random rank operations on a bit vector of 10^8 length.

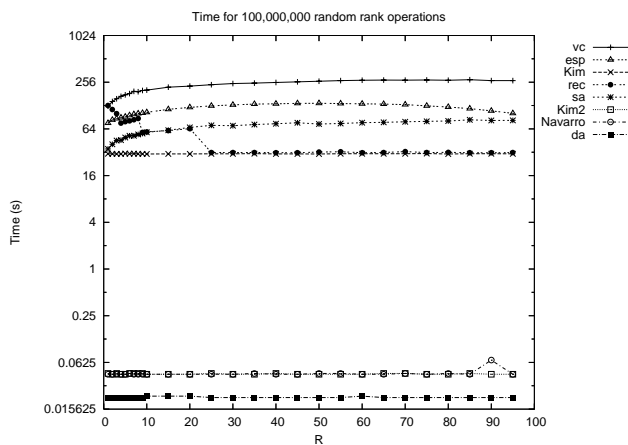


Figure 13: Time for 100,000,000 random rank operations on a bit vector of 10^9 length.

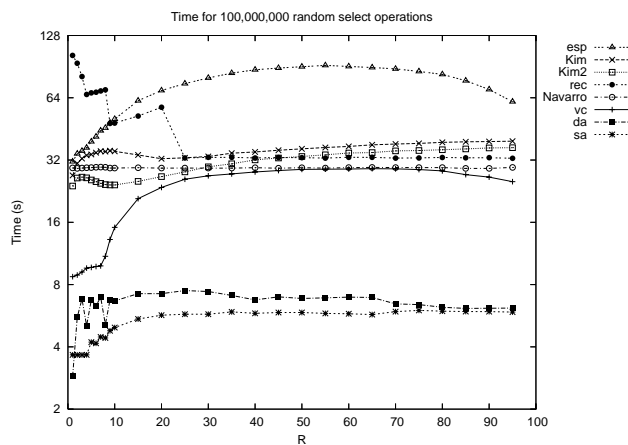


Figure 14: Time for 100,000,000 random select operations on a bit vector of 10^7 length.

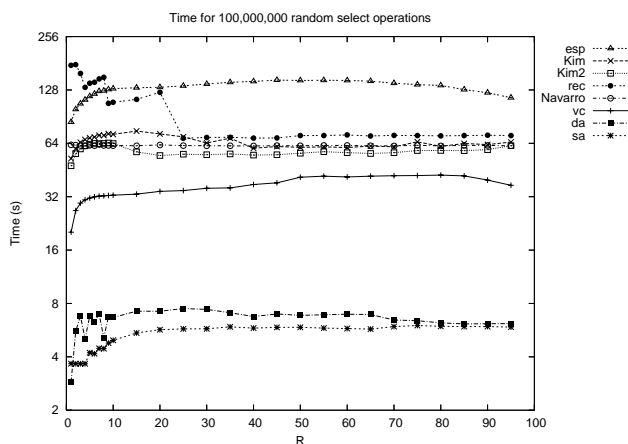


Figure 15: Time for 100,000,000 random select operations on a bit vector of 10^8 length.

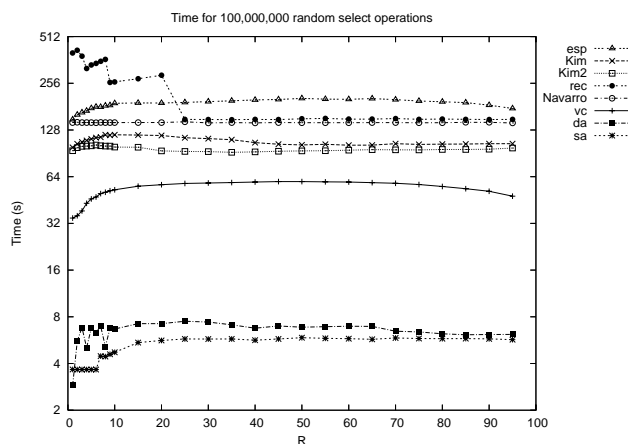


Figure 16: Time for 100,000,000 random select operations on a bit vector of 10^9 length.

Table 2: The space results for **esp**, **recrank**, **vc**, **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.

R	esp	recrank	vc	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

using the *f_{time}* function on a 3.4GHz Xeon with 8GB of main memory.

We define R to be the percentage of 1's in the bit array.

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

Figure 9 shows the measured size of several data structures and Figure 10 shows enhanced detail of the plot of Figure 9 in the domain of R from 1 to 5. 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 R . The size of *esp* is close to the zero-th order *empirical entropy* in all conditions, while the sizes of *rec*, *sa* and *vc* are very close to zero-th order *empirical entropy* when R is very small.

Table 2 shows the sizes of each data structure for $R = 1, 5$. We find that the sizes of proposed data structures are indeed close to nH_0 . We note that **sarray** is the smallest in both cases because the size of its auxiliary data structure is $o(m)$ not $o(n)$, and is small if m is small.

Figures 11, 12 and 13 display 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 result as for **rank** in **verbatim**. On the other hand *vc* is the slowest for **rank** because it requires a binary search using **select** functions. Only *rec* becomes slower when R is small because its computation cost is $O(\log \frac{n}{m})$, thus depending on the inverse number of m . We note that we can swap the roles of 1s and 0s when $R > 1/2$. In this case, *rec* will be a little worse in space than *esp*, but faster than *esp* in the most of R .

Figures 14, 15 and 16 display 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 for smaller R . We also find that *da* exhibits a different behavior for smaller R because it switches data structures as a function of R . We note that the result of *esp* for **rank** and **select** is fast if R is small or large because *esp* employs a decoding table for *enumerative code*, which is only prepared for compressible blocks, that is R is small or large.

We also note that *Navarro*, *rec*, *esp* and *vc*, which use a binary search in **select**, become slower when the size of bit arrays becomes large because the cache effects becomes large as discussed in [9].

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 fastest in **select** queries. We also note that the size of **esp** and **recrank** (with swapping the roles of 1s and 0s when R is large) are small for all conditions. Although we did not examine the performance for a bit vector in which its *gap* is small, we expect **vc** is small for such a bit vector.

8 Concluding Remarks

In this paper, we have proposed four novel Rank/Select dictionaries, **esp**, **recrank**, **vc** 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, **vc** converts the problem into the *popcount* in bytes, and **sdarray** uses separate techniques for *dense set* and *sparse set* problems, which simplifies the problem.

The remaining problem is as follows. Can we design an entropy-compressed Rank/Select dictionary which supports not only a fast **select** operation but also a 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 [8]. We also consider applications employing appropriate data structures, and will also apply them to data compression.

Acknowledgements. The authors would like to thank Prof. G. Navarro, who provided [9] and Prof. D. K. Kim, who provided [17]. The authors would also like to thank Jérémy Barbay for his helpful comments. The authors appreciate useful discussions with Y. Yoshida. 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] Y. T. Chiang, C. C. Lin, and H. I. Lu. Orderly spanning trees with applications. *SIAM Journal on Computing*, 34(4):924–945, 2005.
- [3] T. Cover. Enumerative source encoding. *IEEE Trans. on Information*, 19(1):73–77, 1973.
- [4] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *FOCS*, 2005.
- [5] P. Ferragina and G. manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.
- [6] R. Geary., N. Rahman., R. Raman., and V. Raman. A simple optimal representation for balanced parentheses. In *Proc. of CPM*, pages 159–172, 2004.
- [7] R. Geary., N. Rahman., and V. Raman. Succinct ordinal trees with level-ancestor queries. In *ACM-SIAM SODA*, pages 1–10, 2004.
- [8] A. Golyński. Optimal lower bounds for rank and select indexes. In *Proc. of ICALP*, 2006.
- [9] R. González, S. Grabowski, V. Mäkinen, and G. 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.
- [10] R. Grossi., A. Gupta., and J. Vitter. High-order entropy-compressed text indexes. In *Proc. of SODA*, pages 841–850, 2003.
- [11] 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.
- [12] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
- [13] A. Gupta, W. Hon, R. Shar, and J. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In *Proc. of WEA*, 2006. To appear.
- [14] A. Gupta, W. K. Hon, R. Shar, and J. Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Proc. of DCC*, pages 213–222. IEEE, 2006.
- [15] W. K. Hon, K. Sadakane, and W.K. Sung. Succinct data structures for searchable partial sums. In *Proc. of ISAAC*, pages 505–516, 2003.
- [16] G. Jacobson. Space-efficient static trees and graphs. In *Proc. of FOCS*, pages 549–554, 1989.
- [17] 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.
- [18] P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proc. of SODA*, pages 11–12, 2005.
- [19] J. I. Munro. Tables. In *Proc. of FSTTCS*, pages 37–42, 1996.
- [20] J. I. Munro and S. S. Rao. Succinct representations of functions. In *Proc. of ICALP*, pages 1006–1015, 2004.
- [21] J. I. Munro, V. Rman, and S. S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.
- [22] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM J. Computation*, 31(2):353–363, 2001.
- [23] C. K. Poon and W. K. Yiu. Opportunistic data structures for range queries. In *Proc. of COCOON*, pages 560–569, 2005.
- [24] 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.
- [25] S. S. Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, 2002.
- [26] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *ACM-SIAM SODA*, pages 225–232, 2002.
- [27] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.