Algorithms **2014**, *7*, 608-620; doi:10.3390/a7040608

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

High-Order Entropy Compressed Bit Vectors with Rank/Select

Kai Beskers [1] and Johannes Fischer [2,*]

[1] Institut für Theoretische Informatik, Karlsruhe Institute of Technology, Kaiserstraße 12, 76131 Karlsruhe, Germany; E-Mail: kai.beskers@gmx.de

[2] Fakultät für Informatik, Technical University of Dortmund, Otto-Hahn-Straße 14, 44227 Dortmund, Germany

* Author to whom correspondence should be addressed; E-Mail: johannes.fischer@cs.tu-dortmund.de; Tel.: +49-231-755-7711; Fax: +49-231-755-7740.

External Editor: Tak-Wah Lam

Received: 21 August 2014; in revised form: 8 October 2014 / Accepted: 28 October 2014 / Published: 3 November 2014

Abstract: We design practical implementations of data structures for compressing bit-vectors to support efficient rank-queries (counting the number of ones up to a given point). Unlike previous approaches, which either store the bit vectors plainly, or focus on compressing bit-vectors with low densities of ones or zeros, we aim at low entropies of higher order, for example $101010\ldots10$. Our implementations achieve very good compression ratios, while showing only a modest increase in query time.

Keywords: design and analysis of algorithms; data compression; implementation and testing of algorithms

---

1. Introduction

Bit vectors are ubiquitous in data structures. This is even more true for *succinct* data structures, where the aim is to store objects from a universe of size u in asymptotically optimal $(1 + o(1)) \log u$ bits. The best example are succinct trees on n nodes, which can be represented by parentheses in optimal $2n+o(n)$ bits, while allowing many navigational operations in optimal time [1].

Bit vectors often need to be augmented with data structures for supporting *rank*, which counts the number of 1's up to a given position, and *select*, which finds the position of a given 1 [2–6]. Often, the bit-vectors are *compressible*, and ideally this compressibility should be exploited. To the best of our

knowledge this has so far been done mostly for *sparse* bit-vectors [4,5]. For other kinds of regularities, we are only aware of two recent implementations [7,8], which are, however, tailored towards bit vectors arising in specific applications (FM-indices [8] and wavelet trees on document arrays [7]), though they may also be effective in general.

1.1. Our Contribution

We design practical implementations for compressing bit-vectors for rank- and select-queries that are compressible under a different measure of compressibility, namely bit-vectors having a low empirical entropy of order $k > 0$ (see Section 2 for formal definitions). Our algorithmic approach is to encode fixed-length blocks of the bit-vector with different kinds of codes (Section 3.1). While this approach has been theoretically proposed before [9], we are not aware of any practical evaluations of this scheme in the context of bit-vectors and rank-/select-queries. An interesting finding (Section 3.2) is that we can show that the block sizes can be chosen rather large (basically only limited by the CPU's word size), which automatically lowers the redundancy of the (otherwise incompressible) rank-information. We focus on data structures for rank, noting that select-queries can either be answered by binary searching over rank-information, or by storing select-samples in nearly the same way as we do for rank [10].

We envision at least two possible application scenarios for our new data structures: (1) compressible trees [11], where the bit-vectors are always *dense* (same number of opening and closing parentheses), but there exist other kinds of regularities. Easy examples are stars and paths, having balanced parentheses representations $(()()\ldots())$ and $(((\ldots)))$, respectively. (2) wavelet trees on repetitive collections of strings that are otherwise incompressible, like individual copies of DNA of the same species [7].

2. Preliminaries

2.1. Empirical Entropy

Let T be a text of size n over an alphabet Σ. The empirical entropy of order zero [12] is defined as:

$$H_0(T) := \sum_{c \in \Sigma, n_c > 0} \frac{n_c}{n} \log \frac{n}{n_c},$$

where n_c is the number of occurrences of the symbol c in T. H_0 is a lower bound for any compression scheme that encodes each occurrence of a symbol $c \in \Sigma$ within T into the same code regardless of its context. Using such compression schemes any symbol of the original text can be decoded individually, allowing random access to the text with little overhead. However, H_0 only takes into account the relative frequency of each symbol, and so ignores frequent combinations. Much better compression is often possible by taking the context into account.

The empirical entropy of order k is defined as:

$$H_k(T) = \frac{1}{n} \sum_{s \in \Sigma^k, T^s \neq \epsilon} |T^s| \cdot H_0(T^s),$$

where T^s is the concatenation of all symbols in T directly following an occurrence of s.

The empirical entropy of order k is a lower bound for any compressor that requires at most k preceding symbols in addition to the code in order to decode a symbol. We have $\log |\Sigma| \geq H_0 \geq H_1 \geq \ldots \geq 0$. For texts derived from a natural source H_k is often significantly smaller than H_0 even for a quite small k. Thus, H_k is the measure of choice for any context based random access compression scheme.

2.2. Empirical Predictability

Recently, Gagie [13] defined a new complexity measure called *empirical predictability*. If we are asked to guess a symbol chosen uniformly at random without any context given, our best chance is to guess the most frequent symbol. The probability of a correct prediction $P_0(T) := \max_{c\in\Sigma} \frac{n_c}{n}$ is called *the empirical predictability of order 0*. Now we consider being asked the same question, but we are allowed to read the preceding k bits of context or, if there are not enough preceding bits, the exact symbol. We then call the chance to predict correctly the *empirical predictability of order* k. It is defined as $P_k(T) = \frac{1}{n}\sum_{s\in\Sigma^k, T^s\neq\epsilon} |T^s| \cdot P_0(T^s)$. (Actually, Gagie [13] adds a summand of $\frac{k}{n}$ for the first k bits of the text. In contrast, we consider k to be a small constant. Considering $\lim_{n\to\infty} \frac{k}{n} = 0$ we can omit this summand for both empirical entropy and empirical predictability.) Note that $1 \geq P_k(T) \geq \frac{1}{|\Sigma|}$. If $P_k(S)$ is high we intuitively expect a compressed version of the text to be small. This is inverse to the other complexity measures, where we need a low value to get a small compressed text. In fact, we prove:

Lemma 1. *For binary texts ($\Sigma = \{0, 1\}$) the empirical entropy $H_k(T)$ is an upper bound for $1 - P_k(T)$.*

Proof. Let k be fixed. For $s \in \Sigma^*$ let $p_s := |T^s| \cdot P_0(T^s)$ be the number of correct predictions after reading s and $m_s := |T^s| - p_s$ the number of incorrect predictions after reading s.

To simplify notation we assume $|T^{sx}| \neq 0 \; \forall s \in \{0, 1\}^k, x \in \{0, 1\}$. Then:

$$\begin{aligned}
H_k &= \sum_{s\in\{0,1\}^k} \frac{|T^s|}{n} \sum_{x\in\{0,1\}} \frac{|T^{sx}|}{|T^s|} \log \frac{|T^s|}{|T^{sx}|} \\
&= \sum_{s\in\{0,1\}^k} \left(\frac{|T^{s0}|}{n} \log \frac{|T^s|}{|T^{s0}|} + \frac{|T^{s1}|}{n} \log \frac{|T^s|}{|T^{s1}|} \right) \\
&= \sum_{s\in\{0,1\}^k} \frac{m_s}{n} \underbrace{\log \frac{\overbrace{|T^s|}^{\geq 2}}{m_s}}_{\geq 1} + \sum_{s\in\{0,1\}^k} \frac{p_s}{n} \underbrace{\log \frac{\overbrace{|T^s|}^{\geq 1}}{p_s}}_{\geq 0} \\
&\geq \sum_{s\in\{0,1\}^k} \frac{m_s}{n} \\
&= 1 - P_k,
\end{aligned}$$

which proves the claim. □

Note that we omitted some significant terms; the actual $1 - P_k$ should be even lower. This relation between the empirical entropy and the empirical predictability had not been established before. Lemma 1 will be used in Section 4, since there the test data generator mimics the process of empirical predictability; but we think that the lemma could also have different applications in the future.

2.3. Succinct Data Structures for Rank

Let $T[0,n)$ be a *bit-string* of length n. The fundamental *rank*- and *select*-operations on T are defined as follows: $\mathsf{rank}_1(T,i)$ gives the number of `1`'s in the prefix $T[0,i]$, and $\mathsf{select}_1(T,i)$ gives the position of the i'th `1` in T, reading T from left to right ($0 \leq i < n$). Operations $\mathsf{rank}_0(T,i)$ and $\mathsf{select}_0(T,i)$ are defined similarly for `0`'s.

In [2,3] Munro and Clark developed the basic structure to solve rank/select queries in constant time within $n + o(n)$ space. Most newer rank/select dictionaries are based on their work, and so is ours. We partition the input bit string into large blocks of length $b_l := \log^2 n$. For each large block we store the number of `1`'s before the block, which is equal to a rank sample at the block border. As there are $\frac{n}{b_l}$ large blocks and we need at most $\log n$ bits to store each of those values, this requires only $o(n)$ bits of storage. We call this the *top-level structure*.

Similarly, we partition each large block into small blocks of size $b_s := \frac{\log n}{2}$. For each small block we store the number of `1`'s before the small block and within the same large block, which is equal to a rank sample relative to the previous top-level sample. As there are $\frac{n}{b_s}$ small blocks and we need at most $\log(b_l)$ bits, this requires only $o(n)$ bits of storage. We call this the *mid-level structure*.

As the size of small blocks is limited to b_s, there are $2^{b_s} = \sqrt{n}$ different small blocks possible. Note that not all possible small blocks necessarily occur. For every possible small block and each position within a small block we store the number of `1`'s within this small block before the given position in a table. This table requires $\sqrt{n} \cdot b_s \log b_s = o(n)$ bits. We call this table the *lookup table*.

For each small block we need to find the index of the corresponding line in the lookup table. This is, in fact, the original content of the block, and we get it from the text T itself. As we intend to solve this problem with something different than a plain storage of T, we call the representation of T the *bottom-level structure*. If we use an alternative bottom-level structure the rank structure can still simulate T for constant time read-only random access.

3. New Data Structure

As described in the previous section, rank/select dictionaries break down into top-level, mid-level, bottom-level, and a lookup table. While the top and mid-level structures basically contain rank/select information, the text T is stored in the bottom-level and the lookup table. In fact the bottom-level and the lookup table are sufficient to reconstruct T. Thus we try to exploit the compressibility of T here.

The mid-level structure conceptually partitions T into blocks, which are then used to address the lookup table. Thus the most native approach to compress the bottom-level is to encode *frequent* blocks into *short* code words. Using the shortest code for the most frequent blocks produces the best possible compression. Amongst such block encoding techniques is the Canonical Code (Section 3.1.1). However, it produces variable length code words and thus we do not have an easy constant time random access (though this is possible with a bit of work). In order to provide constant time random access we investigate three approaches. Two of them are totally independent of the used code and differ in size and speed. The third one is a code that tries to get fixed length codes in most cases while using small auxiliary data structures for the exceptions. We call it the *Exception Code*.

The upper-level rank structures already contain the information of how many 1's a desired block contains. Considering this, a block's code does not need to be unique amongst all blocks, but only unique amongst all blocks with the same number of 1's. The actual unique code of a block b then is the pair $(\#(b), \sigma(b))$, where $\#(b)$ is the number of 1's in b, and $\sigma(b)$ is the stored code word. $\sigma(b)$ is calculated for every $\#(b)$ separately, and we require a decoding table for each $\#(b)$, but all code words $\sigma(b)$ can be stored in the same data structure. Note that we still only have one decoding table entry per occurring block, and thus memory consumption only increases by $O(b_s) = o(n)$. This is done for each of the codes in Section 3.1.

3.1. Coding Schemes

3.1.1. Canonical Code

This coding scheme corresponds to the idea of Ferragina and Venturini [9], where it is also shown that the coding achieves empirical entropy of order k. It works as follows. We order all possible bit strings by their length and, as secondary criterion, by their binary value. We get the following sequence: $\epsilon, 0, 1, 00, 01, 10, 11, 000, 001 \ldots$. Note that we can calculate the position j of a bit string s in this sequence: $j = 1s - 1$, where $1s$ is the bit string s with a preceding 1, interpreted as a binary number. We use the first element of this sequence to encode the most frequent block, the second one for the second most frequent block, *etc.* This is obviously optimal in terms of memory consumption as long as we assume the beginnings and endings of code words to be known.

As long as each block is encoded in a code of fixed length b the code of a specific block i can be found by simple arithmetic. However, the Canonical Code produces variable length code words and thus we require a technique to find the desired block code. We next describe two approaches to achieve that.

One option is to maintain a two-level pointer structure [9], similar to the top and mid-level of the rank directory described in Section 2.3. For each large block we store the beginning of its first encoded small block directly in $\log n$ bits using $O(\frac{n \log n}{b_l}) = o(n)$ bits in total. We call it the *top-level pointers*. The *mid-level pointers* defined analogously and use $\frac{n \log(b_l)}{b_s} = o(n)$ bits. As for rank structures this only requires constant time to access, but the space consumption is much more significant than the $o(n)$ indicates.

An alternative approach is to store a bit vector parallel to the encoded bit vector, where we mark the beginning of each code word with a 1 and prepare it for select queries (as previously done in [14,15], for example). To find the ith code word we can use $\mathsf{select}_1(V, i)$ on this *beginnings vector* V. If the original text is highly compressible, the encoded bit vector and thus the beginnings vector is rather short compared to other data structures used. Otherwise, it is sparsely populated and can be compressed by choosing an appropriate select dictionary, for example the sparse array by Sadakane and Okanohara [5]. On the downside, using select to find the proper code word comes with constant but significant run-time costs in practice.

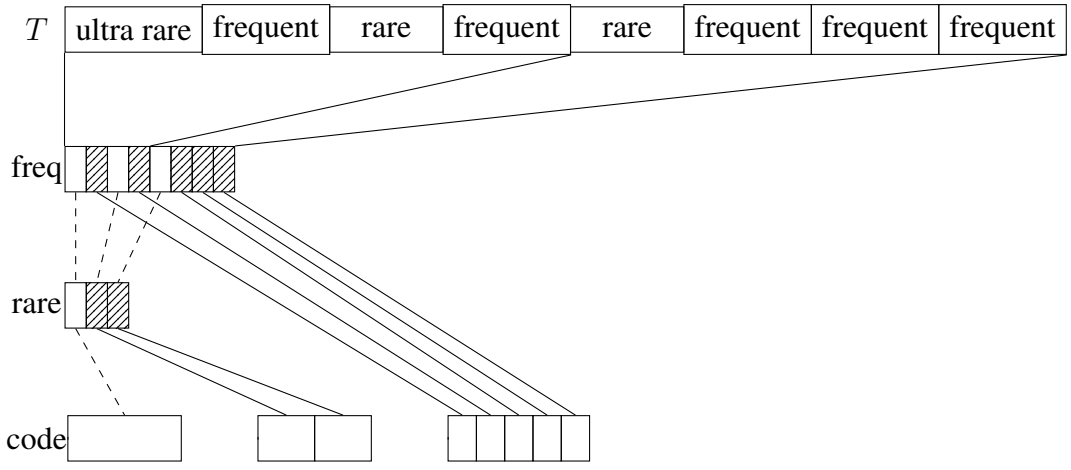
3.1.2. Exception Code

For this code we categorize the blocks into 3 classes (see also [16] for the a generalization of this idea called *alphabet partitioning*): frequent blocks, rare blocks and ultra rare blocks. For the first two classes we choose parameters b_{freq} and b_{rare}, defining how many bits we use to encode their blocks. Ultra rare blocks are stored verbatim, so they still need their original size b_s. We define the most frequent $2^{b_{\text{freq}}}$ blocks to be frequent blocks. Rare blocks are the most frequent $2^{b_{\text{rare}}}$ blocks that are not yet classified as frequent. All other blocks are classified ultra rare.

We maintain simple arrays for each of those classes of blocks, encoding all blocks of that class in the order of their occurrence. In order to access a block we need to find out what class it belongs to and how many blocks of that class occur before. For this we maintain a bit vector *freq* with one bit per block. We mark frequent blocks with a `1` and prepare it for rank queries using an appropriate rank dictionary. Those rank queries tell us how many frequent blocks occur before a desired block, so it is already sufficient for accessing frequent blocks. For non-frequent blocks subtracting the number of previous frequent blocks from its index we obtain its position in a conceptional list of non-frequent blocks. With this we can proceed exactly the same way as above: maintaining a bit vector *rare* with one bit per non-frequent block marking rare blocks with a `1` and preparing it for rank queries. To access a non-frequent block we need to rank it both in *freq* and in *rare*. Figure 1 illustrates the data structures.

The Exception Code uses more space than the Canonical Code for the encoded blocks, and it is not clear at all if it achieves kth order empirical entropy in theory. However, the former only requires two rank structures on *freq* and *rare* to enable random access. As they are built upon bit vectors with only one bit per block, they require at most $\frac{n}{b}(1 + o(1))$ bits for blocks of size b, and our hope is that this is small compared to the beginnings vector or even the two-level pointer structure required for the Canonical Code structure. In terms of execution time we need little more than one subordinate rank query for frequent blocks, and two for non-frequent blocks, hopefully resulting in faster average query times.

Figure 1. Illustration to the Exception Code. Shaded blocks denote `1`'s, and empty blocks `0`'s.



3.2. Table Lookup

3.2.1. Verbatim Lookup vs. Broadword Computation

In the original Munro and Clark version as described in Section 2.3, the lookup table stores the relative rank or select results for each of the 2^{b_s} possible small blocks and each of the b_s possible queries in $\log(b_s)$ bits requiring $2^{b_s} \cdot b_s \cdot \log b_s$ bits of storage. To keep the table small, b_s had to be chosen very small. Vigna [6] bypasses the lookup table and instead calculates its entries on the fly via *broadword computation*. According to Vigna [6], "broadword programming uses large (say, more than 64-bit wide) registers as small parallel computers, processing several pieces of information at a time." This is efficient as long as blocks fit into processor registers. In our approach bottom-level blocks are encoded only based on their frequency, and not on their content. Thus we inevitably need a decoding table. After looking up the appropriate table row it needs to provide rank information. This can be stored directly within the table, turning it into a combined decoding and lookup table.

3.2.2. Expected Table Size

In the rank/select structure as described in Section 2.3, all *possible* small blocks are listed. This implies that b_s has to be small, which increases the number $N := \frac{n}{b_s}$ of small blocks. In our structure the lookup table only contains blocks that do actually occur in T. (This is possible since both of the codes above—Canonical Code and Exception Code—imply an implicit enumeration of the occurring blocks *without gaps*; see, for example, the first paragraph of Section 3.1.1 for how to compute this numbering.) Thus the table size is limited by $O(n)$, without restricting the choice of b_s. For predictable texts ($P_k(T) \gg \frac{1}{2}$), we intuitively expect most blocks to occur very often within T (and thus not increasing the lookup table size). This intuition is confirmed in the following, by calculating the expected number of entries in the lookup table.

To this end, look at a fixed block $B \in \{0,1\}^b$ of $b := b_s$ bits. Its probability of occurrence depends mainly on the number i of bits that are not as predicted. Assuming the first k bits are distributed uniformly at random, the probability of B (with i mispredictions) to occur is:

$$P_{b,i} = \frac{1}{2^k} \cdot P_k^{b-k-i}(1-P_k)^i \,.$$

There are

$$C_{b,i} = 2^k \cdot \binom{b-k}{i}$$

blocks with i mispredictions, so we expect

$$E_{b,i}(N) = C_{b,i} \cdot (1-(1-P_{b,i})^N)$$

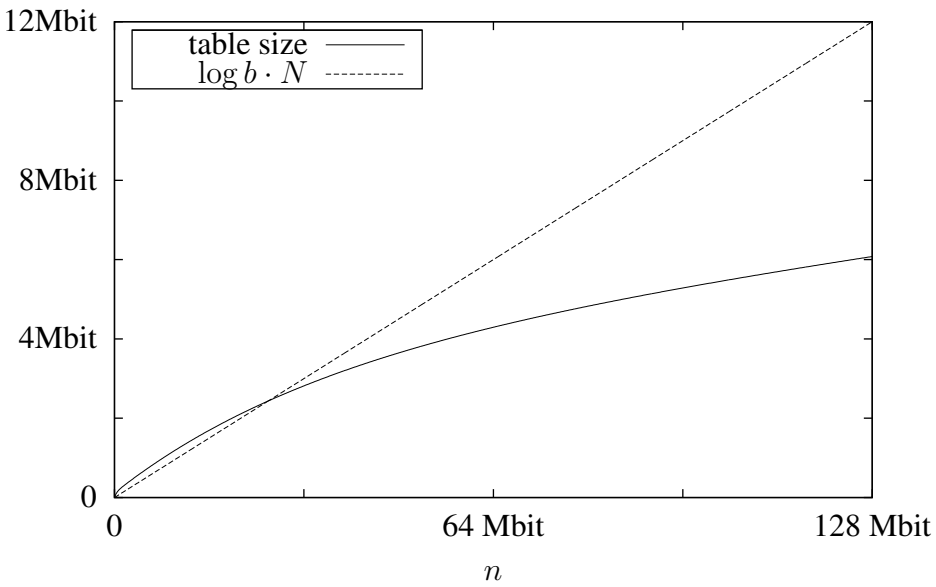
of them to appear within T. This is the well known formula of bounded exponential growth, a monotonic concave function bounded from above by $C_{b,i}$. Its derivative $E'_{b,i}(N)$ is proportional to the satiation $C_{b,i} - E_{b,i}(N)$, and $E'_{b,i}(0) = P_{b,i}$. For small i the upper bound $C_{b,i}$ is quite small. For larger i the growth

rate $E'_{b,i}$ quickly falls behind the linear growth of other data structures used. Summing over all possible i, we get the expected number of blocks occurring in the lookup table:

$$E_{k,b}(N) = \sum_{i=0}^{b-k} E_{b,i}(N) = \sum_{i=0}^{b-k} \left(1 - (1 - P_{b,i})^N\right) \cdot \left(2^k \cdot \binom{b-k}{i}\right).$$

In Figure 2 we plot the expected table size, which is $E_{k,b}(N) \cdot b$, using $P_k = 0.99$ and $b = 64$ as an example. For comparison we also plot $N \cdot \log b$, which is a lower bound for the mid-level data structure. We can see that despite the large block size the table's memory consumption is less than that of the mid-level data structure. As the lookup table is the only data structure that grows with increasing block sizes and the mid-level structure significantly profits from large block sizes, this shows that $b = b_s$ has to be chosen quite high.

Figure 2. Expected memory usage for the lookup table for a text with $P_k = 0.99$ and block size $b = 64$.



4. Test Data Generator

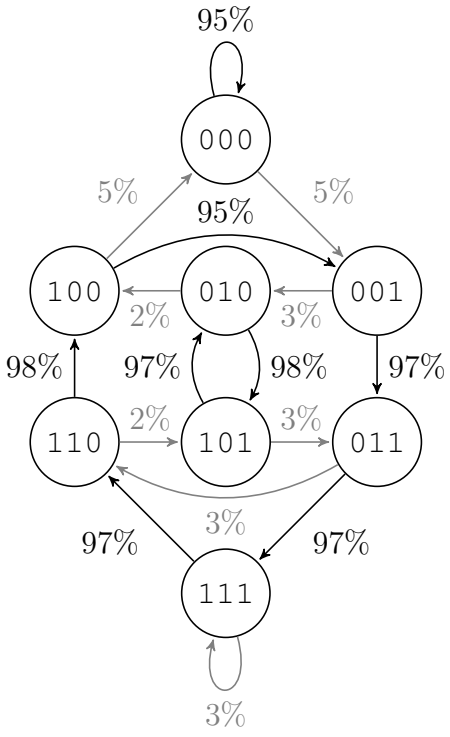
Our data structures aim to fit data with low empirical entropy of higher order even if the empirical entropies of lower order are quite high. We created such data by a specific generator that we are going to describe next, which might be of independent interest. It produces bit vectors with lower order empirical entropies very close to 1 ($H_0 \approx H_1 \approx \ldots \approx H_{k-1} \approx 1$) and higher order empirical entropies significantly lower ($1 \gg H_k \geq H_{k+1} \geq \ldots$).

The test data generator rolls each bit with a probability $P(1|s')$ for a 1 dependent on s', where s' are the k preceding bits. We get these probabilities from a table. The first half of this table, containing probabilities for bit patterns starting with a 0, is filled with values close to 1 or close to 0 at random. By Lemma 1 and since our test data generator mimics the empirical predictability, this ensures a low H_k. The second half is filled with exactly the opposite probabilities namely $P(1|0s) = 1 - P(1|1s)$

for all bit patterns s of length $k-1$. This ensures lower order empirical entropies to be 1, as shown in Lemma 2 below.

From a different point of view this test data generator is a probabilistic finite state machine with one state for each bit pattern of length k and two state transitions out of each state, one with high probability and one with low probability. Due to filling the second half of the table opposite to the first, the two incoming state transitions of each state behave like the outgoing ones: One of them has a high probability and the other has a low probability. Thus we get circles of high probability edges and a low probability to switch the circle or jump within a circle. Figure 3 shows such a finite state machine, the according probability table, and an example output bit string.

Figure 3. Example of a test data generation. (**a**) The associated finite state machine, low probability edges are shown gray. (**b**) Table of probabilities to roll a `1`. (**c**) Output, low probability rolls are underlined.



(a)

s	$P[1\|s]$	s	$P[1\|s]$
000	0.05	100	0.95
001	0.97	101	0.03
010	0.98	110	0.02
011	0.97	111	0.03

(b)

```
1100111001110000000000
0000000000001110011100
1110011100111001110011
1001010101010101001110
0111000000000000000...
```

(c)

Lemma 2. *An infinite text generated that way has* $H_0 = \ldots = H_{k-1} = 1$.

Proof. We have to show that the finite state machine enters each state with the same probability in the long term, thus we have to show that the uniform distribution vector $\mathcal{E} = (1, \ldots, 1)$ is an eigenvector of the state transition matrix $\mathcal{M}$ of the Markov process. For each $s \in \{0, 1\}^{k-1}$ we have:

$$\mathcal{ME} = \begin{array}{c} \\ s0\to \\ s1\to \end{array}\overset{\begin{array}{ccccc} & 0s & & 1s & \\ & \downarrow & & \downarrow & \end{array}}{\begin{pmatrix} & 0 & & 0 & \\ & \vdots & & \vdots & \\ & 0 & & 0 & \\ 0\ldots 0 & p & 0\ldots 0 & 1-p & 0\ldots 0 \\ 0\ldots 0 & 1-p & 0\ldots 0 & p & 0\ldots 0 \\ & 0 & & 0 & \\ & \vdots & & \vdots & \\ & 0 & & 0 & \end{pmatrix}} \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{array}{c} 0s\to \\ \\ 1s\to \end{array}\begin{pmatrix} \vdots \\ 1 \\ \vdots \\ 1 \\ \vdots \end{pmatrix};$$

as this holds for all $s \in \{0,1\}^{k-1}$, we get $\mathcal{ME} = \mathcal{E}$, which proves the claim. □

Lemma 2 also holds for finite texts except for statistical errors, which are negligible for long enough texts.

5. Experimental Results

We implemented the data structures described in this article in C++ and made extensive tests tuning the parameters. The source code is available at [17]. We used GNU g++ 4.6.3 with the flags -O3 -funroll-loops -DNDEBUG -std=c++0x to compile our programs. Our test computer was an Intel i5@3.33 GHz with 32 kiB L1-, 256 kiB L2-, and 4 MiB L3-cache and 8 GiB RAM. Interestingly, tuning for space also resulted in faster query times, mainly due to caching effects. This resulted in the following implementations:

canRnk2LPTR A basic 2-level structure, using the Canonical Code from Section 3.1. Backed by the analysis of the expected table size (Section 3.2), we used large block sizes on both levels, namely $b_s = 2^6$ and $b_l = 2^{12}$.

canRnkBeg The same as canRnk2LPTR, but this time using a bit-vector marking the beginnings of the codewords.

exceptionRank The Exception Code from Section 3.1, using parameters $b_s = 2^6$, $b_l = 2^{12}$, $b_{\text{freq}} = 2^5$, and $b_{\text{rare}} = 2^{12}$.

We compare our data structures to several existing implementations of rank and select dictionaries from different sources. We stress that none of these data structures was designed to achieve high order compression; we just include them in our tests to give some idea of how our new structures compare with existing implementations.

sdslMcl Munro and Clark's original rank-dictionary, using an implementation from Simon Gog's sdsl [18].

sdslv5 A space-conscious rank implementation from sdsl, tuned to use only 5% of space on top of the bit vector.

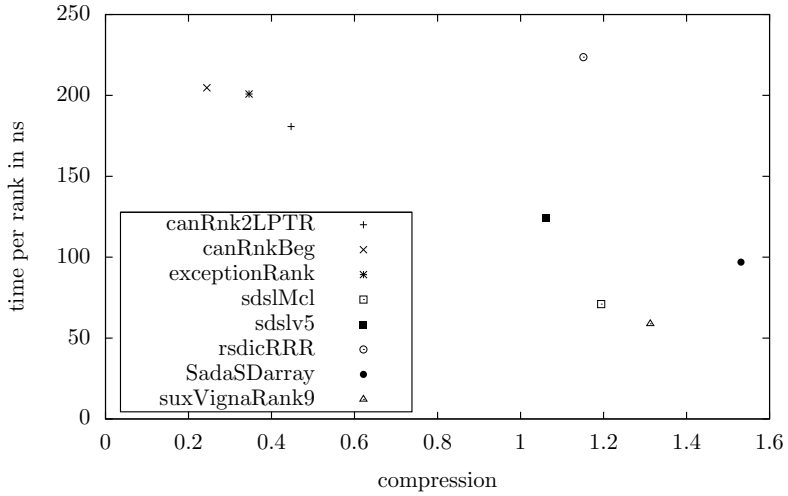
rsdicRRR A 0-order compressed version of Navarro and Providel's rank-dictionary [10], using an implementation by Daisuke Okanohara [19].

SadaSDarray Okanohara and Sadakane's SDArray [5].

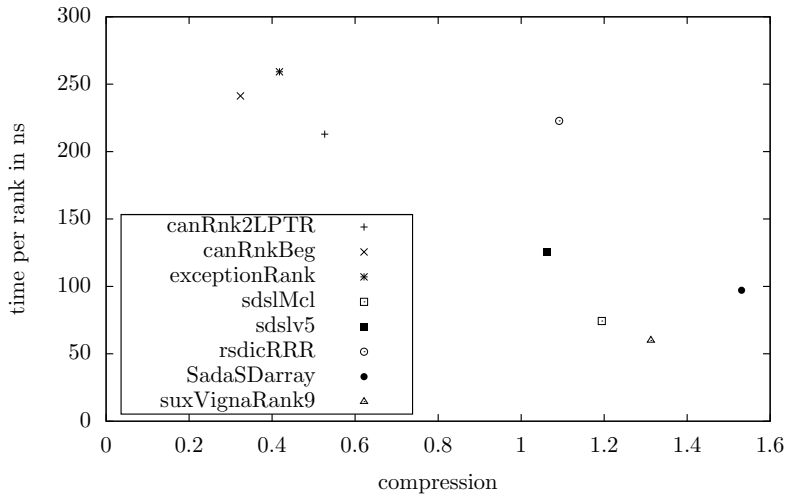
suxVignaRank9 Sebastiano Vigna's rank9-implementation [6].

Figure 4 shows the compression and query times for example texts with $n = 250$MB and $H_4 = 0.044$ or $H_4 = 0.112$, respectively (averaged over 10 repeats of 10^7 random queries). Other distributions of entropies yielded similar results. We can see that our data structures are significantly smaller than the others. At the extreme end, our smallest structure (canRnkBeg) uses less that 1/4 of the space used by the smallest existing structure (sdslv5). However, we trade this advantage for higher query times, in the example mentioned by a factor of 2.

Figure 4. Space/time tradeoff for various rank dictionaries. (**a**) $H_4 = 0.044$, $H_k \approx 1$ for $k \leq 3$. (**b**) $H_4 = 0.112$, $H_k \approx 1$ for $k \leq 3$.



(a)



(b)

6. Conclusions

We introduced new methods for compressing dense bit-vectors, while supporting rank/select queries, and showed that the methods perform well in practice. An interesting extension for future

work is engineering compressed balanced parentheses implementations, with full support for all navigational operations.

Author Contributions

Both authors developed the algorithms and wrote the manuscript. Implementation and testing was done by Kai Beskers.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Sadakane, K.; Navarro, G. Fully-Functional Succinct Trees. In Proceedings of Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, Austin, TX, USA, 17–19 January 2010; pp. 134–149.
2. Clark, D. Compact Pat Trees. PhD thesis, University of Waterloo, Ontario, Canada, 1996.
3. Munro, J.I. Tables. In Proceedings of 16th Foundations of Software Technology and Theoretical Computer Science, Hyderabad, India, 18–20 December 1996; pp. 37–42.
4. Raman, R.; Raman, V.; Rao, S.S. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA, 6–8 January 2002; pp. 233–242.
5. Okanohara, D.; Sadakane, K. Practical Entropy-Compressed Rank/Select Dictionary. In Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments, New Orleans, LA, USA, 6 January, 2007.
6. Vigna, S. Broadword implementation of rank/select queries. In *Experimental Algorithms*; Springer: Berlin Heidelberg, Germany, 2008; pp. 154–168.
7. Navarro, G.; Puglisi, S.J.; Valenzuela, D. Practical compressed document retrieval. In *Experimental Algorithms*; Springer: Berlin Heidelberg, Germany, 2011; pp. 193–205.
8. Kärkkäinen, J.; Kempa, D.; Puglisi, S.J. Hybrid Compression of Bitvectors for the FM-Index. In Proceedings of Data Compression Conference, Snowbird, UT, USA, 26–28 March 2014; pp. 302–311.
9. Ferragina, P.; Venturini, R. A Simple Storage Scheme for Strings Achieving Entropy Bounds. In Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, LA, USA, 7–9 January, 2007; pp. 690–696.
10. Navarro, G.; Providel, E. Fast, small, simple rank/select on bitmaps. In *Experimental Algorithms*; Springer: Berlin Heidelberg, Germany, 2012; pp. 295–306.
11. Jansson, J.; Sadakane, K.; Sung, W.K. Ultra-succinct representation of ordered trees with applications. *J. Comput. Syst. Sci.* **2012**, *78*, 619–631.
12. Manzini, G. An Analysis of the Burrows-Wheeler Transform. *J. ACM* **2001**, *48*, 407–430.
13. Gagie, T. A Note on Sequence Prediction over Lage Alphabets. *Algorithms* **2012**, *5*, 50–55.

14. Brisaboa, N.R.; Ladra, S.; Navarro, G. DACs: Bringing Direct Access to Variable-Length Codes. *Inf. Process. Manag.* **2013**, *49*, 392–404.
15. Fischer, J.; Heun, V.; Stühler, H.M. Practical Entropy Bounded Schemes for $O(1)$-Range Minimum Queries. In Proceedings of Data Compression Conference, Snowbird, UT, USA, 25–27 March 2008; pp. 272–281.
16. Barbay, J.; Claude, F.; Gagie, T.; Navarro, G.; Nekrich, Y. Efficient Fully-Compressed Sequence Representations. *Algorithmica* **2014**, *69*, 232–268.
17. Source code. Available online: http://ls11-www.cs.uni-dortmund.de/_media/fischer/research/diplcode.tar.gz (accessed on 30 October 2014).
18. Simon Gog's sdsl. Available online: https://github.com/simongog/sdsl (accessed on 30 October 2014).
19. Daisuke Okanohara's Rsdic. Available online: https://code.google.com/p/rsdic/ (accessed on 30 October 2014).