

Rank/Select Operations on Large Alphabets: a Tool for Text Indexing

Alexander Golynski *

J. Ian Munro *

S. Srinivasa Rao *

Abstract

We consider a generalization of the problem of supporting rank and select queries on binary strings. Given a string of length n from an alphabet of size σ , we give the first representation that supports *rank* and *access* operations in $O(\lg \lg \sigma)$ time, and *select* in $O(1)$ time while using the optimal $n \lg \sigma + o(n \lg \sigma)$ bits. The best known previous structure for this problem required $O(\lg \sigma)$ time, for general values of σ . Our results immediately improve the search times of a variety of text indexing methods.

1 Introduction

Given a bit vector B of length n the following operations are very useful in several applications, for $b \in \{0, 1\}$:

- $\text{rank}_B(b, i)$: gives the number of b bits up to position i in B , and
- $\text{select}_B(b, i)$: gives the position of i -th b bit in B .

Jacobson [9] was the first to consider this problem, and gave a structure that takes $o(n)$ extra bits to support both the queries in $O(\lg n)$ bit look-up's. Clark and Munro improved this to a structure that answers both the queries in $O(1)$ time, on a RAM with word-size $\Theta(\lg n)$, using $o(n)$ bits of extra space [1]. Further work on this problem improved the space to $nH_0 + o(n)$ bits (without having to explicitly store the bit vector) [12] while keeping query times $O(1)$ [13], where H_0 is the 0-th order entropy of the given binary string.

Grossi et al. [6] considered a generalization of the problem to larger alphabets. Given a string of length n from an alphabet of size σ , their wavelet tree structure supports the following operations in $O(\lg \sigma)$ time, using $nH_0 + o(n)$ bits:

- $\text{rank}(c, i)$: - finds the number of occurrences of character c before position i ,
- $\text{select}(c, i)$ - finds the position of i -th occurrence of character c in the sequence, and
- $\text{access}(i)$: returns the i -th character in the given sequence.

Ferragina et al. [5] improved the time to $O(1)$ (using essentially the same space) for the case of small alphabets, i.e., when $\sigma = \text{polylog}(n)$, while using $nH_0 + O(n \lg \lg n / \lg \sigma)$ bits. More recently, Grossi and Sadakane [7] improved the space for this structure to $nH_k + O(n \lg \lg n / \lg \sigma)$.

We give two data structures: one that supports rank in $O(\lg \lg \sigma)$ time and select in $O(1)$ time using $nH_0 + O(n)$ bits; and another that supports select in $O(1)$ time, and rank and access in $O(\lg \lg \sigma)$ time, using $n \lg \sigma + o(n \lg \sigma)$ bits. For the latter data structure we give another trade-off that allows access in $O(1)$ time and select in $O(\lg \lg \sigma)$ time while slightly increasing the time for rank.

We also give a relation between our problem in the case of large alphabets, i.e., when $\sigma = n^\epsilon$ for some constant $0 < \epsilon < 1$ and the problem of representing permutations considered by Munro et al. [11]. Namely, we show that an improvement for one problem will lead to an improvement for the other.

1.1 Applications. Our data structure has applications to the problem of text indexing.

Text is “fully indexed” to facilitate fast searches for query patterns. There are, of course, several interpretations of exactly what this means. In He et al. [8] the following query types are discussed :

- *Existential*: does the pattern occur in the text?
- *Cardinality*: how many times does the pattern occur?
- *Listing*: give the positions of each of each occurrence.

*School of Computer Science, University of Waterloo, Waterloo, ON, Canada N2L 3G1. {agolynsk, imunro, ss-rao}@uwaterloo.ca. This work was supported by NSERC and the Canada Research Chairs Programme.

To this we add another useful operation:

Context: give the characters immediately after or before a specific occurrence (similar to what is done in search engines such as Google). Our result is that we can retrieve characters after an occurrence at the cost of $O(1)$ per character and before an occurrence at the cost of $O(\lg \lg \sigma)$ per character. These results are presented in Sections 3.2 and 3.3.

In what follows we describe some applications of our data structure to existing methods. Ferragina and Manzini [3] have shown that, given a text string T over an alphabet of size σ , one can find the number of occurrences of a pattern P of length m by performing $O(m)$ rank queries on the Burrows-Wheeler transform (BWT) of T . Using our structure to store the BWT of T , one can support cardinality queries for pattern of length m in $O(m \lg \lg \sigma)$ time. The space used by this structure is $nH_0 + O(n)$ bits, where n is the length of the text T , and H_0 is the 0-th order entropy of T .

Ferragina et al. [4] have also shown that if there exists a representation of a sequence S of length n over an alphabet of size σ taking $f(n, \sigma)$ bits which answers access in time t_a and rank in time t_r , then we can represent a given text T of length n over an alphabet of size σ using $f(n, \sigma)$ bits. This supports cardinality queries for a pattern of length m in $O(mt_r)$ time, listing queries in $O((t_a + t_r) \lg^{1+\epsilon} n)$ time per occurrence, and retrieving any text substring of length l in $O((l + \lg^{1+\epsilon} n)(t_a + t_r))$ time.

Using our results, we get obtain an index that uses $n \lg \sigma + o(n \lg \sigma)$ bits which supports cardinality queries in $O(m \lg \lg \sigma)$ time, listing queries in $O(\lg \lg \sigma \lg^{1+\epsilon} n)$ time per occurrence, and retrieving any text substring of length l in $O((l + \lg^{1+\epsilon} n) \lg \lg \sigma)$ time.

Combining our results with the techniques from He et al. [8], we can show the following:

THEOREM 1.1. *Given a text string T of length n over an alphabet of size σ , for any constants λ and ϵ such that $0 < \lambda, \epsilon < 1$, we can construct an $O(n \lg \sigma)$ -bit index that answers cardinality queries on a pattern of length m in $O(m \lg \lg \sigma)$ time, and lists each occurrence of the pattern in the text in $O(\lg \lg \sigma \lg^\lambda n)$ time. When $m = \Omega(\lg^{1+\epsilon} n)$, we can in fact list all the occ occurrences of the pattern in $O(m \lg \lg \sigma + occ)$ time. We can also output a substring of T of length l at a given position in $O((l / \lg^{1-\lambda} n + \lg^\lambda n) \lg \lg \sigma)$ time.*

2 Succinct representation of sequences

In this section, we consider the problem of representing a sequence of n integers in the range $[t] = \{1, \dots, t\}$. We are interested in the case when $t \leq n$. We consider the following operations on sequences:

- $\text{rank}(c, j)$ - finds the number of occurrences of c before position j ;
- $\text{select}(c, i)$ - finds the position of i -th occurrence of c in the sequence;
- $\text{access}(i)$ - returns the number at position i in the sequence.

We present two data structures. The first one supports rank in $O(\lg \lg t)$ time and select in $O(1)$ time using $nH_0 + O(n)$ bits (where H_0 is the 0-th order entropy of the sequence). The second structure supports rank and access in $O(\lg \lg t)$ time, and select in $O(1)$ time, using $n \lg t + o(n \lg t)$ bits. We also give a different trade-off for the second data structure that supports access in $O(1)$ time, rank in $O(\lg \lg t \lg \lg \lg t)$ time and select in $O(\lg \lg t)$ time, using the same space.

First we will describe a reduction that will be used in all these data structures. This reduction allows us to consider the problem over a sequence of length t instead n (note that $t \leq n$). We represent the given sequence of length n as a $t \times n$ table T with rows indexed by $1, \dots, t$ and columns by positions in the sequence (i.e. from 1 to n). Entry $T[c, i]$ indicates whether c occurs in position i in the sequence. Let A be a bit vector of length tn obtained by writing T in row major order. There is a simple relation between operations $\text{rank}(c, j)$ and $\text{select}(c, i)$ on the original string and $\text{rank}_A(1, j)$ and $\text{select}_A(1, i)$ on bit vector A :

$$\begin{aligned} \text{rank}(c, i) &= \text{rank}_A(1, (c-1)n + i) \\ &\quad - \text{rank}_A(1, (c-1)n) \\ \text{select}(c, i) &= \text{select}_A(1, \text{rank}_A(1, (c-1)n) + i) \end{aligned}$$

Divide A into *blocks* of size t . We define and implement restricted versions of rank and select called rank-b and select-b with respect to these blocks. Define $\text{rank-b}(i) = \text{rank}(i)$ if i is a multiple of t (i.e. i is the last position in some block) and undefined otherwise; and $\text{select-b}(i)$ is the block number in which the i -th one is located.

$$\begin{aligned} \text{rank-b}(it) &= \text{rank}_A(1, it) \\ \text{select-b}(i) &= \lfloor \frac{\text{select}_A(1, i)}{t} \rfloor \end{aligned}$$

Let us define the *cardinality* of block i as $k_i = \text{rank-b}(it) - \text{rank-b}((i-1)t)$. We construct a bit vector B by writing the cardinalities of all the blocks in unary, i.e., $B = 1^{k_1}01^{k_2}0 \dots 1^{k_n}0$. It is easy to see that

$$\begin{aligned} \text{rank-b}(it) &= \text{rank}_B(1, \text{select}_B(0, i)), \\ \text{select-b}(i) &= \text{rank}_B(0, \text{select}_B(1, i)). \end{aligned}$$

Note that the length of B is $2n$, so that the space to store B and support rank and select on it is $2n + o(n)$ bits.

2.1 Supporting rank and select. To implement full rank and select, we need to be able to do operations of rank and select that are defined locally for each block of A . For the i -th block, we define:

$$\begin{aligned} \text{rank-l}_i(j) &= \text{rank}_A(1, j + it) - \text{rank}_A(1, it), \\ \text{select-l}_i(j) &= \text{select}_A(1, j + \text{rank}_A(1, (i-1)t)). \end{aligned}$$

Thus full rank and select are given by

$$\begin{aligned} \text{rank}(j) &= \text{rank-b}(it) + \text{rank-l}_i(j - it), \\ &\quad \text{where } i = \lfloor j/t \rfloor, \text{ and} \\ \text{select}(j) &= \text{select-l}_i(j - \text{rank-b}((i-1)t)) \\ &\quad + (i-1)t, \text{ where } i = \text{select-b}(j). \end{aligned}$$

We describe an encoding of the i -th block that supports rank-l in $O(\lg \lg t)$ time and select-l in $O(1)$ time, using $k_i \lg t + O(k_i)$ bits. Let A_i be the bit vector of i -th block (of length t). The first part of our storage is just the sorted array E_i of positions of all 1's in A_i , in increasing order. Thus, select-l operation is done in just one access to E_i .

Let F_i be a *sparsified* set that contains every $\lg t$ -th element of E_i . Store a y -fast trie of Willard [14] for F_i . Since F_i is of length $k_i/\lg t$, this structure takes $O(k_i)$ bits (we use a word size of $O(\lg t)$ bits instead of $O(\lg n)$ bits, for the trie structure), and supports rank queries on F_i in $O(\lg \lg t)$ time (since $F_i \subseteq [t]$). Let us call $(\lg t)\text{rank}_{F_i}(j)$ an *approximate rank* of j in E_i . Observe that an approximate rank is within $\lg t$ of the exact rank:

$$(\lg t)\text{rank}_{F_i}(j) \leq \text{rank-l}_i(j) < (\lg t)\text{rank}_{F_i}(j) + \lg t.$$

Now to find $\text{rank-l}_i(j)$, we can do binary search for j in E_i between the positions $(\lg t)\text{rank}_{F_i}(j)$ and $(\lg t)\text{rank}_{F_i}(j) + \lg t$. Thus, rank-l can be implemented in time $O(\lg \lg t)$.

For the i -th block, the table E_i takes $O(k_i \lg t)$ bits, and the Willard's data structure takes an additional $O(k_i)$ bits. Thus, the total space requirement

for all the blocks is $n \lg t + O(n)$ bits. However, if we use a better encoding of Raman et al. [13] for each set E_i then the total space becomes $nH_0 + O(n)$ bits, where H_0 is 0-th order entropy [10].

THEOREM 2.1. *A sequence of n elements from an alphabet of size t can be stored in $nH_0 + O(n)$ bits, to support rank in $O(\lg \lg t)$ time, and select in $O(1)$ time, where H_0 is the 0-th order entropy of the given sequence.*

Some applications do not require $\text{rank}(c, i)$ defined for all positions in the string, but only for positions i , such that i -th character of the string is c . By storing an indexable dictionary [13] on each character of alphabet separately, one can obtain a structure that takes $nH_0 + o(n)$ bits and supports part-rank and select in constant time.

2.2 Supporting rank, select, and access. We now describe a structure that supports rank, select, and access. We divide the given sequence of length n into *chunks* of length t . (For simplicity of presentation, we assume that n is divisible by t .) We first use the $O(n)$ -bit structure to support rank-b and select-b operations as described earlier. Note that each chunk consists of t blocks, one corresponding to every $i \in [t]$. In what follows, we will index blocks by pairs (C, i) in rank-l and select-l operations (where C denotes a chunk).

It is sufficient to describe how to support the three operations for a given chunk. Given a chunk C , for each $i \in [t]$ in increasing order, we write down the positions of all the occurrences of i in C also in increasing order. Observe that the resulting sequence is a permutation on $[t]$, let us denote it by π .

Let l_i be the number of occurrences of i in C . We construct a bit vector X that stores multiplicities l_i in unary, i.e., $X = 01^{l_1}01^{l_2}0 \dots 1^{l_t}$. This vector facilitates access and select-l operations:

$$\begin{aligned} \text{access}_C(j) &= C[j] \\ &= \text{rank}_X(0, \text{select}_X(1, \pi^{-1}(j))) \\ \text{select-l}_{C,i}(j) &= \pi(\text{select}_X(0, i) + j - i) \end{aligned}$$

For each $i \in [t]$, we store every $\lg t$ -th occurrence of i in C using a y -fast trie. We can support $\text{rank-l}_{C,i}(j)$ as earlier: first we find an approximate rank r (using y -fast trie) and then do binary search on π in the range from position $q := \text{rank}_X(1, \text{select}_X(0, i)) + r$ to position $q + \lg t$. The total space used by Willard's y -fast tries is at most $O(t)$ bits per chunk for the total $O(n)$ bits.

To encode permutation π , we use representation of Munro et al. [11]. If we use a representation that supports $\pi(i)$ in $O(1)$ and $\pi^{-1}(i)$ in $O(\lg \lg t)$ time, then the time to support access and rank is $O(\lg \lg t)$, and select is $O(1)$. On the other hand, if we use a representation that supports $\pi(i)$ in $O(\lg \lg t)$ and $\pi^{-1}(i)$ in $O(1)$ time, then the time to support access is $O(1)$, select is $O(\lg \lg t)$, and rank is $O(\lg \lg t \lg \lg \lg t)$ if we index every $\lg \lg t$ -th (instead of every $\lg t$ -th) element in Willard's y -fast trie. Both these structures take $n \lg t + O(n \lg t / \lg \lg t)$ bits. Note that the data structure in [11] dominates the space usage.

Note that in the second tradeoff, we are storing the original sequence explicitly. For such data structures, Grossi and Sadakane [7] offered a new way to improve the space to the k -th order entropy.

THEOREM 2.2. *A sequence of n elements from an alphabet of size t can be stored in $n \lg t + o(n \lg t)$ bits, to support*

1. *rank and access in $O(\lg \lg t)$ time, and select in $O(1)$ time, or*
2. *rank in $O(\lg \lg t \lg \lg \lg t)$ time, select in $O(\lg \lg t)$ time, and access in $O(1)$ time.*

For parameter k , such that $k + \lg t = o(\lg n)$, the space for 2) can be improved to k -th order entropy: $nH_k + o(n \lg t)$ bits.

3 Applications to text indexing and labeled trees

We describe some applications of the above data structure to text indexing. We start with an application to counting number of occurrences of a given pattern P in a text T . Then we describe how to retrieve the context before/after an occurrence of the pattern, without explicitly retrieving the position of occurrence.

3.1 Cardinality queries. Let T be a text string on an alphabet of size σ and let BWT be its Burrows-Wheeler transform. A special symbol '#' is appended to the string T , and the character '#' is defined to be lexicographically smallest among all characters in the alphabet. We store BWT using $n \lg \sigma + O(n)$ space while allowing rank query in $O(\lg \lg \sigma)$ time and select query in $O(1)$ time. In addition, we store a bit vector X of length n with rank and select supported on it, X is of the form

$$X := 11 \underbrace{00 \dots 0}_{o_1 \text{ times}} 1 \underbrace{00 \dots 0}_{o_2 \text{ times}} \dots$$

where o_i is the number of occurrences of i -th character in T , special one in the front of X corresponds to character "#".

The algorithm to count number of occurrences of P is as follows. (It is essentially same as the "backward search" used in [3, 8]. We present it here for completeness.) Let m be the length of P .

- 1: $Q_s \leftarrow 1$ {start of search zone in the suffix array}
- 2: $Q_e \leftarrow n$ {end of search zone in the suffix array}
- 3: **for** $i = m$ to 1 by -1 **do**
- 4: $c \leftarrow P[i]$ {next symbol in P from right to left}
- 5: $p \leftarrow \text{rank}_X(0, \text{select}_X(1, c))$ {start of the c -zone in the suffix array}
- 6: $s \leftarrow \text{rank}_{BWT}(c, Q_s)$ {the number of characters c that point to something before the search zone}
- 7: $e \leftarrow \text{rank}_{BWT}(c, Q_e)$ {the number of characters c that point to the search zone}
- 8: $Q_s \leftarrow p + s$
- 9: $Q_e \leftarrow p + e$

At the end of the algorithm, the interval $[Q_s, Q_e]$ in the suffix array contains the positions of all the occurrences of P in T . Therefore, the number of occurrences of P is $Q_e - Q_s + 1$.

3.2 Decoding text after an occurrence. Let p be a position in the suffix array that points the position $SA[p]$ in text T . We can decode the text to the right of $SA[p]$ using select_{BWT} queries only at a cost of $O(1)$ per character.

- 1: input p {position in the suffix array of interest}
- 2: $c \leftarrow \text{rank}_X(p)$ {we are in c -zone of the suffix array}
- 3: $s \leftarrow \text{select}_X(c)$ { c -zone starts at s }
- 4: $q \leftarrow c - s + 1$ {we are at q -th position of c -zone}
- 5: $p' \leftarrow \text{select}_{BWT}(c, q)$ { p' -th position of suffix array $SA[p']$ points to the $SA[p] + 1$ -st in the text}
- 6: output c {decoded character}
- 7: $p \leftarrow p'$
- 8: repeat as many times as needed

THEOREM 3.1. *Given a text string of length n over an alphabet of size σ , we can store it using $nH_0 + O(n)$ bits, such that given a position p in the suffix array, we can retrieve the context after $SA[p]$ at cost $O(1)$ per character, where SA is the suffix array of the given text.*

3.3 Decoding text before an occurrence. Let position p in the suffix array point to some position $SA[p]$ in text T that is of interest to us. Here we

show how to decode the text to the left of $SA[p]$ using access_{BWT} and rank_{BWT} queries. The cost of outputting each new character is $O(\lg \lg \sigma)$.

- 1: input p {position in the suffix array of interest}
- 2: $c \leftarrow \text{access}_{BWT}(p)$ { $T[SA[p]] = c$ }
- 3: $s \leftarrow \text{select}_X(c)$ { c -zone starts at s }
- 4: $q \leftarrow \text{rank}_{BWT}(c, p)$ {it is q -th character c }
- 5: $p' \leftarrow s + q - 1$ { p' -th position of suffix array $SA[p']$ points to the $(SA[p] - 1)$ -th in the text}
- 6: output c {decoded character}
- 7: $p \leftarrow p'$
- 8: repeat as many times as needed

THEOREM 3.2. *Given a text string of length n over an alphabet of size σ , we can store it using $n \lg \sigma + o(n \lg \sigma)$ bits, such that given a position p in the suffix array, we can retrieve the context before $SA[p]$ at cost $O(\lg \lg \sigma)$ per character, where SA is the suffix array of the given text.*

3.4 Representation of labeled trees. In a recent paper, Ferragina et al. [2] generalized the Burrows-Wheeler transform for text strings to the case of labeled trees. Given a labeled tree on n nodes with labels from an alphabet Σ , **xbw** (generalized Burrows-Wheeler transform) produces a string (over the same alphabet) and a couple of bit vectors, all of length n . They require rank , select and access queries on this string to support Parent , Child , and SubPathSearch operations. There are two tradeoffs that are mentioned: either (1) one doubles the information theoretic minimum space and achieves optimal time for those three operations [2, Theorem 3] (this might not be desirable for large trees); or (2) one uses wavelet trees [6] with space of $nH_0 + o(n)$ bits, but increasing run-times of all three operations by a factor of $O(\lg \sigma)$. Using our results we can improve the running time of all three operations by a factor of $\lg \sigma / \lg \lg \sigma$ as compared to the second tradeoff while keeping the space asymptotically equal to the information theoretic minimum. However, our results do not allow to compress the data structure to 0-th order entropy as compared to wavelet trees.

4 Relation with permutation problem

In this section, we show the tightness of our approach by reducing the problem of supporting rank , select , and access queries to the problem of succinct representation of a permutation considered by Munro et al. [11]. The latter problem is defined as follows. Given a permutation π , we store a data structure that can answer two queries: forward and inverse. Given $i \in [n]$, the forward (inverse) query returns $\pi(i)$ ($\pi^{-1}(i)$). We

call this the *permutation* problem.

We first observe that the permutation problem can be also viewed as a special case of our problem where $t = n$ and each character occurs exactly once. Namely, $\pi(i)$ translates to $\text{access}(i)$, and $\pi^{-1}(i)$ to $\text{select}(i, 1)$ query. Note that the extra space required by our data structure matches the one in [11] if we require that π^{-1} queries should be answered in $O(1)$ time and π in $O(\lg \lg n)$ time.

We now give a more involved reduction that shows that our result matches [11] in the case where $t = n^\epsilon$ for any constant $0 < \epsilon < 1$. Given a permutation π , define a sequence S as follows:

$$S[i] := \lfloor \frac{\pi(i)t}{n} \rfloor$$

In other words, $S[i]$ stores the $\lfloor \lg t \rfloor$ most significant bits of $\pi(i)$, for each i . We encode the rest of the information about π (i.e., the $\lfloor \lg n \rfloor - \lfloor \lg t \rfloor$ least significant bits of each $\pi(i)$) in t permutations $\pi_1, \pi_2, \dots, \pi_t$ each on $\lfloor n/t \rfloor$, as follows. If j -th occurrence of i in S is at position k then

$$\pi_i(j) := \pi(k) - (i - 1)t$$

We encode these t permutations recursively.

It is not hard to see that the following procedures retrieve π and its inverse.

- | | |
|---|--|
| <ol style="list-style-type: none"> 1: Computing $\pi(i)$ 2: $c \leftarrow \text{access}(i)$ 3: $j \leftarrow \text{rank}(c, i)$ 4: $k \leftarrow \pi_c(j)$ 5: Return $n(c - 1)/t + k$ | <ol style="list-style-type: none"> 1: Computing $\pi^{-1}(i)$ 2: $c \leftarrow \lfloor \frac{it}{n} \rfloor$ 3: $j \leftarrow i - c \frac{n}{t}$ 4: $k \leftarrow \pi_c^{-1}(j)$ 5: Return $\text{select}(c, k)$ |
|---|--|

Let t_r , t_s , and t_a denote the times for operations rank , select , and access respectively. Then $\pi(i)$ can be computed in $(t_a + t_r) \lg n / \lg t$ time, and π^{-1} can be computed in $O(t_s \lg n / \lg t)$ time. The space storage for each level of recursion is $n \lg t + O(n \lg t / \lg \lg t)$ bits for the total $n \lg n + O(n \lg n / \lg \lg t)$. Consider the case of large alphabets, i.e., $t = n^\epsilon$ so that we have constant number of levels of recursion, and substitute $t_s = O(1)$, $t_r = t_a = O(\lg \lg t)$, we obtain results that match the best bounds by Munro et al. [11] discussed above.

This implies that progress on our problem would improve the data structure for representing permutation. The converse is also true, as it was noted earlier that the space to store permutation and its inverse is dominating in our data structure.

References

- [1] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.
- [2] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. *Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science*, 2005.
- [3] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.
- [4] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval*, pages 150–160, 2004.
- [5] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Succinct representation of sequences. Technical Report TR/DCC-2004-5, Department of Computer Science, University of Chile, August 2004.
- [6] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.
- [7] Roberto Grossi and Kunihiko Sadakane. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2006.
- [8] Meng He, J. Ian Munro, and S. Srinivasa Rao. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 23–32, 2005.
- [9] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.
- [10] Veli Mäkinen and Gonzalo Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical Report C-2004-20, Univ. Helsinki, Dept. CS, April 2004.
- [11] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pages 345–356, 2003.
- [12] Rasmus Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.
- [13] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applica-
- [14] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.