In: Software Engineering and Development
Editor: Enrique A. Belini, pp. 3-12
ISBN 978-1-60692-146-3

Expert Commentary A

SUCCINCT REPRESENTATION OF BIT VECTORS SUPPORTING EFFICIENT *rank* AND *select* QUERIES

Jesper Jansson[1,*,†] ***and Kunihiko Sadakane***[2,‡]
[1]Ochanomizu University, 2-1-1 Otsuka,
Bunkyo-ku, Tokyo 112-8610, Japan
[2]Department of Computer Science and Communication Engineering,
Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan

Abstract

In the design of succinct data structures, the main objective is to represent an object compactly while still allowing a number of fundamental operations to be performed efficiently. In this commentary, we consider succinct data structures for storing a bit vector B of length n. More precisely, in this setting, one needs to represent B using $n + \mathrm{o}(n)$ bits so that $rank$ and $select$ queries can be answered in $\mathrm{O}(1)$ time, where for any $i \in \{1, 2, \ldots, n\}$, $rank_0(B, i)$ is the number of 0s in the first i positions of B, $select_0(B, i)$ is the position in B of the ith 0 (assuming B contains at least i 0s), and $rank_1(B, i)$ and $select_1(B, i)$ are defined analogously. These operations are useful because bit vectors supporting $rank$ and $select$ queries are employed as a building block for many other more complex succinct data structures.

We first describe two succinct *indexing* data structures for supporting $rank$ and $select$ queries on B in which B is stored explicitly together with some auxiliary information. We then present some matching lower bounds. Finally, we discuss generalizations and related open problems for supporting $rank$ and $select$ queries efficiently on strings over non-binary alphabets.

1. Introduction

Let $B \in \{0, 1\}^n$ be a bit vector of length n. For any $i \in \{1, 2, \ldots, n\}$, let $B[i]$ denote the value of B at position i, and for any $i, j \in \{1, 2, \ldots, n\}$ with $i \leq j$, let $B[i..j]$ be the bit

*E-mail address: Jesper.Jansson@ocha.ac.jp
†Funded by the Special Coordination Funds for Promoting Science and Technology, Japan.
‡E-mail address: sada@csce.kyushu-u.ac.jp

vector consisting of $B[i]$, $B[i+1]$, ..., $B[j]$. (If $i > j$ then $B[i..j]$ is defined to be $\emptyset$.) Next, define the following operations:

- $rank_0(B, i)$ – Return the number of 0s in $B[1..i]$.
- $rank_1(B, i)$ – Return the number of 1s in $B[1..i]$.
- $select_0(B, i)$ – Return the position in B of the ith 0.
- $select_1(B, i)$ – Return the position in B of the ith 1.

In this commentary, we consider the problem of constructing a data structure for storing any given B such that $rank_0(B, i)$, $rank_1(B, i)$, $select_0(B, i)$, and $select_1(B, i)$ queries can be carried out efficiently. We focus on *indexing* data structures for B, where B is stored verbatim in n bits and one is allowed to use $\mathrm{o}(n)$ extra bits of storage (called the *index*) to efficiently support $rank$ and $select$ queries on B.

We assume the word-RAM model of computation with word length $w = \lceil \log n \rceil$ bits[1] in order to handle pointers to the data structure in constant time. In the word-RAM model, the CPU can perform logical operations such as AND and OR, and arithmetic operations such as addition, subtraction, multiplication, and division between two integers in the interval $[0,\ 2^w - 1]$ (w-bit integers) in constant time. The CPU can also read/write a w-bit integer from/to a specific memory cell in constant time; in other words, if B is a stored bit vector of length n, then for any given $i \in \{0, 1, \ldots, n - w\}$, $B[(i+1)..(i+w)]$ can be obtained in $\mathrm{O}(1)$ time.

The commentary is organized as follows: In Section 2., we outline how to construct in $\mathrm{O}(n)$ time an index for B of size $\mathrm{O}(n \log\log n / \log n) = \mathrm{o}(n)$ bits which allows each subsequent $rank$ or $select$ query to be answered in $\mathrm{O}(1)$ time. The presentation in Section 2. is based on [20] for $rank$ and [28] for $select$. Next, in Section 3., we state some lower bounds from [11] and [19] which match the upper bounds given in Section 2. Then, in Section 4., we discuss generalizations to non-indexing data structures as well as generalizations to non-binary vectors, and finally, in Section 5., we provide examples of other data structures that depend on efficient $rank$ and $select$ data structures for bit vectors and non-binary vectors, and mention some directions for further research.

2. Upper Bounds for Indexing Data Structures

Jacobson [15] presented a space-efficient indexing data structure for B which allows $rank$ and $select$ queries on B to be answered in $\mathrm{O}(1)$ and $\mathrm{O}(\log n)$ time, respectively, while requiring only $\mathrm{O}(n \log\log n / \log n)$ bits for the index. A series of improvements to Jacobson's data structure were made by Clark [4], Munro [20], Munro *et al.* [23], and Raman *et al.* [28], reducing the time needed to answer each $select$ query to $\mathrm{O}(1)$ while using an index of size $\mathrm{O}(n \log\log n / \log n)$ bits.

Below, we describe two simplified indexing data structures for B based on [20] for $rank$ and [28] for $select$, respectively, of size $\mathrm{O}(n \log\log n / \log n) = \mathrm{o}(n)$ bits. We

[1] Throughout this commentary, "log" denotes the base-2 logarithm and "$\log_\sigma$" denotes the base-σ logarithm.

remark that Golynski [11] recently gave a more sophisticated implementation for supporting both $rank$ and $select$ queries on B in $\mathrm{O}(1)$ time that uses an index of size $(1+\mathrm{o}(1))(n\log\log n/\log n)+\mathrm{O}(n/\log n)$ bits only.

To make the presentation more readable, we omit "$\lceil$", "$\rceil$", "$\lfloor$", and "$\rfloor$" symbols where obvious.

Also, we allow the last block in any partition into blocks to be smaller than the specified block size.

2.1. An Indexing Data Structure for $rank$ Queries (based on [20])

Conceptually divide the bit vector B into blocks of length $\ell = \log^2 n$ each, and call each such block a *large block*. Next, divide every large block into *small blocks* of length $s = \frac{1}{2}\log n$ each. Create auxiliary data structures for storing the values of $rank_1$ for the boundaries of these blocks as follows: Use an integer array $R_\ell[0..n/\ell]$ in which every entry $R_\ell[x]$ stores the number of 1's in $B[1..x\ell]$, and an integer array $R_s[0..n/s]$ in which every entry $R_s[y]$ stores the number of 1's in $B[(\lfloor\frac{y}{\ell/s}\rfloor\cdot\ell+1)..ys]$, i.e., the number of 1's in the yth small block plus the total number of 1's in all small blocks which belong to the same large block as small block y and which occur before small block y. The space needed to store R_ℓ is $\mathrm{O}(\frac{n}{\ell}\cdot\log n)=\mathrm{O}(\frac{n}{\log n})$ bits because each of its entries occupies $\mathrm{O}(\log n)$ bits, and the space needed to store R_s is $\mathrm{O}(\frac{n}{s}\cdot\log(\ell+1))=\mathrm{O}(\frac{n\log\log n}{\log n})$ bits because all of its entries are between 0 and ℓ.

To answer the query $rank_1(B,\,i)$ for any given $i\in\{1,2,\ldots,n\}$, compute $x=\lfloor\frac{i}{\ell}\rfloor$, $y=\lfloor\frac{i}{s}\rfloor$, and $z=i-ys$, and use the relation $rank_1(B,\,i)=rank_1(B,\,ys)+\sum_{q=1}^{z}B[ys+q]=R_\ell[x]+R_s[y]+\sum_{q=1}^{z}B[ys+q]$, where the first two terms are directly available from R_ℓ and R_s. To compute the third term in constant time, the following table lookup technique can be applied: In advance, construct a table $T_r[0..(2^s-1),\,1..s]$ in which each entry $T_r[i,\,j]$ stores the number of 1's in the first j bits of the binary representation of i. Then, whenever one needs to compute $\sum_{q=1}^{z}B[ys+q]$, first read the memory cell storing $B[(ys+1)..(ys+s)]$ (because $s<w$, this can be done in constant time), interpret this s-bit vector as an integer p, where $p\in\{0,1,\ldots,2^s-1\}$, and find the value $T_r[p,\,z]$ in the table. Hence, $rank_1(B,\,ys+z)$ can be computed in constant time. The size of the table T_r is $2^s\cdot s\cdot\log(s+1)=\mathrm{O}(\sqrt{n}\cdot\log n\cdot\log\log n)=\mathrm{o}(n)$ bits, and all of the auxiliary data structures R_ℓ, R_s, T_r may be constructed in $\mathrm{O}(n)$ time.

To compute $rank_0(B,i)$, no additional data structures are necessary because $rank_0(B,i)=i-rank_1(B,i)$. Therefore we have:

Theorem 1. *Given a bit vector of length n, after $\mathrm{O}(n)$ time preprocessing and using an index of size $\mathrm{O}(n\log\log n/\log n)$ bits, each $rank_1$ and $rank_0$ query can be answered in $\mathrm{O}(1)$ time.*

2.2. An Indexing Data Structure for $select$ Queries (based on [28])

Define $\ell=\log^2 n$ and construct an array storing the position of the $(i\ell)$th occurrence of a 1 in B for all $i=1,2,\ldots,\frac{n}{\ell}$. Regions in B between two consecutive positions stored in the array are called *upper blocks*. If the length of an upper block is at least $\log^4 n$, it is *sparse*. For every sparse upper block, store the positions of all its 1's explicitly in sorted

order. Since the number of such blocks is at most $\frac{n}{\log^4 n}$, the space required for storing the positions of all 1's in all sparse upper blocks is at most $\frac{n}{\log^4 n} \cdot \log^2 n \cdot \log n = \frac{n}{\log n}$ bits.

For every non-sparse upper block U, further divide it into *lower blocks* of length $s = \frac{1}{2}\log n$ each and construct a complete tree for U with branching factor $\sqrt{\log n}$ whose leaves are in one-to-one correspondence with the lower blocks in U. The height of the tree is at most 7, i.e., at most a constant, because the number of leaves is at most $\frac{\log^4 n}{s} = 2\log^3 n$. For each non-leaf node v of the tree, let C_v be an array of $\sqrt{\log n}$ integers such that $C_v[i]$ equals the number of 1's in the subtree rooted at the ith child of v. (All C_v-arrays can be computed in $O(n)$ time preprocessing.) The entire bit vector B contains at most $\frac{n}{s} = \frac{2n}{\log n}$ lower blocks, so the total number of nodes in all trees representing all the upper blocks is $O(\frac{n}{\log n})$ and furthermore, the total number of entries in all C_v-arrays is at most this much. Since the number of 1's in any tree is at most $\log^2 n$, every entry in a C_v-array can be stored in $\mathrm{O}(\log\log n)$ bits. Therefore, the total space needed to store all trees (including all the C_v-arrays) is $\mathrm{O}(\frac{n}{\log n} \cdot \log\log n)$ bits.

To answer the $select_1(B, i)$ query in constant time, first divide i by ℓ to find the upper block U that contains the ith 1, and check whether U is sparse or not. If U is sparse, the answer to the $select_1$ query is stored explicitly and can be retrieved directly. If U is not sparse, start at the root of the tree that represents U and do a search to reach the leaf that corresponds to the lower block with the jth 1, where j equals i modulo ℓ. At each step, it is easy to determine which subtree contains the jth 1 in $O(1)$ time by a table lookup using the C_v-array for the current node v, and then adjust j and continue to the next step. (For the lookup, use an $(\sqrt{\log n}+1)$-dimensional table T such that entry $T[c_1, c_2, \ldots, c_{\sqrt{\log n}}, j] = x$ if and only if the first subtree contains exactly c_1 1's, the second subtree contains exactly c_2 1's, etc. and the jth 1 belongs to the xth subtree. The space needed to store T is $o(n)$ bits because the index of T is encoded in $(\sqrt{\log n}+1) \cdot 2\log\log n \leq 0.5\log n$ bits for large enough n, so T has $O(2^{0.5\log n}) = O(n^{0.5})$ entries which each need $\log\log n$ bits.) Finally, after reaching a leaf and identifying the corresponding lower block, find the relative position of the jth 1 inside that lower block by consulting a global table of size $2^{1/2\log n} \cdot \frac{1}{2}\log n \cdot \log\log n = O(\sqrt{n}\log n\log\log n)$ bits which stores the relative position of the qth 1 inside a lower block for every possible binary string of length $\frac{1}{2}\log n$ and every possible query q in $\{1, 2, \ldots, \frac{1}{2}\log n\}$.

To answer $select_0$ queries, construct data structures analogous to those for $select_1$ described above. We obtain the following.

Theorem 2. *Given a bit vector of length n, after $\mathrm{O}(n)$ time preprocessing and using an index of size $\mathrm{O}(n\log\log n/\log n)$ bits, each $select_1$ and $select_0$ query can be answered in $\mathrm{O}(1)$ time.*

3. Lower Bounds for Indexing Data Structures

By applying two different techniques, one consisting of a reduction from a vector addition problem and the other one a direct information-theoretical argument involving reconstructing B from any given indexing data structure for B together with an appropriately defined binary string, Miltersen [19] proved the following theorem. (Recall that B is assumed to be

stored explicitly in addition to the bits used by the indexing data structure.)

Theorem 3. *[19] It holds that:*

1. *Any indexing data structure for $rank$ queries on B using word size w, index size r bits, and query time t must satisfy $2(2r + \log(w+1))tw \geq n\log(w+1)$.*
2. *Any indexing data structure for $select$ queries on B using word size w, index size r bits, and query time t must satisfy $3(r+2)(tw+1) \geq n$.*

In particular, for the case $t = \mathrm{O}(1)$ and $w = \mathrm{O}(\log n)$, Theorem 3 immediately implies the lower bounds $r = \Omega(n \log\log n/\log n)$ for $rank$ indexing data structures and $r = \Omega(n/\log n)$ for $select$ indexing data structures.

Using a counting argument based on binary choices trees, these lower bounds were strengthened by Golynski [11] as follows:

Theorem 4. *[11] If there exists an algorithm for either $rank$ or $select$ queries on B which reads $\mathrm{O}(\log n)$ different positions of B, has unlimited access to an index of size r bits, and is allowed to use unlimited computation power, then $r = \Omega(\frac{n\log\log n}{\log n})$.*

Hence, the upper bounds given in Theorems 1 and 2 are asymptotically optimal. Note that Theorem 4 is very general; it does not impose any restrictions on the running time or require the read positions of B to be consecutive for the lower bound to hold.

Golynski [11] also showed that:

Theorem 5. *[11] Suppose that B has exactly m positions set to 1 for some integer m. If there exists an algorithm for either $rank$ or $select$ queries on B which reads at most t different positions of B, has unlimited access to an index of size r bits, and is allowed to use unlimited computation power, then $r = \Omega(\frac{m}{t} \cdot \log t)$.*

4. Generalizations

The indexing data structures in Sections 2. and 3. assume that the bit vector B is always stored explicitly. However, the space used by this type of encoding is far from optimal if the number of 1's in B is much smaller than n, or close to n. This is because the number of bit vectors of length n having m 1's is $\binom{n}{m} \approx 2^{nH_0}$, where $H_0 = \frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}$ is the 0th order entropy of the bit vector, which may be much less than 2^n, the number of distinct bit vectors of length n. In fact, there exist data structures for $rank/select$ using only $nH_0 + \mathrm{O}(n\log\log n/\log n)$ bits to store B such that any consecutive $\mathrm{O}(\log n)$ bits of B can still be retrieved in constant time[2]:

Theorem 6. *[28] For a bit vector B of length n with m 1's, after $\mathrm{O}(n)$ time preprocessing and using $nH_0 + \mathrm{O}(n\log\log n/\log n)$ bits, where $H_0 = \frac{m}{n}\log\frac{n}{m} + \frac{n-m}{n}\log\frac{n}{n-m}$, each $rank_1$, $rank_0$, $select_1$, and $select_0$ query can be answered in $\mathrm{O}(1)$ time. Moreover, any consecutive $\mathrm{O}(\log n)$ bits of B can be retrieved in $\mathrm{O}(1)$ time.*

[2]Observe that these data structures do not store B directly, so to retrieve $\mathrm{O}(\log n)$ consecutive bits of B in $\mathrm{O}(1)$ time is no longer trivial.

The $rank$/$select$ data structures can be extended to non-binary vectors. A string S of length n over an alphabet $\mathcal{A}$ is a vector $S[1..n]$ such that $S[i] \in \mathcal{A}$ for $1 \leq i \leq n$. Let σ be the alphabet size, i.e., $\sigma = |\mathcal{A}|$. We assume that $\mathcal{A}$ is an integer alphabet of the form $\{0, 1, \ldots, \sigma - 1\}$ and that $\sigma \leq n$. (Without loss of generality, we further assume that σ is a power of 2.) Below, we consider succinct data structures for S supporting the following operations for any $i \in \{1, 2, \ldots, n\}$ and $c \in \{0, 1, \ldots, \sigma - 1\}$:

- $access(S, i)$ – Return $S[i]$.
- $rank_c(S, i)$ – Return the number of occurrences of c in $S[1..i]$.
- $select_c(S, i)$ – Return the position of the ith c in S.

S may be encoded in $n \log \sigma$ bits by the obvious representation using $\log \sigma$ bits for each position $S[i]$, but there exist other encodings which improve the $rank$ and $select$ query time complexities at the cost of increasing the space complexity and the time needed to retrieve $S[i]$ for any given $i \in \{1, 2, \ldots, n\}$. Hence, there is a trade-off between the size of a data structure and the $access$/$rank$/$select$ query times. Table 1 lists the performance of two straightforward data structures D_1 and D_2 (explained below) and three improved data structures proposed in [1, 12, 13].

Table 1. **The trade-off between the size (in bits) and the time needed to answer each $access$, $rank_c$, and $select_c$ query for various data structures. $|S|$ denotes the number of bits to encode S, H_0 is the 0th order entropy of S, and $\alpha = \log\log\sigma \log\log\log\sigma$.**

Reference	Size of data structure	$access$ time	$rank_c$ time	$select_c$ time
D_1 in Section 4.	$n(H_0 + \log e) + \sigma \cdot \mathrm{o}(n)$	$\mathrm{O}(\sigma)$	$\mathrm{O}(1)$	$\mathrm{O}(1)$
D_2 in Section 4.	$\lvert S\rvert + (\sigma + 1) \cdot \mathrm{o}(n)$	$\mathrm{O}(1)$	$\mathrm{O}(\log \sigma)$	$\mathrm{O}(\log \sigma)$
[13]	$nH_0 + \log \sigma \cdot \mathrm{o}(n)$	$\mathrm{O}(\log \sigma)$	$\mathrm{O}(\log \sigma)$	$\mathrm{O}(\log \sigma)$
[12]	$n \log \sigma + n \cdot \mathrm{o}(\log \sigma)$	$\mathrm{O}(\log\log \sigma)$	$\mathrm{O}(\log\log \sigma)$	$\mathrm{O}(1)$
[1]	$\lvert S\rvert + n \cdot \mathrm{o}(\log \sigma)$	$\mathrm{O}(1)$	$\mathrm{O}(\alpha \log\log \sigma)$	$\mathrm{O}(\alpha)$

The first straightforward data structure D_1 stores σ bit vectors $V_0, V_1, \ldots, V_{\sigma-1}$ of length n such that $V_c[i] = 1$ if and only if $S[i] = c$, along with $rank_1$ and $select_1$ indexing data structures for these bit vectors. Then $rank_c(S, i) = rank_1(V_c, i)$ and $select_c(S, i) = select_1(V_c, i)$, and therefore they can be obtained in constant time. On the other hand, $access$ requires $\mathrm{O}(\sigma)$ time because it must examine all of $V_0[i], V_1[i], \ldots, V_{\sigma-1}[i]$. Each bit vector V_c can be encoded in $\log \binom{n}{m_c} \approx m_c(\log e + \log \frac{n}{m_c})$ bits by Theorem 6, where m_c denotes the number of c's in S. In total, the space is $\sum_c \{m_c(\log e + \log \frac{n}{m_c}) + \mathrm{O}(n \log\log n / \log n)\} = n(H_0 + \log e) + \sigma \cdot \mathrm{O}(n \log\log n / \log n)$ bits. The second straightforward data structure D_2 stores S explicitly in $n \log \sigma$ bits. In addition, it stores a $rank_1$ and $select_1$ indexing data structure for each of the bit vectors $V_0, V_1, \ldots, V_{\sigma-1}$ of D_1. The bit vectors $V_0, V_1, \ldots, V_{\sigma-1}$ are not stored explicitly, so to answer $rank_c$ and $select_c$ queries, D_2 must have a method to compute any consecutive $\log n$ bits of V_c that are required by the indexing data structure for V_c. This can be done in $O(\log \sigma)$ time by repeating the following steps $2 \log \sigma$ times, each time obtaining $\frac{1}{2} \log_\sigma n$ bits of V_c: In

$O(1)$ time, read $\frac{1}{2}\log n$ consecutive bits from S and put them in a bit vector r. To find the $\frac{1}{2}\log_\sigma n$ bits of V_c that correspond to r, let s be the bit vector of length $\frac{1}{2}\log n$ consisting of $\frac{1}{2}\log_\sigma n$ copies of the length-($\log\sigma$) pattern $000\ldots01$, let t be s multiplied by c, and and let u be the bitwise exclusive-or between r and t. Note that for any non-negative integer i, the length-($\log\sigma$) pattern of u starting at position $i\cdot\log\sigma$ equals $000\ldots00$ if and only if the corresponding position in S contains the symbol c. Finally, look up entry u in a table having $2^{1/2\log n}=\sqrt{n}$ entries to obtain a bit vector of size $\frac{1}{2}\log_\sigma n$ containing a 1 in position i if and only if $u[(i\cdot\log\sigma)..((i+1)\cdot\log\sigma)-1]=000\ldots00$. Thus, $rank_c$ and $select_c$ take $\mathrm{O}(\log\sigma)$ time. The $access$ query takes constant time because S is explicitly stored. The total space is that of storing S plus $\sigma\cdot\mathrm{O}(n\log\log n/\log n)$ bits for the $rank_1$ and $select_1$ indexing data structures, plus the size of the lookup table which is $\sqrt{n}\cdot\frac{1}{2}\log_\sigma n=\mathrm{o}(n)$ bits.

In Table 1, $|S|$ denotes the number of bits to encode S. It is $n\log\sigma$ if S is not compressed; however, it can be reduced by applying a compression algorithm which supports instant decoding [8]:

Theorem 7. *There exists a succinct data structure for storing a string $S[1..n]$ over an alphabet $\mathcal{A}=\{0,1,\ldots,\sigma-1\}$ in*

$$nH_k+\mathrm{O}\left(\frac{n(\log\log_\sigma n+k\log\sigma)}{\log_\sigma n}\right)$$

bits for any $k\geq 0$, where H_k is the kth order empirical entropy of S, such that any substring of the form $S[i\ldots i+\mathrm{O}(\log_\sigma n)]$ with $i\in\{1,2,\ldots,n\}$ can be decoded in $\mathrm{O}(1)$ time on the word-RAM.

By using this theorem, we can compress S into $nH_k+\mathrm{o}(n\log\sigma)$ bits. Furthermore, we can regard the compressed data as an uncompressed string. Therefore the query time in Table 1 does not change.

5. Concluding Remarks

Succinct data structures that support efficient $rank$ and $select$ queries on bit vectors and non-binary vectors are important because they form the bases of several other more complex data structures. Some examples include succinct data structures for representing trees [2, 6, 10, 16, 17, 22, 23], graphs [3, 15], permutations and functions [21, 24], text indexes [7, 14, 29, 30], prefix or range sums [26], and polynomials and others [9]. In these data structures, a typical use of $rank$ and $select$ queries on bit vectors is to encode pointers to blocks of data. For example, suppose that to compress some data we partition it into blocks, compress each block independently into a variable numbers of bits, and concatenate the result into a bit vector C. Then we can use another bit vector $B[1..|C|]$ such that $B[i]=1$ if and only if the ith bit of C is the starting position of a block, and apply $select_1$ queries on B to find the correct starting and ending positions in C when decompressing the data corresponding to a particular block. Some directions for further research include dynamization to support operations that allow B to be modified online [27], proving lower bounds on the size of succinct data structures [5, 11, 19] (note that the lower bounds shown

in Section 3. hold only if the bit vector is stored explicitly using n bits, and thus do not hold for bit vectors stored in a *compressed* form), and practical implementation [25]. Although the sizes of the known indexing data structures for bit vectors are asymptotically optimal, the $o(n)$ additional space needed by an index is often too large for real data and cannot be ignored. Therefore, for practical applications, it is crucial to develop other implementations of succinct data structures. Another open problem involves $access$/$rank$/$select$ operations on non-binary vectors. No single data structure listed in Table 1 supports constant time $access$, $rank$ and $select$ queries. What are the best possible lower and upper bounds on the number of bits required to achieve this? Finally, a related topic is *compressed suffix arrays* [14], which are data structures for efficient substring searches. The suffix array [18] uses $n \log n$ bits for a string of length n with alphabet size σ, while the compressed suffix array uses only $\mathrm{O}(n \log \sigma)$ bits, which is linear in the string size. On the other hand, the compressed suffix array does not support constant time retrieval of an element of the suffix array. An important open problem is to establish whether there exists a data structure using linear space and supporting constant time retrieval.

References

[1] J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2007), pages 680–689, 2007.

[2] 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.

[3] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2003), pages 679–688, 2003.

[4] D. Clark. *Compact Pat Trees*. PhD thesis, The University of Waterloo, Canada, 1996.

[5] E. D. Demaine and A. López-Ortiz. A Linear Lower Bound on Index Size for Text Retrieval. *Journal of Algorithms*, **48**(1):2–15, 2003.

[6] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science* (FOCS 2005), pages 184–196, 2005.

[7] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, **52**(4):552–581, 2005.

[8] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, **372**(1):115–121, 2007.

[9] A. Gál and P. B. Miltersen. The cell probe complexity of succinct data structures. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming* (ICALP 2003), volume 2719 of *Lecture Notes in Computer Science*, pages 332–344. Springer-Verlag, 2003.

[10] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching* (CPM 2004), volume 3109 of *Lecture Notes in Computer Science*, pages 159–172. Springer-Verlag, 2004.

[11] A. Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, **387**(3):348–359, 2007.

[12] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2007), pages 368–373, 2006.

[13] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2003), pages 841–850, 2003.

[14] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, **35**(2):378–407, 2005.

[15] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science* (FOCS 1989), pages 549–554, 1989.

[16] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct Representation of Ordered Trees. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2007), pages 575–584, 2007.

[17] H.-I. Lu and C.-C. Yeh. Balanced Parentheses Strike Back. To appear in *ACM Transactions on Algorithms*, 2008.

[18] U. Manber and G. Myers. Suffix arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, **22**(5):935–948, October 1993.

[19] P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2005), pages 11–12, 2005.

[20] J. I. Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science* (FSTTCS 1996), volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer-Verlag, 1996.

[21] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming* (ICALP 2003), volume 2719 of *Lecture Notes in Computer Science*, pages 345–356. Springer-Verlag, 2003.

[22] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, **31**(3):762–776, 2001.

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

[24] J. I. Munro and S. S. Rao. Succinct Representations of Functions. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming* (ICALP 2004), volume 3142 of *Lecture Notes in Computer Science*, pages 1006–1015. Springer-Verlag, 2004.

[25] D. Okanohara and K. Sadakane. Practical Entropy-Compressed Rank/ Select Dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments* (ALENEX 2007), 2007.

[26] C. K. Poon and W. K. Yiu. Opportunistic Data Structures for Range Queries. In *Proceedings of Computing and Combinatorics, 11th Annual International Conference* (COCOON 2005), volume 3595 of *Lecture Notes in Computer Science*, pages 560–569. Springer-Verlag, 2005.

[27] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Proceedings of Algorithms and Data Structures, 7th International Workshop* (WADS 2001), volume 2125 of *Lecture Notes in Computer Science*, pages 426–437. Springer-Verlag, 2001.

[28] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, **3**(4):Article 43, 2007.

[29] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, **48**(2):294–313, 2003.

[30] K. Sadakane. Compressed Suffix Trees with Full Functionality. *Theory of Computing Systems*, **41**(4):589–607, 2007.