PAPER

Fast Computation of Rank and Select Functions for Succinct Representation*

Joong Chae NA[†], ***Member***, **Ji Eun KIM**[††], **Kunsoo PARK**[†††], ***and*** **Dong Kyue KIM**[††††a)], ***Nonmembers***

SUMMARY Succinct representation is a space-efficient method to represent n discrete objects using space close to the information-theoretic lower bound. In order to directly access the ith object of succinctly represented data structures in constant time, two fundamental functions, `rank` and `select`, are commonly used. In this paper we propose two implementations supporting `rank` and `select` in constant time for non-compressed bit strings. One uses $O(n \log\log n/\sqrt{\log n})$ bits of extra space and the other uses $n + O(n \log\log n/\log n)$ bits of extra space in the worst case. The former is rather a theoretical algorithm and the latter is a practical algorithm which works faster and uses less space in practice.

key words: *succinct representation, rank function, select function*

1. Introduction

To analyze the performance of data structures, the processing time and the amount of used space are measured in general. With the rapid proliferation of information, it is increasingly important to focus on the space requirements of data structures. Traditionally, discrete objects such as elements of sets or arrays, nodes of trees, vertices and edges of graphs, etc., are represented as integers which are the indices of elements in a consecutive memory block or values of logical addresses in main memory. If we store n discrete objects in this way, they occupy $O(n)$ words, i.e., $O(n \log n)$ bits.

Recently, a method to represent n objects using space close to the information-theoretic lower bound, which is called *succinct representation*, was developed. Various succinct representation techniques have been developed to represent data structures such as sets, static and dynamic dictionaries [1], [2] trees [3], graphs [4], [5], permutations [6], and functions [7]. Moreover, succinct representation is indispensible to develop compressed index data structures [8]–[11]. If we use these succinctly represented data structures, we can perform very fast pattern searching using little space.

Most succinct representations use `rank` and `select` for a bit-string as their basic functions. The rank and select functions are defined as follows. Given a static bit-string A,

- $\texttt{rank}_A(x)$: Counts the number of 1's up to and including the position x in A.
- $\texttt{select}_A(y)$: Finds the position of the yth 1 bit in A.

For example, if $A = 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0$, $\texttt{rank}_A(5) = 2$ and $\texttt{select}_A(2) = 4$.

The study of the implementation of `rank` and `select` was started by Jacobson [1], [5], who proposed a data structure supporting the functions in $O(\log n)$ time. Jacobson constructs a two-level directory structure and performs a direct access and a binary search on the directory for `rank` and `select`, respectively. Later, Munro [12] and Clark [13] improved Jacobson's result to support `rank` and `select` in constant time by adding lookup tables to complement the two-level directory of Jacobson's. Their data structure requires $n + O(n \log\log n/\log n)$ bits for `rank` and $n + O(n/\log\log n)$ bits for `select` (including the original bit-string). Pagh [14] and Raman et al. [15] studied succinct representations of compressible strings. Their data structure supports `rank` and `select` in constant time and requires $nH_0(A) + O(n \log\log n/\log n)$ bits, where $H_0(A)$ is the zeroth-order entropy of A. Their data structure can also be applied to uncompressed bit-strings. González et al. [16] studied the practicality of these solutions for incompressible sequences. Miltersen [17] studied lower bounds on the space and query time of `rank` and `select`. For constant query time, he obtained a lower bound of $\Omega(n \log\log n/\log n)$ space for auxiliary data structures of `rank` and a lower bound of $\Omega(n/\log n)$ space for those of `select`. Recently, Okanohara and Sadakane [18] proposed practical implementations of `rank` and `select`, which do not support constant query time.

In this paper we consider implementations of `rank` and `select` (especially, focusing on `select`) supporting worst-case $O(1)$ query time for non-compressed bit strings. The `rank` and `select` implementations use hierarchical directory structures, called *rank-directory* and *select-directory*, respectively. In rank-directories, a bit-string is partitioned into substrings with the same number of bits, and thus rank-directories consist of regular tables of uniform sizes. Meanwhile, in select-directories, a bit-string is partitioned into substrings with the same number of 1's, and thus select-directories consist of irregular tables of diverse sizes. In previous algorithms [13], [15], the key idea for handling this irregularity in select-directories is to divide substrings into

Manuscript received January 8, 2009.
Manuscript revised June 4, 2009.

[†]The author is with the Department of Computer Science and Engineering, Sejong University, Seoul 143–747, South Korea.
[††]The author is with Agency for Defense Development, Daejeon 305–600, South Korea.
[†††]The author is with the School of Computer Science and Engineering, Seoul National University, Seoul 151–742, South Korea.
[††††]The author is with the Department of Electronics and Communications Engineering, Hanyang University, Seoul 133–791, South Korea.

*The preliminary version of this paper was presented in WEA'2005.

a) E-mail: dqkim@hanyang.ac.kr (corresponding author)

DOI: 10.1587/transinf.E92.D.2025

Copyright © 2009 The Institute of Electronics, Information and Communication Engineers

two classes, *dense* substrings and *sparse* substrings, and to apply different techniques. For this reason, the select query time varies significantly according to the distribution of 0's and 1's in bit-strings even though the differences are bounded by a constant. This kind of phenomenon was also reported in [16].

In this paper we propose two alternative implementations of `rank` and `select`, whose query time is uniform regardless of distribution of 1's. Our contributions are as follows:

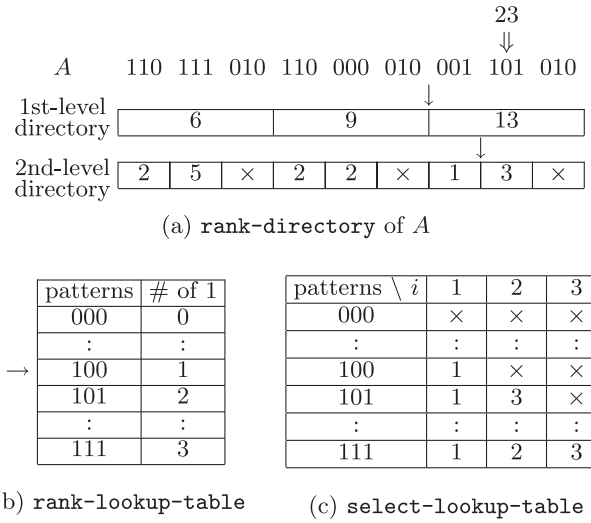
- Algorithm I: Its main idea is to transform the given bit-string A into a bit-string where 1's are distributed quite regularly. Algorithm I uses a select-directory but its structure is simpler and more regular than the select-directories of previous algorithms. Algorithm I uses at most $n + O(n \log\log n/\sqrt{\log n})$ bits for `select`, which is better than Clark's $n + O(n/\log\log n)$ bits but worse than Raman et el.'s $n + O(n \log\log n/\log n)$ bits
- Algorithm II: It is a more practical algorithm. Its query time is faster than that of Algorithm I. It needs $2n + O(n \log\log n/\log n)$ bits in the worst case. When it is necessary to support only `select`, this algorithm needs $n + O(n \log\log n/\log n)$ bits in the worst case. The main advantage of Algorithm II is that it uses only rank directories for `select` as well as for `rank`. Thus, its data structure is the most regular and so it is the most suitable for applying byte-based implementation method.

We analyze the performance of our algorithms and compare ours with previous algorithms on various random strings. Our algorithms take uniform query time regardless of distribution of 1's. Especially Algorithm II is fastest in practice. Moreover, our algorithms are easy to implement because data structures are regular. In our experiment, we implemented [13] but not [15], because [13] and [15] follow the same idea and [13] is easier to implement. (The algorithm of [15] is quite complicated since it is originally designed for compressed bit-strings.)

This paper is organized as follows. We first define basic data structures and introduce previous algorithms in Sect. 2. We present new algorithms for `select` in Sects. 3 and 4 and describe the experimental results in Sect. 5. We conclude in Sect. 6.

2. Preliminaries

In this section we introduce basic data structures that are used in previous algorithms as well as in our algorithms and describe briefly Clark's algorithm [13]. Let A be a static bit-string of length n. We denote the ith bit by $A[i]$ and the substring $A[i]A[i+1]\cdots A[j]$ by $A[i..j]$. For simplicity, we assume that $\sqrt{\log n}$, $\log n$ and $\log^2 n$ are integers, where $\log^2 n$ means $(\log n)^2$. We assume the word RAM model. On this model, n can be represented by one machine word, and arithmetic operations and memory accesses for $O(\log n)$-bits word can be done in constant time.



(a) `rank-directory` of A

(b) `rank-lookup-table` (c) `select-lookup-table`

Fig. 1 Examples of data structures.

We first define a hierarchical directory structure for `rank`. Given a bit-string A, we define `rank-directory` of A as the following two-level directory (see Fig. 1 (a)):

- We partition A into *big blocks* of size $\log^2 n$. Each big block of the 1st-level directory records the accumulated number of 1's from the first big block. That is, the ith entry contains the number of 1's in $A[1..i\log^2 n]$ for $1 \le i \le \lfloor n/\log^2 n \rfloor$.
- We partition A into *small blocks* of size $\log n$. Each small block of the 2nd-level directory records the accumulated number of 1's from the first small block within each big block. That is, the ith entry contains the number of 1's in $A[i'\log^2 n + 1..i'\log^2 n + i\log n]$ for $1 \le i \le \lfloor n/\log n \rfloor$, where $i' = \left\lfloor i\log n/\log^2 n \right\rfloor$.

Lemma 1: [13] Given a bit-string A of length n, `rank-directory` of A can be stored in $O(n \log\log n/\log n)$ bits.

We define lookup tables that enable us to compute `rank` and `select` in constant time. For some integer $c > 1$ ($c = 2$ suffices), `rank-lookup-table` is the table where an entry contains the number of 1's in each possible bit pattern of length $(\log n)/c$. Similarly, `select-lookup-table` is the table where an entry contains the position of the ith 1 bit in each possible bit pattern of length $(\log n)/c$, for $1 \le i \le \log n/c$. Figure 1 (b) and (c) show `rank-lookup-table` and `select-lookup-table` for patterns of length 3, respectively.

Lemma 2: [13] `rank-lookup-table` and `select-lookup-table` can be stored in $O(n^{1/c}\log\log n)$ bits and $O(n^{1/c}\log n\log\log n)$ bits, respectively, where $c > 1$ is a constant.

We describe briefly Clark's algorithm. For $\texttt{rank}_A$, Clark used `rank-directory` of A and `rank-lookup-table`. To get $\texttt{rank}_A(x)$, one first computes $\texttt{rank}_A(i)$ for an ending position i of the $\lfloor x/\log n \rfloor$th small block using

`rank-directory`, and then counts the number of 1's in remaining x mod $\log n$ bits, which can be found by adding at most c entries in `rank-lookup-table` after masking out unwanted trailing bits.

EXAMPLE 1: For bit-string A in Fig. 1, let $\log^2 n = 9$ and $\log n = 3$. Suppose that we want to compute $\texttt{rank}_A(23)$. We first get $\texttt{rank}_A(21) = 10$ by adding the value 9 in the 2nd entry of the 1st-level directory and the value 1 in the 7th entry of the 2nd-level directory, which give the numbers of 1's in $A[1..18]$ and $A[19..21]$, respectively. Then, we get a bit-pattern 100 by mask out $A[22..24] = 101$ with 110 and get the number of 1's in 100 using `rank-lookup-table`. So, we get $\texttt{rank}_A(23) = 11$ by adding 1 to 10. □

For $\texttt{select}_A$, Clark partitioned A into blocks so that each block has the same number of 1's. Differently from `rank-directory`, the sizes of the blocks are diverse. Thus, Clark divided the blocks into two classes, dense blocks and sparse blocks, and handled two classes with different techniques. For a dense block, a similar partition is performed one more time, and for a sparse block, all answers of `select` are recorded explicitly. Finally, `select` can be computed by scanning a small number of bits using lookup tables. We omit the detailed algorithm and refer the reader to [13].

3. Algorithm I

In this section we present our first algorithm. This algorithm also adopts the approach of using multi-level directories and lookup tables. The difficulty of developing an algorithm for `select` results from the irregular distribution of 1's. While Clark's algorithm overcomes the difficulty by classifying the subranges of directories into two groups: dense one and sparse one, we do it by transforming A into a bit-string where 1's are distributed quite regularly.

3.1 Definitions

Given a bit-string S of length m, we divide S into blocks of size b. There are two kinds of blocks. One is a block where all elements are 0 and the other is a block where there is at least one 1. We call the former a *zero-block* and the latter a *nonzero-block*.

- The *contracted* string of S is defined as a bit-string S_c of length m/b such that $S_c[i] = 0$ if the ith block of S is a zero-block, $S_c[i] = 1$ otherwise.
- The *extracted* string of S is defined as a bit-string S_e which is formed by concatenating nonzero-blocks of S in order. Hence, the length of S_e is m in the worst case, and the distance between the ith 1 bit and the jth 1 bit is at most $(j - i + 1)b - 1$ for $i < j$.
- The *delimiter* string of S is defined as a bit-string S_d such that $S_d[i] = 0$ if the ith 1 bit and the $(i-1)$st 1 bit of S are contained in the same block, and $S_d[i] = 1$ otherwise. We define $S_d[1] = 1$. Note that the length of S_d is equal to the number of 1's in S and so it is m in the worst case. The value of $\texttt{rank}_{S_d}(i)$ means the number of nonzero blocks up to the block (including itself) containing the ith 1 bit of S.

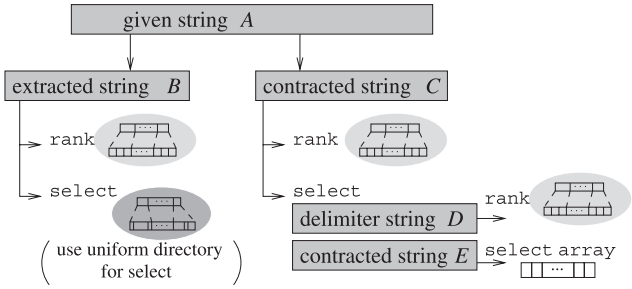


Fig. 2 Structure of Algorithm I.

EXAMPLE 2: Bit-strings S, S_c, S_e and S_d. We assume that we divide S into blocks of size 4.

S	0 1 1 1	0 0 0 0	0 0 1 0	1 1 0 1
S_c	1	0	1	1
S_e	0 1 1 1		0 0 1 0	1 1 0 1
S_d	1 0 0		1	1 0 0

□

3.2 Skeleton of Algorithm I

Let B be the extracted string of A, and C be the contracted string of A when dividing A into blocks of size $\sqrt{\log n}$. We transform A into B and C, and we will compute $\texttt{rank}_A$ and $\texttt{select}_A$ using B and C. We first consider the properties of B and C and describe how to compute $\texttt{rank}_A$ and $\texttt{select}_A$ using `rank`'s and `select`'s of B and C. In the next section we will present algorithms and data structures for `rank`'s and `select`'s of B and C. Figure 2 shows the overall structure of Algorithm I.

Recall that blocks of B contain at least one 1 bit, and thus the distance between the ith 1 bit and the jth 1 bit in B is at most $(j - i + 1)\sqrt{\log n} - 1$ for $i < j$. Bit-string C represents a mapping between A and B. The length of C is $n/\sqrt{\log n}$ and the length of B is n in the worst case.

EXAMPLE 3: Bit-strings A, B and C. We assume that $\sqrt{\log n} = 5$.

						↓	⇓
A	01101	00000	00000	11010	00000	01001	00100
C	1	0	0	1	0	1	1
B	01101	11010	01001 ↑	00100 ⇑			

□

We describe how to compute $\texttt{rank}_A(x)$ and $\texttt{select}_A(y)$ using B and C. For computing $\texttt{rank}_A(x)$, we find the index x' of B which corresponds to $A[x]$ and compute $\texttt{rank}_B(x')$. Let x_b be the block number of A which contains $A[x]$ and x_p be the position of $A[x]$ in the x_bth block of A, that is, $x_b = \lceil x/\sqrt{\log n} \rceil$ and $x_p = x - (x_b - 1)\sqrt{\log n}$. Then,

$$x' = \texttt{rank}_C(x_b - 1) \times \sqrt{\log n} + x_p \times C[x_b] \text{ and}$$
$$\texttt{rank}_A(x) = \texttt{rank}_B(x').$$

For computing $\texttt{select}_A(y)$, we compute $\texttt{select}_B(y)$ and

find an index of A which corresponds to $B[\texttt{select}_B(y)]$. Let s_b be the block number of B which contains the yth 1 bit, that is, $s_b = \lceil \texttt{select}_B(y)/\sqrt{\log n}\ \rceil$. Then,

$$\texttt{select}_A(y) = \texttt{select}_B(y) + (\texttt{select}_C(s_b) - s_b)\sqrt{\log n}.$$

EXAMPLE 4: For A of Example 3, suppose that we want to compute $\texttt{rank}_A(28)$ and $\texttt{select}_A(9)$. For $\texttt{rank}_A(28)$, we get $x_b = 6$ and $x_p = 3$, that is, $A[28]$ is the 3rd bit in the 6th block of A. Because $\texttt{rank}_C(5) = 2$, we get $x' = 2 \times 5 + 3 = 13$, so $\texttt{rank}_A(28) = \texttt{rank}_B(13) = 6$. For $\texttt{select}_A(9)$, we get $\texttt{select}_B(9) = 18$, so $s_b = 4$. Because $\texttt{select}_C(4) = 7$, we get $\texttt{select}_A(9) = 18 + (7 - 4) \times 5 = 33$. □

3.3 Ranks and Selects for B and C

We describe algorithms and data structures for `rank`'s and `select`'s of B and C. For $\texttt{rank}_B$, we build `rank-directory` of B and use `rank-lookup-table`. For $\texttt{rank}_C$, we build `rank-directory` of C and use `rank-lookup-table`. We can compute $\texttt{rank}_B(x)$ and $\texttt{rank}_C(x)$ using these data structures in constant time as in Sect. 2.

3.3.1 Select for Extracted String B

For $\texttt{select}_B$, we use lookup tables and a two-level directory which is similar to `rank-directory` because 1's in B are distributed regularly.

- The 1st-level directory records the position of every $\log^2 n$'th 1 bit in B. This directory has at most $n/\log^2 n$ entries and each entry requires $\log n$ bits. Thus the space of this directory is $n/\log^2 n \times \log n$ bits.
- The 2nd-level directory records the position of every $\sqrt{\log n}$'th 1 bit in the ranges of the 1st-level directory. This directory has at most $n/\sqrt{\log n}$ entries and each entry requires $\log(\log^2 n \times \sqrt{\log n})$ bits because all blocks of size $\sqrt{\log n}$ have at least one 1 bit. Thus the space of this directory is $n/\sqrt{\log n} \times 5/2 \times \log\log n$ bits.
- We also maintain `rank-lookup-table` and `select-lookup-table` for bit strings of length $(\log n)/c$.

We can compute $\texttt{select}_B(y)$ using the above directory and the lookup tables. Let $y' = y \bmod \sqrt{\log n}$ and $p = \texttt{select}_B(y - y')$, which can be computed using the above directory because $y - y'$ is a multiple of $\sqrt{\log n}$. Since the distance between the $(y - y')$th 1 bit and the yth 1 bit is at most $(y' + 1)\sqrt{\log n} - 1 < \log n$, the yth 1 bit in B exists in $B[p + 1..p + \log n]$ and the remaining thing is to find the position of the y'th 1 bit in $B[p+1..p+\log n]$. It can be done in constant time using the lookup tables. We divide $B[p+1..p+\log n]$ into c pieces of length $\log n/c$. We first find the piece containing the y'th 1 bit by accessing `rank-lookup-table` at most c times, and then we get the position of the y'th 1 bit in the piece using `select-lookup-table`.

3.3.2 Select for Contracted String C

We use a different approach for $\texttt{select}_C$ because the range of contiguous 0's is not bounded in C. When dividing C into blocks of size $\log n$, let D and E be the delimiter string and the contracted string of C, respectively. Since the length of C is $n/\sqrt{\log n}$, the length of D is $n/\sqrt{\log n}$ in the worst case and the length of E is $n/\log n\sqrt{\log n}$.

EXAMPLE 5: Bit-strings C, D and E. We assume that $\log n = 4$.

												↓				
C	0	1	1	1	0	0	0	0	0	0	1	0	1	0	1	1	0	1	0	1
E		1				0				1				1				1		
D		1	0	0							1		1		0	0		1		0

□

We describe how to compute $\texttt{select}_C(y)$ using D and E. We first compute the block number u containing the yth 1 bit in C using $\texttt{rank}_D$ and $\texttt{select}_E$. Notice that $\texttt{rank}_D(y)$ means the number of nonzero blocks up to the block (including itself) containing the yth 1 bit of C and $\texttt{select}_E(k)$ represents the block number of the kth nonzero block. Hence,

$$u = \texttt{select}_E(\texttt{rank}_D(y)).$$

Let r be the number of 1's in the first $(u - 1)$ blocks of C, i.e.,

$$r = \texttt{rank}_C((u - 1) \times \log n).$$

Let s be the position of the $(y - r)$'th 1 bit in the uth block of C, which can be found by scanning the uth block using the lookup tables. Then,

$$\texttt{select}_C(y) = (u - 1) \times \log n + s.$$

EXAMPLE 6: Suppose that we want to find $\texttt{select}_C(6)$ in Example 5. Because $\texttt{rank}_D(6)$ is 3 and $\texttt{select}_E(3)$ is 4, the 6th 1 bit is in the 4th block of C. We can also know that the first 3 blocks of C contains four 1 bits using $\texttt{rank}_C$ and that the position of the 2nd 1 bit in the 4th block is 3 using lookup tables. So, we get $\texttt{select}_C(6) = 3 \times 4 + 3 = 15$. □

We describe data structures for $\texttt{rank}_D$ and $\texttt{select}_E$.

- For $\texttt{rank}_D$, we build `rank-directory` of D and use `rank-lookup-table`.
- For $\texttt{select}_E$, we construct an array whose ith entry records $\texttt{select}_E(i)$ naively. This array has at most $n/(\log n\sqrt{\log n})$ entries and each entry requires $\log(n/(\log n\sqrt{\log n}))$ bits. Thus the space of this array is $O(n/\sqrt{\log n})$. We call this array the *select array* of E. Note that we do not need to maintain bit-string E.

Theorem 1: Algorithm I performs $\texttt{rank}_A$ and $\texttt{select}_A$ in constant time using at most $n + O(n \log\log n/\log n)$ bits and at most
$n + O(n \log\log n/\sqrt{\log n})$ bits, respectively.

Proof: We transform A into B and discard A. That is, we maintain B using the space charged for A. Although A is discarded, we can easily restore a substring of A using B

and C. Furthermore, we can know in constant time what a character $A[i]$ is, which is the most fundamental operation in bit-strings.

All other bit-strings take $O(n/\sqrt{\log n})$ bits. The directory for $\texttt{select}_B$ takes $O(n \log\log n/\sqrt{\log n})$ bits. All other auxiliary data structures, directories and lookup tables take $O(n \log\log n/\log n)$ bits. It has already been shown that the algorithm has constant retrieval time. □

4. Algorithm II

In this section we describe Algorithm II, which is simpler and more practical than Algorithm I but theoretically uses at most $2n + O(n \log\log n/\log n)$ bits. We also propose a new byte-based implementation method reducing a waste of space, which is suitable for Algorithm II because its data structures are regular.

4.1 Description of Algorithm II

We only describe the algorithm for $\texttt{select}_A$ since the algorithm for $\texttt{rank}_A$ is the same as Clark's. Figure 3 shows the overall structure of Algorithm II. The approach for $\texttt{select}_A$ is similar to the one for $\texttt{select}_C$ in Sect. 3.3.2. When dividing A into blocks of size $\log n$, let P and Q be the delimiter string and the contracted string of A, respectively. The length of P is n in the worst case and the length of Q is $n/\log n$. Note that P (resp. Q) is to A what D (resp. E) is to C in Sect. 3.3.2.

To compute $\texttt{select}_A(y)$, we use the same algorithm as the one for computing $\texttt{select}_C(y)$. Thus, we need to compute $\texttt{rank}_P$ and $\texttt{select}_Q$. For $\texttt{rank}_P$, we build `rank-directory` of P. For $\texttt{select}_Q$, we use a new approach, which uses little space in practice. The reason why we don't use the select array of Q (i.e., naive implementation of $\texttt{select}_Q$) is because it takes nearly n bits in practice as well as in the worst case.

We describe our approach for $\texttt{select}_Q$. We call a bundle of contiguous 0's a *clump*. In Example 7, there are 4 clumps in Q. We define the *clump-delimiter* string R of Q as follows: $R[i] = 0$ if the ith 1 bit of Q is adjacent to the $(i-1)$st 1 bit, and $R[i] = 1$ otherwise. We define $R[1]$ as 0 if $Q[1] = 1$, and 1 otherwise. See Example 7. The length of R is $n/\log n$ in the worst case since the length of Q is $n/\log n$. We construct the following auxiliary data structures.

- We construct a data structure for $\texttt{rank}_R$. The value of $\texttt{rank}_R(i)$ means how many clumps there are in front of the ith 1 bit of Q. In Example 7, there are 3 clumps in front of the 7th 1 bit.
- We construct an array where the ith entry represents the accumulated number of 0's up to the ith clump (including itself) in Q. We call it the *clump* array of Q. Note that we do not need to maintain bit-string Q.

EXAMPLE 7: Bit-strings Q and R, and the clump array of Q.

												↓				
Q:	0	0	1	1	1	0	1	1	0	0	1	1	1	1	0	1
R:			1	0	0		1	0			1	0	0	0		1

The clump array of Q

index	1	2	3	4
	2	3	5	6

□

In order to compute $\texttt{select}_Q(i)$, we first find how many clumps there are in front of the ith 1 bit of Q using $\texttt{rank}_R(i)$ and get the number j of 0's in front of the ith 1 bit using the clump array. Then $\texttt{select}_Q(i) = i + j$.

Theorem 2: Algorithm II performs $\texttt{rank}_A$ and $\texttt{select}_A$ in constant time using $n + O(n \log\log n/\log n)$ bits and at most $2n + O(n \log\log n/\log n)$ bits (including the original bit-string), respectively.

Proof: We omit the case of $\texttt{rank}_A$ since the algorithm for $\texttt{rank}_A$ is the same as Clark's. Auxiliary data structures for $\texttt{select}_A$ are bit-strings P and R, their rank-directories and the lookup tables, and the clump array of Q. The data structures except string P and the clump array of Q takes $O(n \log\log n/\log n)$ bits.

We show that string P and the clump array of Q takes at most n bits in total. Let p be the number of 1's in A, i.e., the length of P is p. Since Q is the contracted string of A whose size is $n/\log n$, the number of 0's in Q is at most $(n-p)/\log n$. Therefore, the clump array of Q has at most $(n-p)/\log n$ entries and each entry requires $\log(n/\log n)$ bits. The clump array requires at most $n - p$ bits. Hence we can get the theorem. □

4.2 Byte-Based Implementation of Algorithm II

We present an efficient byte-based implementation method of Algorithm II. The directories and lookup tables are based on bits, while atomic units in modern computers are not bits but bytes. Therefore, we need *bit-operations* such as *bitwise-and, bitwise-or*, and *shift* in order to get the values of entries. It causes inefficiency in time. One method avoiding such inefficiency is to allocate space to entries by the units of bytes. For example, we allocate 2 bytes to an entry which requires 12 bits. However, this method may waste much space. We present a *byte-based implementation* reducing waste of space and avoiding bit-operations. A similar method was used in [16]. This method is applied to Q

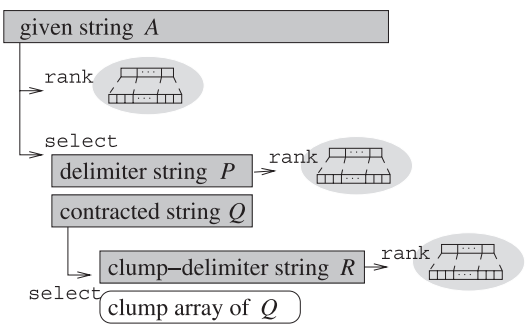


Fig. 3 Structure of Algorithm II.

and R as well as A. We assume that n is less than 2^{32}, the maximum value represented by a word in 32-bit machines. However, our method can be extended to the case of $n \geq 2^{32}$.

The key idea of our method is that ranges of directories and lookup tables are adjusted to multiples of 8. The following data structure, which we use to implement Algorithm II, is an instance of our implementation method.

- The 1st-level directory contains `rank` for every multiple of 2^8. Each entry requires at most 32 bits. Particularly, the ranges of the 1st-level directory is adjusted to 2^k, where k is a multiples of 8 in order to allocate space to entries of the 2nd-level directory by byte units.
- The 2nd-level directory contains `rank` for every multiple of 2^5, within the subranges of size 2^8. Each entry requires 8 bits.
- For each possible bit pattern of length 8, the rank-lookup-table gives the number of 1's in the pattern. Each entry requires 3 bits but we allocate 8 bits for each entry in order to avoid bit-operations. We may access the rank-lookup-table four times to get `rank`.

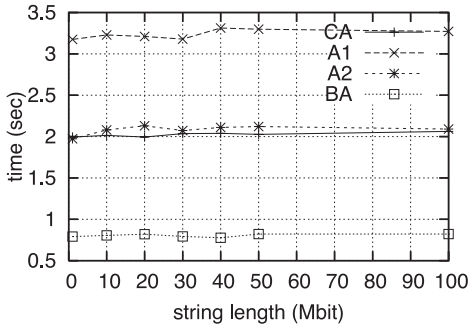
Because each entry in all directories is stored in bits of a multiple of 8, we can find values of entries without bit-operations. So retrievals in our method are fast.

Algorithm II is the most suitable for applying this method because it uses a rank directory but not a select directory. Subranges of select directories are determined according to the distribution of 0's and 1's and they are of various lengths. Note that we can bound the sizes of subranges of select directories, but we cannot control them. Hence, such a byte-based implementation as described above cannot be applied to select directories of Algorithm I and Clark's algorithm, which are main data structures of these algorithms.

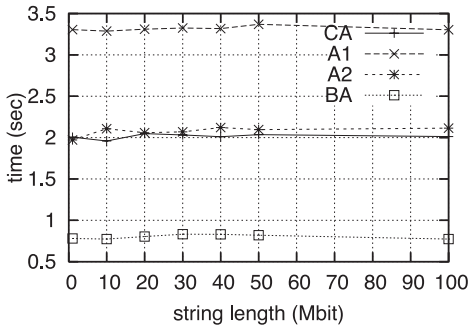
5. Experimental Results

In this section we present experimental results comparing Clark's algorithm (CA) with Algorithm I (A1), Algorithm II (A2), and byte-based Algorithm II (BA). All algorithms except algorithm BA are allowed to use bit-operations, i.e., they are optimized for space. We included algorithm BA in our experiments to look into efficiency in time and inefficiency in space of byte-based implementations.

We compare the performance of the algorithms on 98 ($= 7 \times 14$) random bit-strings of 7 kinds of lengths (1, 10, 20, 30, 40, 50, and 100 Mbits) and with 14 kinds of ratios of 1's (1, 3, 5, 10, 20, 25, 30, 40, 50, 60, 70, 75, 80, and 90%). We measured rank retrieval time, select retrieval time, construction time, and space of auxiliary data structures in order to compare the performances. For rank and select retrieval time, we measured the time taken to perform 10^7 random queries. We used Microsoft Visual C++ 6.0 to implement the algorithms and performed these experiments on the 2.8 GHz Pentium IV with 2 GB main memory.



(a) bit-strings with 10% 1's



(b) bit-strings with 50% 1's

Fig. 4 Rank retrieval time.

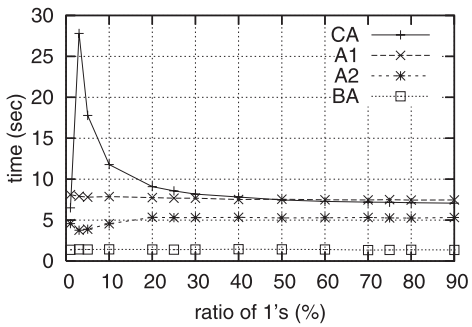


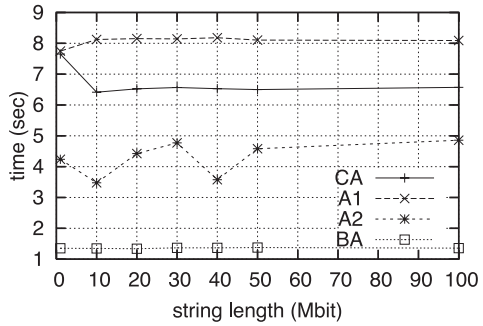
Fig. 5 Select retrieval time on bit-strings of length 50 Mbit.

5.1 Rank Retrieval Time

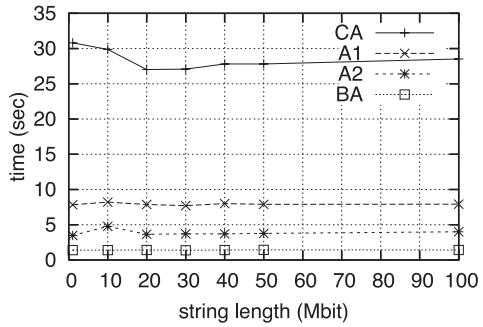
In every algorithm, rank retrieval time is uniform regardless of the ratio of 1's and the length of a string. Figure 4 (a) and (b) show rank retrieval times for bit-strings with 10% 1's and 50% 1's, respectively. In these figures, the vertical axis represents the time taken to perform 10^7 queries and the horizontal axis represents the length of bit-string A. Byte-based Algorithm II is the fastest and Algorithm I is the slowest. The reason why Algorithm I is slow is that it needs two `rank`'s. Algorithm II and Clark's algorithm have the same performance in `rank` because the two algorithms for `rank` are the same.

5.2 Select Retrieval Time

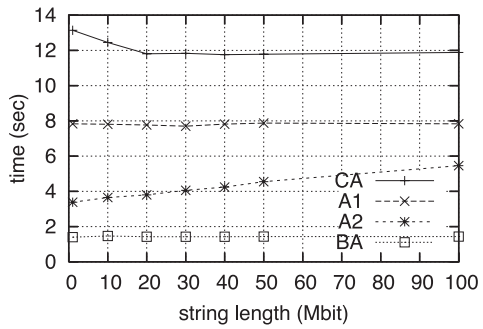
In every algorithm, select retrieval time is slower than rank



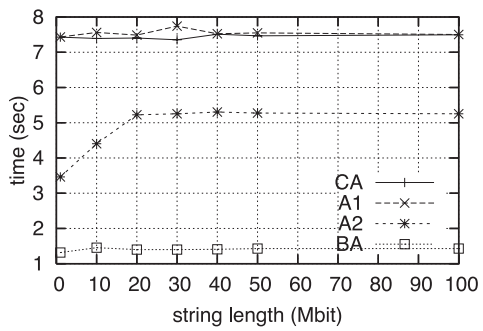
(a) bit-strings with 1% 1's



(b) bit-strings with 3% 1's

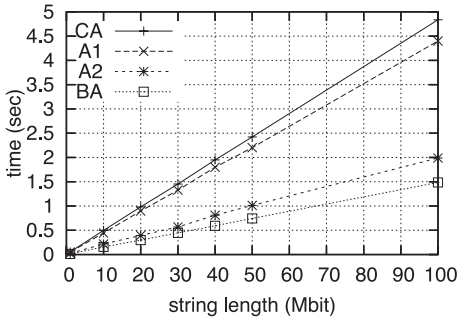


(c) bit-strings with 10% 1's

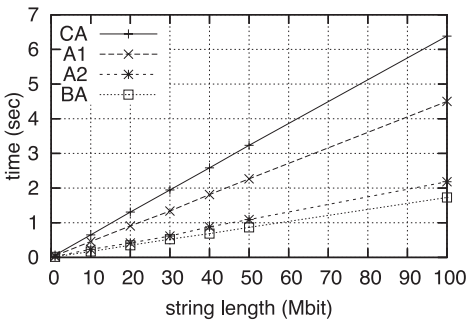


(d) bit-strings with 50% 1's

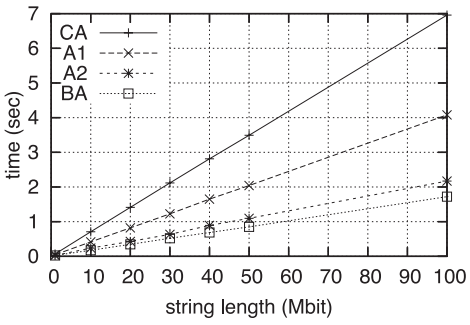
Fig. 6 Select retrieval time on various bit-strings.



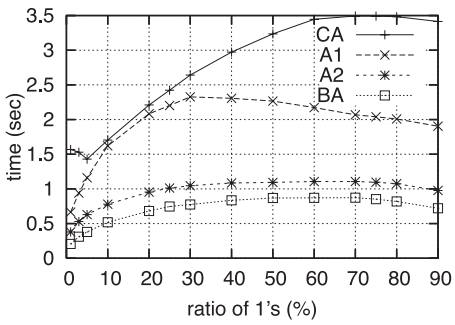
(a) bit-strings with 25% 1's



(b) bit-strings with 50% 1's



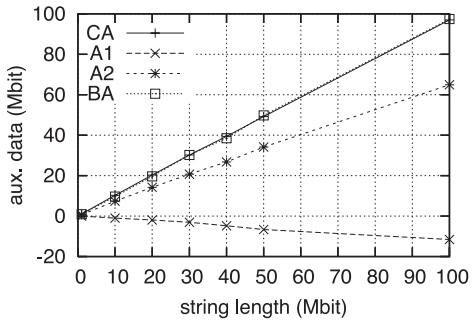
(c) bit-strings with 75% 1's



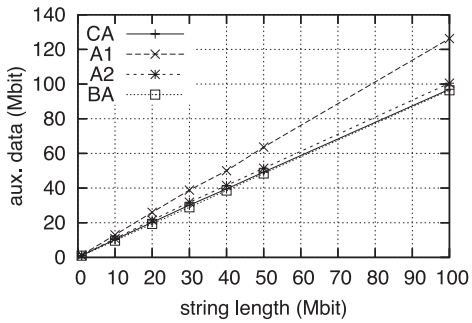
(d) bit-strings of length 50Mbit

Fig. 7 Construction time.

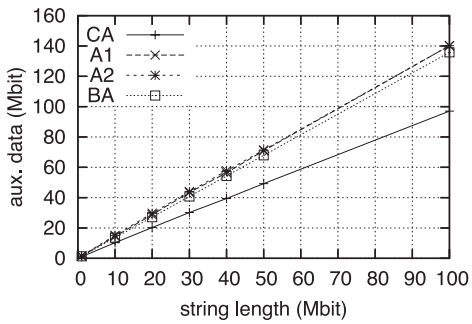
retrieval time. Figure 5 shows select retrieval times for bit-strings of length 50 Mbit, where the horizontal axis represents the ratio of 1's in A. Figure 6 (a)~(d) show select retrieval times for bit-strings with 1%, 3%, 10%, and 50% 1's, respectively. Our algorithms have roughly uniform select retrieval time regardless of both the ratio of 1's and the length of a string. The performance of Clark's algorithm varies according to the ratio of 1's. Its retrieval time becomes slower as the ratio of 1's becomes lower except for 1%. Select queries in strings with 3% 1's are more than three times as slow as those in strings with 50% 1's. The reason is because the block size in the directory for `select` become long and so many accesses to lookup tables are needed as the ratio of 1's becomes lower. An exception is the case of 1% 1's. In this case, no accesses to lookup tables are needed in



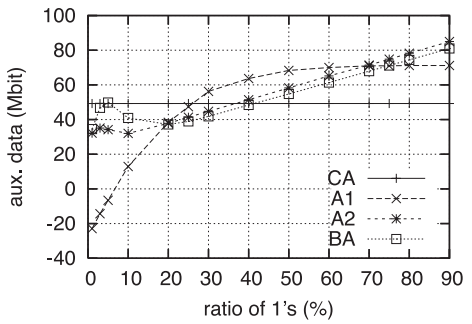
(a) bit-strings with 5% 1's



(b) bit-strings with 40% 1's



(c) bit-strings with 70% 1's



(d) bit-strings of length 50Mbit

Fig. 8 Size of auxiliary data structures.

many queries because answers are recorded explicitly in the directory for `select`. This behavior of Clark's algorithm is observed also in another implementation [16]. In all cases, byte-based Algorithm II is the fastest and Algorithm II is the second. In cases that the ratio of 1's is less than 40% (except for 1%), Algorithm I is faster than Clark's, and the two algorithms have a similar performance in other cases.

5.3 Construction Time

In every algorithm, the construction time of auxiliary data structures is proportional to the length of a string. Figure 7 (a)~(c) show construction times for bit-strings with 25%, 50%, and 75% 1's, respectively. In all cases, byte-based Algorithm II is the fastest, Algorithm II is the second, and Algorithm I is the third. Clark's algorithm is the slowest due to the complication and irregularity of the data structures. Figure 7 (d) shows construction times for bit-strings of length 50 Mbit, where the horizontal axis represents the ratio of 1's in A. While the construction times are generally uniform in Algorithm II and byte-based Algorithm II, the construction time in Clark's algorithm greatly increases as the ratio of 1's becomes higher.

5.4 Space

In every algorithm, the size of auxiliary data structures looks proportional to the length of a string. Theoretically, the size of auxiliary data structures are bounded by $O(n \log\log n/\sqrt{\log n})$ or $O(n \log\log n/\log n)$. It seems, however, that 100 Mbits is so short that these complexities are not discriminated from $O(n)$. Figure 8 (a)~(c) show spaces of auxiliary data structures for bit-strings with 5%, 40%, and 70% 1's, respectively. Figure 8 (d) shows spaces for bit-strings of length 50 Mbit, where the horizontal axis represents the ratio of 1's in A. We do not count the space for a given string A. The reason why Algorithm I has negative values in Fig. 8 (a) is that a transformed string B is even shorter than given string A. Because it is very difficult and complex to implement Clark's algorithm using dynamic memory allocation, we used static allocation. Thus, data of Clark's algorithm in Fig. 8 (d) represent the size in the worst case, which are uniform regardless of the ratio of 1's. If we implement Clark's algorithm using dynamic allocation, the space of Clark's algorithm is decreased, and the retrieval time and the construction time are increased. Algorithm I uses the least space when the ratio of 1's is low, and Clark's algorithm does when the ratio of 1's is high.

6. Conclusions

We proposed two algorithms supporting `rank` and `select` in constant time. Algorithm I requires at most $n + O(n \log\log n/\sqrt{\log n})$ bits and Algorithm II does at most $2n + O(n \log\log n/\log n)$ bits. Our algorithms have constant query time for `select` in practice. Furthermore, we proposed the byte-based implementation method which is efficient in space as well as in time and described how to apply this method to Algorithm II. This implementation method can be applied to some parts of Algorithm I but not to Clark's algorithm for `select` because of irregularity in data structures. The experimental results show that our algorithms are faster than Clark's algorithm for select queries.

They also show that Algorithm II (using the byte-based implementation) is the most efficient in practice when considering both time and space.

Acknowledgements

We would like to thank the anonymous referees for their valuable comments. This work was supported by Grant FPR08-A1-021 of 21C Frontier Functional Proteomics Project from the Korean Ministry of Education, Science & Technology, and also supported by the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Advancement (IITA-2009-C1090-0902-0003).

References

[1] G. Jacobson, Succinct Static Data Structures, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, 1988.
[2] R. Raman and S.S. Rao, "Succinct dynamic dictionaries and trees," Proc. 30th International Colloquium on Automata, Languages, and Programming, pp.357–368, 2003.
[3] J.I. Munro and V. Raman, "Succinct representation of balanced parentheses and static trees," SIAM J. Comput., vol.31, no.3, pp.762–776, 2001.
[4] G. Turan, "Succinct representations of graphs," Discrete Appl. Math., vol.8, pp.289–294, 1984.
[5] G. Jacobson, "Space-efficient static trees and graphs," Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, pp.549–554, 1989.
[6] J.I. Munro, R. Raman, V. Raman, and S.S. Rao, "Succinct representations of permutations," Proc. 30th International Colloquium on Automata, Languages, and Programming, pp.345–356, 2003.
[7] J.I. Munro and S.S. Rao, "Succinct representations of functions," Proc. 31st International Colloquium on Automata, Languages, and Programming, pp.1006–1015, 2004.
[8] J.I. Munro, V. Raman, and S.S. Rao, "Space efficient suffix trees," J. Algorithms, vol.39, no.2, pp.205–222, 2001.
[9] R. Grossi and J.S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," SIAM J. Comput., vol.35, no.2, pp.378–407, 2005.
[10] K. Sadakane, "Succinct representations of lcp information and improvements in the compressed suffix arrays," Proc. 11th ACM-SIAM Symposium on Discrete Algorithms, pp.225–232, 2002.
[11] P. Ferragina and G. Manzini, "Indexing compressed text," J. ACM, vol.52, no.4, pp.552–581, 2005.
[12] J.I. Munro, "Tables," Proc. Conference on Foundations of Software Technology and Theoretical Computer Science, pp.37–42, 1996.
[13] D.R. Clark, Compact Pat Trees, Ph.D. thesis, University of Waterloo, Waterloo, 1996.
[14] R. Pagh, "Low redundancy in static dictionaries with constant query time," SIAM J. Comput., vol.31, no.2, pp.353–363, 2001.
[15] R. Raman, V. Raman, and S.S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets," ACM Trans. Algorithms, vol.3, no.4, article 43, 2007.
[16] R. González, S. Grabowski, V. Mäkinen, and G. Navarro, "Practical implementation of rank and select queries," Poster Proc. 4th International Workshop on Efficient and Experimental Algorithms, pp.10–13, 2005.
[17] P.B. Miltersen, "Lower bounds on the size of selection and rank indexes," Proc. 16th ACM-SIAM Symposium on Discrete Algorithms, pp.11–12, 2005.
[18] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," Proc. Workshop on Algorithm Engineering and Experiments, 2007.

Joong Chae Na received a B.S., an M.S., and a Ph.D. in Computer Science and Engineering from Seoul National University in 1998, 2000, and 2005, respectively. He worked as a visiting postdoctoral researcher in the Department of Computer Science at the University of Helsinki in 2006. He is currently a professor in Department of Computer Science and Engineering at Sejong University. His research interests include design and analysis of algorithms, and bioinformatics.

Ji Eun Kim received the BS and MS degrees in Computer Engineering from Pusan National University in 2003 and 2005, respectively. From 2005 to 2006, she worked as a researcher in ETRI(Electronics and Telecommunications Research Institute). She is now a researcher in ADD(Agency for Defense Development). Her research interests are in design of algorithms and theory of computations.

Kunsoo Park received a B.S. and an M.S. in computer engineering from Seoul National University in 1983 and 1985, respectively, and a Ph.D. in computer science from Columbia University in 1991. From 1991 to 1993 he was a lecturer at King's College, University of London. He is currently a professor in the School of Computer Science and Engineering at Seoul National University. His research interests include design and analysis of algorithms, bioinformatics, and cryptography.

Dong Kyue Kim received the B.S., M.S. and Ph.D. degrees in Computer Engineering from Seoul National University in 1992, 1994, and 1999, respectively. From 1999 to 2005, he was an assistant professor in the Division of Computer Science and Engineering at Pusan National University. He is currently an associate professor in the Division of Electronics and Computer Engineering at Hanyang University, Korea. His research interests are in the areas of embedded security systems, crypto-coprocessors, and information security.