# Compressed bit vectors based on variable-to-fixed encodings

SEUNGBUM JO[1], STELIOS JOANNOU[2], DAISUKE OKANOHARA[3], RAJEEV RAMAN[2] AND SRINIVASA RAO SATTI[1]

[1]*Seoul National University, Korea*
[2]*University of Leicester, UK*
[3]*Preferred Infrastructure, Japan*
*Email: sbcho@tcs.snu.ac.kr, sj148@leicester.ac.uk, daisuke.okanohara@gmail.com, r.raman@le.ac.uk, ssrao@cse.snu.ac.kr*

**We consider practical implementations of *compressed* bitvectors, which support rank and select operations on a given bit-string, while storing the bit-string in compressed form. Our approach relies on *variable-to-fixed* encodings of the bit-string, an approach that has not yet been considered systematically for practical encodings of bitvectors. We show that this approach leads to fast practical implementations with low *redundancy* (i.e., the space used by the bitvector in addition to the compressed representation of the bit-string), and is a flexible and promising solution to the problem of supporting rank and select on moderately compressible bit-strings, such as those encountered in real-world applications.**

*Keywords: bitvector, rank and select, variable-to-fixed encoding, entropy.*

## 1. INTRODUCTION.

A *bitvector* is a fundamental building block of many space-efficient data structures. Given a bit-string $X$ of *length* $n$ with *weight* $m$ (i.e., with $m$ **1** bits), the aim is to pre-process $X$ to support the following operations, for any $b \in \{0, 1\}$:

- $\mathsf{rank}_b(X, i)$ returns the number of occurrences of $b$ in the first $i$ positions of $X$.
- $\mathsf{select}_b(X, i)$ returns the position of the $i$th $b$ in $X$.

These operations can be supported in $O(1)$ time using $n + o(n)$ bits of space [1]. If $X$ is a (uniformly) random bit-string, it cannot be compressed, and this space bound is therefore, in the worst case, optimal to within lower-order terms. However, bit-strings encountered in practical applications are often compressible, and many algorithmic applications use bitvectors on bit-strings that are constructed to be *sparse*—contain $m = o(n)$ **1**s—and such bit-strings are compressible to $o(n)$ bits. Starting from the work of [2, 3], there is now a rich theory of *compressed* bitvectors, which aim to use space approaching that used by a compressed representation of the bit-string, for many different measures of compressibility[4]. The most basic measures of compressibility are *density-sensitive*, i.e. they depend only upon the length $n$ and weight $m$ of the bit-string. These are the *information-theoretic minimum*, $B(n, m) \stackrel{\text{def}}{=} \lceil \lg \binom{n}{m} \rceil$ bits[5], and the *zeroth-order empirical entropy*, $H_0(X) \stackrel{\text{def}}{=} -\sum_{i=0}^{1} p_i \lg p_i$, where $p_1 = m/n$ and $p_0 = 1 - p_1$; the compressed bit-string size should then be $nH_0(X) + O(1)$ bits. Note that if $m = o(n)$ then $B(n, m) \approx nH_0(X) = o(n)$.

*Instance-sensitive* measures[6], where the compressibility of the string $X$ is a function of $X$, are more diverse, and include the *$k$-th order empirical entropy* $H_k$ and functions of the gaps between successive **1**s [4], or the size of the output produced by a grammar-based compressor to $X$. In general, such measures would show that a bit-string $X$ is at least as compressible as a density-sensitive measure on $X$.

Although there have been many papers on implementations of bitvectors [5, 6, 7, 8, 9, 10] (and some researchers have implemented bitvectors as part of more complex data structures), there are fewer papers on compressed bitvectors for sparse bit-strings. It should be noted that supporting $O(1)$-time rank/select operations using reasonable space is possible only when $m = n/(\lg n)^{O(1)}$ [11]. In this range, even the density-sensitive measure gives $O(m \lg(n/m)) = O(m \lg \lg n)$ bits, so a compressed bitvector is potentially significantly smaller than either an uncompressed bitvector, which takes $\Theta(n)$ bits, or viewing $X$ as the characteristic vector of a set and storing the set explicitly, which requires $O(m \lg n)$ bits. Such moderately sparse bit-strings are also of great practical interest. One focus of this paper is on representing such bit-strings.

The following authors have considered practical data structures for sparse bit-strings. Geary et al. [12] considered "uniformly" sparse bit-strings, but their techniques do not apply to general sparse bit-strings, and they do not

[4] As is common in the area of succinct and compressed data structures, we focus on *empirical* measures, i.e., those that are a function of the bitstring $X$ itself, rather than measures derived by postulating a probabilistic model for generating bit-strings.

[5] We use lg to denote the logarithm base 2.

[6] A related term, *data-aware*, is used in [4].

perform a stand-alone evaluation of their bitvector. Gupta et al. [4] considered very sparse bit-strings, and showed that instance-sensitive measures related to the $\gamma$ and $\delta$ codes outperform density-sensitive ones, but they did not report on moderately sparse bit-strings. Delpratt et al. [5] considered Golomb coding in the context of the $\mathsf{select}_1$ operation. Okanohara and Sadakane [13] performed arguably the first comprehensive evaluation, but focused mostly on the density-sensitive measures. Navarro et al. [14] considered rank and select on grammar-compressed bit-strings, but do not consider general sparse bit-strings. Navarro and Providel [15] also provide an implementation of compressed bitvectors. This, again, targets the density-sensitive measures. Very recently, Kärkkäinen et al. [16] presented a hybrid approach combining run-length encoding (RLE), raw encoding and explicit encoding, and showed good performance on a class of bit-strings obtained from text indexing applications.

In this paper we explore the use of *variable-to-fixed (V2F)* encodings of a bit-string, which have only been partially explored previously. Our results show that this approach leads to very compact and high-performance compressed bitvectors. Indeed, we give a theoretical basis for the low *redundancy* (wasted space) of the codes as well as that of the bitvector. An $\ell$-bit V2F code partitions the input bit-string into a concatenation of variable-length *phrases*. Each phrase, except the last one, is constrained to belong to a given dictionary $D$ of $\leq 2^\ell$ bit-strings; the last phrase is a non-null prefix of a dictionary entry. Once the input bit-string is parsed, each phrase is replaced by its position in the dictionary, stored as a $\ell$-bit *codeword*. V2F codes are studied in the data compression literature due to their desirable properties such as error-resilience [17], but it appears that there has not yet been a comprehensive investigation of V2F bitvectors. The class of V2F codes includes e.g. RLE and grammar-based compression, and it is possible that there are application-specific implementations of V2F bitvectors inside other data structures.

Our main conceptual contributions are as follows:

- We argue that in general, V2F coding is an effective approach to reduce the *redundancy* of the bitvector, or the difference between the compressed size of the bit-string and the size of the bit-vector data structure. The redundancy can dominate the space usage of compressed bitvectors: e.g. if $m = O(n/(\lg n)^2)$, the space usage of the compressed bitvector of [3], which is $B(n,m) + O(n \lg\lg n/\lg n)$ bits, is dominated by the redundancy. We show that for the density range of interest, V2F compressors give redundancy that is asymptotically smaller than the compressed size of the bit-string.
- In practice, we give an approach for density-sensitive encoding of a bit-vector that has a significantly lower (intrinsic) redundancy over that of Navarro and Providel [15] by using *Tunstall* codes [18]. Furthermore, we show that the Tunstall code always achieves $H_0$ empirical entropy with low redundancy (previously this was known only for random inputs).
- We give a new class of *enumerative* V2F codes. These codes generalize both Khodak's code [19, 20], a close relative of the Tunstall code, and RLE. Finally, a hybrid enumerative code which combines Khodak's code with RLE achieves excellent compression performance, even on bit-strings that are relatively incompressible by density-sensitive measures.
- We argue, as does Vigna [10], that practical implementations of select based on the method of "sampling" must address the issue of *long gaps*, which many implementations do not do. This is because in practice, guarding against a worst-case scenario for long gaps (using ideas which derive back to [1]) consumes a lot of space. Although it seems real-life bit-strings *can* have a number of reasonably long gaps, we note that the typical test (select a random **1**) is likely to give running times that are independent of the distribution of the underlying bit-vector. We propose a test that would "fairly" and "naturally" test the handling of a select implementation in the presence of long gaps, and show that implementations that do not guard against long gaps do indeed slow down.

The rest of this paper is structured as follows. Section 2 describes a general result on supporting rank and select operations on bit-strings compressed using V2F schemes. In Section 3, we describe the V2F compression algorithms, and evaluate their compression performance in practice. In Section 4 we propose an alternative practical approach to bit-vectors based on V2F compressors. In Section 5 describes the details of our implementation, and also the results from the experimental evaluation of V2F bit-vectors. Section 6 contains concluding remarks and future directions.

## 2. BIT-VECTORS USING V2F CODING

Recall that the redundancy of a compressed bitvector targeting a particular compressibility measure is the difference between the size of the bit string under that compressibility measure and the size of the bitvector. As noted earlier, we are focussing on the range $m = n/(\lg n)^{O(1)}$. Also as noted earlier, the redundancy of a standard solution [3], which is $O(n \lg\lg n/\lg n)$ bits, is *larger* than even the $H_0$ entropy of any bit-string with $m = o(n/\lg n)$. In particular, since the redundancy of [3] is independent of $m$, it does not allow one to benefit from the increasing sparseness of the bitvector.

Pătraşcu [21] showed that rank/select can be supported in $O(1)$ time using $B(n,m) + n/(\lg n)^c$ bits for any constant $c$, and that for $m = \Theta(n)$, this is optimal [22]. By choosing a large enough $c$, the redundancy can be made up to a poly-log factor smaller than the $H_0$ entropy of the bit-string for any bit-string with $m = n/(\lg n)^{O(1)}$. However, there is no evidence yet that the approach of [21] is feasible in practice. Another approach to low-redundancy compressed bitvectors achieves $B(n,m)+O(m(\lg\lg n)^2/\lg n)$ bits and $O(1)$ time for the range $m = n/(\lg n)^{O(1)}$ [7, 2]. Observe that for this range of $m$, $B(n,m) = O(m \lg\lg n/\lg n)$. Thus, the

redundancy of the approach of [7, 2] is roughly a log factor less than the $H_0$ entropy of the bit-string, even if it is not as low as Pătraşcu's. We now show that this holds in general for V2F codes under modest assumptions:

THEOREM 2.1. *Given a bit-string $X$ of $n$ bits encoded as $C$ codewords using a V2F code of $\ell$ bits each. Further assume that there is a data structure, which given a codeword $c$, supports* rank *and* select *in $O(1)$ time on the phrase $p(c)$ that the codeword $c$ stands for. Then we can support $\mathsf{select}_1(X, i)$ and $\mathsf{rank}_1(X, i)$ in $O(1)$ time using $C\ell + O(C \lg(n/C))$ bits, provided that $C = n/(\lg n)^{O(1)}$.*

REMARK 1. We will typically choose $\ell = \Theta(\lg n)$ bits. Thus, provided that $\lg(n/C) = o(\lg n)$, the redundancy will be smaller than the size of the compressed output, which is $C\ell$ bits. In particular, for $C = n/(\lg n)^{O(1)}$, $\lg(n/C) = O(\lg \lg n) = o(\lg n)$.

We use the following two lemmas from [23] that can be used to support rank and select operations on moderately dense bit strings (i.e., bit strings in which the number of zeros and ones is at most a poly-log factor smaller than the length of the string).

LEMMA 2.1 ([23]). *Given a bit-string of length $n$ and weight $m$, providing that $m \geq n/(\lg n)^c$ for some constant $c > 0$, we can support $\mathsf{rank}_1$ and $\mathsf{select}_1$ in $O(1)$ time using $B(n, m) + O(m)$ bits.*

LEMMA 2.2 ([23]). *Given integers $N_0, N_1 > 0$ and $n = N_0 + N_1$, such that $\min\{N_0, N_1\} \geq n/(\lg n)^c$ for some constant $c$, we can store a bit string $X$ with $n_0 \leq N_0$ 0s and $n_1 \leq N_1$ 1s, using $B(n, N_0) + O(\min\{N_0, N_1\})$ bits, such that $\mathsf{select}_0$ and $\mathsf{select}_1$ are supported in $O(1)$ time.*

*Proof. (of Theorem 2.1)* For any bit-string $s$, let $w(s)$ denote the weight of $s$, and for $i = 1, \ldots, C$, let $c_i$ denote the $i$-th codeword, and let $m = w(X)$. The data structure consists of two bitvectors on the following bit-strings:

- the *ones distribution* bit-string $OD = \mathbf{0}^{w(p(c_1))}\mathbf{1}\mathbf{0}^{w(p(c_2))}\mathbf{1}\ldots\mathbf{0}^{w(p(c_C))}\mathbf{1}$.
- the *phrase size* bit-string $PS = \mathbf{1}0^{|p(c_1)|-1}\mathbf{1}0^{|p(c_2)|-1}\mathbf{1}\ldots\mathbf{1}0^{|p(c_C)|-1}$.

It is easy to see that $|OD| = m+C, w(OD) = C, |PS| = n$ and $w(PS) = C$.

- To compute $\mathsf{select}_1(X, i)$, we first determine the number of codewords before the codeword in which the selected **1** lies as $j = \mathsf{rank}_1(OD, \mathsf{select}_0(OD, i))$. We then determine the total number of **1**s in $c_1, \ldots, c_j$ as $k = \mathsf{select}_1(OD, j) - j$, and the start position of $c_{j+1}$ in $X$ as $d = \mathsf{select}_1(PS, j + 1) - 1$. Finally, we select the $i-k$-th **1** in $p(c_{j+1})$, add $d$ to the answer and return.
- To compute $\mathsf{rank}_1(X, i)$, we first find the codeword $j$ in which the $i$-th position lies by $j = \mathsf{rank}_1(PS, i)$. We then determine $d$, the start position of $c_j$, and $k$, the number of **1**s in $c_1, \ldots, c_{j-1}$, as before, and return $k + \mathsf{rank}_1(p(c_j), i - d)$.

We store $OD$ using Lemma 2.2, which uses $O(C \lg(m/C))$ bits. In addition, we pad $OD$ to length $n$ by adding zeros at the end (so that the condition in Lemma 2.1 applies), and store the resulting bit-string as well as $PS$ using Lemma 2.1, which takes $O(C \lg(n/C))$ bits. □

## 3. V2F COMPRESSION ALGORITHMS FOR BIT-STRINGS

We now describe different V2F compression schemes that we use to compress the given bit-string $X$. Each of these schemes partitions $X$ into a sequence of variable-length *phrases*. Each phrase, except the last one, belongs to a *dictionary* of size $M = 2^\ell$ that is constructed from the source string. The dictionary entries are also referred to as *code words*. The compressed representation of $X$ simply consists of a sequence of $\ell$-bit codes (from the dictionary) corresponding to each phrase. The only difference between various compression algorithms is the way in which they construct the dictionary.

### 3.1. Tunstall code

For a given phrase length $L$, the Tunstall code is designed to maximize $E[L]$, the expected number of source letters per phrase for a memoryless source [18]. Given an input bit-string $X$, the dictionary constructed by Tunstall's algorithm can be represented as a full binary tree $T$ (i.e., every node has 0 or 2 children), which we refer to as the *Tunstall tree*. Each edge in $T$ corresponds to a bit, and each phrase corresponds to a leaf in $T$. The phrase corresponding to a leaf $u$ can be obtained by concatenating the symbols corresponding to the edges on the root-to-leaf path to $u$.

We now describe the algorithm to construct a Tunstall code for $X$ with $M = 2^\ell$ codewords. First, we define some terminology. Letting $n = |X|$, and $m$ be the weight of $X$, define $p_0 = 1 - m/n$ and $p_1 = m/n$. The probability[7] of a bit-string $b_1 b_2 \ldots b_l$ is defined to be $\prod_{i=1}^{l} p_{b_i}$. Each leaf in $T$ is labelled by the probability of the corresponding phrase, and each internal node is labelled by the sum of the probabilities of its children. The algorithm is as follows:

(1) Start with 2-level rooted tree with the root connected to two leaves, corresponding to **0** and **1**.
(2) Pick a leaf node which has the highest probability and grow two leaves on it.
(3) Repeat step (2) while the number of leaves in the tree is at most $M$.

It has long been known that the Tunstall code achieves zeroth-order entropy (defined appropriately) for random sources [18], and the size of the compressed representation has been shown to be asymptotically same as $nH_0$. We now show that the redundancy of the Tunstall code with respect to empirical entropy is also low.

THEOREM 3.1. *Given a bit-string $X$ with length $n$ and weight $m$, suppose that it is encoded using a Tunstall code*

[7] This is not a probability in the true sense, of course, since we are dealing with a given fixed bit-string $X$.

*with $M = 2^\ell$ codewords, constructed taking $p_0 = 1 - m/n$ and $p_1 = m/n$ as the probabilities of **0** and **1** respectively. Assume, without loss of generality, that $p_1 \leq p_0$ and further assume that $\ell = \Theta(\lg n)$ and $\lg(1/p_1) = o(\lg n)$. Then $C\ell \leq nH_0(X) + O(nH_0(X)\lg(1/p_1)/\ell)$.*

*Proof.* Say that a *final* leaf refers to a leaf of the Tunstall tree $T$ at the end of the algorithm. Observe that the probabilities of the leaves of $T$ at any stage of the algorithm add up to 1. Hence, while the number of leaves is less than $M$, there will always be a leaf with probability greater than $1/M$, so we will never expand a leaf with probability at most $1/M$. It follows that the minimum probability of a final leaf is greater than $p_1/M$. Let $p^*$ be the maximum probability of any final leaf. Since all final leaves are created by expanding leaves with probability $\geq p^*$, and at least one final leaf must have probability $\leq 1/M$, it follows that $p^* p_1 \leq 1/M$ or $p^* \leq 1/(p_1 M)$.

Suppose that the output of parsing $X$ according to the Tunstall code comprises $C$ codewords $c_1, c_2, \ldots, c_C$. Let $\Pr(c_i)$ denote the probability of the phrase of $c_i$. Then $-\lg \prod_{i=1}^{C} \Pr(c_i) = -\lg(p_0^{n-m} p_1^m) = nH_0(X)$. However, $\prod_{i=1}^{C} \Pr(c_i) \leq (1/(p_1 M))^C$ from the above, which gives $nH_0(X) \geq C\lg(p_1 M)$, or:

$$nH_0(X) + C\lg(1/p_1) \geq C\ell \tag{1}$$

With the above assumption on $p_1$, it is not hard to verify that $C\ell = O(nH_0(X))$, and plugging this back into Equation (1) we get that $C\ell \leq nH_0(X) + O(nH_0(X)\lg(1/p_1)/\ell)$. □

REMARK 2. 1. Since we assume $\lg(1/p_1) = o(\ell)$, the redundancy is a lower-order term.
2. Note that a similar argument shows that $C\ell \geq nH_0(X) - C\lg(1/p_1)$. In other words, the output of Tunstall coding is never much less than the empirical entropy.

Theorems 2.1 and 3.1 allow us to obtain a small improvement in redundancy over the bitvector of [7, Thm 2], which previously had the lowest known redundancy of any bitvector that does not use the (fairly complex) technique of *informative encoding* [7] or its successors [21].

COROLLARY 3.1. *Let $X$ be a bit-string with length $n$ and weight $m$. There is a bit-vector that supports* $\mathsf{rank}_1$ *and* $\mathsf{select}_1$ *in $O(1)$ time when $m = n/(\lg n)^{O(1)}$ and uses $nH_0(X) + O(m\frac{\lg(n/m)\lg\lg n}{\lg n})$ bits.*

*Proof.* Since $H_0(X) = O((m/n)\lg(n/m))$, from Theorem 3.1 the output of the Tunstall coding occupies $nH_0(X) + O(m(\lg(n/m))^2/\lg n)$ bits. To augment it with $\mathsf{rank}_1$ and $\mathsf{select}_1$, we use Theorem 2.1. The additional data structures use $O(C\lg(n/C)) = O\left(\frac{nH_0(X)}{\lg n}\lg\left(\frac{n\lg n}{nH_0(X)}\right)\right)$ bits. Simplifying, we get that the redundancy of the bitvector is $O\left(m\frac{\lg(n/m)}{\lg n}(\lg(n/m) + \lg\lg n)\right) = O(m\frac{\lg(n/m)\lg\lg n}{\lg n})$ bits.

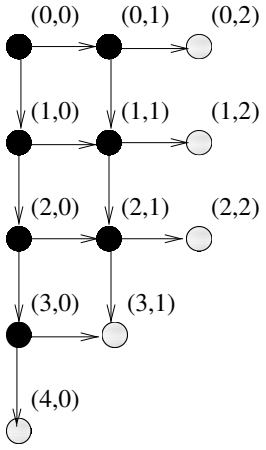
Finally, it only remains to explain how to do $\mathsf{rank}/\mathsf{select}$ on an individual phrase in $O(1)$ time. Taking the notation of Theorem 2.1, we create the concatenated bit-string $p(0)p(1)\ldots p(2^\ell - 1)$. The maximum length $L$ of an individual phrase must satisfy $(p_0)^L \geq p_1/M$, from which one can obtain that $L = O(n\lg n/m)$. Since $n/m = O(\lg n)^{O(1)}$, if we choose $\ell = (\lg n)/2$, the bit-string containing the concatenated phrases will be of size $O(n^{1/2+\epsilon})$, for any positive constant $\epsilon < 1/2$. By building a bit-vector on this bit-string and furthermore explicitly storing the start of each phrase, as well as the cumulative numbers of **1**s in this bit-string (using $O(2^\ell \lg n) = O(n^{1/2+\epsilon})$ bits), $\mathsf{rank}$ and $\mathsf{select}$ on individual phrases can be supported in $O(1)$ time. □

### 3.2. Enumerative codes

We define a class of *enumerative* codes as follows. An enumerative code can be specified as a (directed) graph on a subset of the vertices $(i, j)$, for $i \geq 0$ and $j \geq 0$. A vertex $(i, j)$ may either have no outgoing edges (be a *leaf*) or point to *both* vertices $(i + 1, j)$ and $(i, j + 1)$. Furthermore, a vertex $(i, j)$ is *complete* if either it has indegree 2, or either $i$ or $j$ is 0 (and its indegree is 1); and *incomplete* otherwise. All incomplete vertices must be leaves. Finally, the vertex $(0, 0)$ is always in the graph. Given such a graph, the code is specified as follows. For every complete leaf $(i, j)$ we allocate $\binom{i+j}{j}$ codewords, which code for all phrases with $i$ **0**s and $j$ **1**s. For every incomplete leaf $(i, j)$, if its (sole) predecessor is $(i, j - 1)$ then we allocate all $\binom{i+j-1}{j-1}$ codewords, which code for all phrases with $i$ **0**s and $j$ **1**s that end with a **1**. If its predecessor is $(i - 1, j)$, then we allocate all $\binom{i+j-1}{j}$ codewords, which code for all phrases with $i$ **0**s and $j$ **1**s that end with a **0** (see Fig. 1). Clearly, we must ensure that the total number of codewords is at most $2^\ell$.

Given such a graph, we parse the input-bit string as follows. Each phrase starts at $(0, 0)$. If we are currently at the non-leaf vertex $(i, j)$, upon reading a **1**, we move to $(i, j + 1)$; upon reading a **0**, we move to $(i + 1, j)$. By construction, both these vertices are in the graph. If we are at a complete leaf $(i, j)$ then we have so far read a phrase with $i$ **0**s and $j$ **1**s; since all possible $\binom{i+j}{j}$ such phrases have associated codewords, we choose the appropriate codeword, output it and restart from $(0, 0)$. Arriving at an incomplete leaf $(i, j)$ from $(i, j - 1)$, we must have read a phrase with $i$ **0**s and $j$ **1**s where the last bit is a **1**, so we output the appropriate codeword (the other case is similar), and restart from $(0, 0)$. We now give examples of enumerative codes.

*RLE.*
RLE is a special case of enumerative coding. To have codes for runs of **0**s and **1**s of length $1, \ldots, 2^{\ell-1}$, the corresponding graph contains the non-leaf vertices $(0, i)$ and $(i, 0)$, and the leaf vertices $(1, i)$ and $(i, 1)$ for $i = 1, \ldots, 2^{\ell-1} - 1$, together with the leaf vertices $(0, 2^{\ell-1})$ and $(2^{\ell-1}, 0)$. A codeword is thus assigned to each phrase of the form $\mathbf{0}^i\mathbf{1}$ and $\mathbf{1}^i\mathbf{0}$ for $i = 1, \ldots, 2^{\ell-1} - 1$; and one each for $\mathbf{0}^{2^{\ell-1}}$ and $\mathbf{1}^{2^{\ell-1}}$.




| Codeword | Phrase | Vertex in the graph |
|---|---|---|
| 0 | 11 | (0,2) |
| 1 | 011 | (1,2) |
| 2 | 101 | |
| 3 | 0011 | (2,2) |
| 4 | 0101 | |
| 5 | 1001 | |
| 6 | 0001 | (3,1) |
| 7 | 0010 | |
| 8 | 0100 | |
| 9 | 1000 | |
| 10 | 0000 | (4,0) |

FIGURE 1: An example of an (ad-hoc) enumerative code. The graph is given on the top (leaves shown gray) and the codewords, and their phrases, below.

*Khodak Code.*
The Khodak code [20] is is obtained by modifying Step (2) of the Tunstall algorithm in Section 3.1 to pick all the leaf nodes with highest probability and grow two leaves on all of them. It is known that every Khodak code is a Tunstall code, and that for the same dictionary size, the Khodak code has asymptotically the same average phrase length as the Tunstall code [20]. We show:

THEOREM 3.2. *Any Khodak code is an enumerative code.*

*Proof.* We first prove an auxiliary lemma that implies that, when the probabilities of zero and one are not the same, the dictionary constructed by the Khodak algorithm is a subset of the dictionary constructed by the Tunstall algorithm – by observing that the order in which the leaves are expanded in both the algorithms is the same; but Khodak algorithm may stop earlier if there is not enough space to expand all the leaves with same probability.

LEMMA 3.1. *For rational number $0 < d < 1, d \neq 1/2$, there are no nonnegative integers $x, y, z, w$ such that $x \neq z$ and $d^x(1-d)^y = d^z(1-d)^w$.*

*Proof.* Suppose that there exist nonnegative integers $x, y, z, w$ such that $x \neq z$ and $d^x(1-d)^y = d^z(1-d)^w$. Let $d = n/m$ for positive integers $m$ and $n$, such that $n$ and $m$ are relatively prime. Without loss of generality, assume that $d > \frac{1}{2}$ and $z+w \geq x+y$. Then it is easy to argue that $z \geq x$ and $y \geq w$. Thus, $m^{z+w-x-y}(m-n)^{y-w} = n^{z-x}$. If $m$ is even, then left side is even while right side is odd as $n$ is relatively prime to $m$. If $m$ is odd and $n$ is even, then left side is odd while right side is even. Finally, if both $m$ and $n$ are odd, the left side is even while the right side is odd. □

We now prove Theorem 3.2. Define $T^k$ as the tree whose leaves represent the phrases of the Khodak code (similar to $T$ in the Tunstall code). Next, let $T^k(i,j)$ be the set of all leaves in $T^k$ which represent the phrases with $i$ zeros and $j$ ones. We say that $T^k(i,j)$ is complete if $T^k$ contains all possible $(\binom{i+j}{i})$ phrases with $i$ zeros and $j$ ones (this is analogous to the definition of completeness of nodes in the enumerative codes). Now to prove Theorem 3.2, it is enough to prove the claim that if the Khodak algorithm expands the leaves in $T^k(i,j)$ then $T^k(i,j)$ is complete. The claim holds if the zero density is 1/2, because in this case, $T^k$ is always a complete binary tree (and each expansion step expands all the leaves). Now we assume that the one density is strictly larger than the zero density. Since for every step in the Khodak algorithm, $i$ and $j$ for expanding $T^k(i,j)$ are uniquely determined by the Lemma 3.1, the claim can be proved by the induction on the number of expansion steps taken by the Khodak algorithm.
**(Basis step)** In the first step, we expand the leaf $T^k(0,1)$ which is complete.
**(Inductive step)** Assume the hypothesis that the claim is true if the number of steps is at most $r$. In the $r+1$ step, suppose we expand $T^k(i,j)$ which is not complete. Note that $T^k(i,j)$ is generated by expanding $T^k(i,j-1)$ or $T^k(i-1,j)$. Since both $T^k(i,j-1)$ and $T^k(i-1,j)$ are expanded before $r+1$-th step (because they have the smaller probability than $T^k(i,j)$), by induction hypothesis, they are complete. But if we expand $T^k(i,j-1)$ and $T^k(i-1,j)$ which are complete, $T^k(i,j)$ becomes complete, contradicting the assumption. □

*Hybrid Enumerative Coding.*
To obtain better compression using enumerative encoding, we reserve a fraction of codewords for run-length codes, and use the remaining for the Khodak codewords. The run-length codewords are divided among **0** runs and **1** runs based on the densities of **0**s and **1**s. Details can be found in the next section.

### 3.3. LZW algorithm

Lempel-Ziv-Welch (LZW) algorithm [24] is a well-known dictionary-based compression algorithm. Our next compressor is a loose adaptation of the original. The main constraint is that while the dictionary constructed by the classic LZW algorithm has no fixed bound on its size, our approach uses a bounded-size dictionary (with $M$ codewords). Furthermore, the phrases output by the LZW parsing can be prefixes of each other, which is undesirable in our context. Our approach works as follows. Let $R$ be a

| Bit-string | Total Size ($10^6$ bits) | Density of **0**s | Max run-length of **0**s | Max run-length of **1**s | Tunstall | LZW | Enumerative code | | | $H_0$ | $Logsum$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | Khodak | RLE | Hybrid | | |
| factor9.6 | 812.0 | 0.964 | 2927 | 1 | 0.242 | **0.151** | 0.241 | 0.573 | 0.228 | 0.223 | 0.236 |
| proteins | 374.9 | 0.900 | 27376 | 1 | 0.466 | **0.104** | 0.475 | 1.585 | 0.546 | 0.466 | 0.484 |
| Z-Accidents | 903.3 | 0.996 | 4,250,294 | 1,315 | 0.045 | 0.035 | 0.046 | 0.058 | **0.030** | 0.041 | 0.111 |
| Z-Pumsb2 | 1661.1 | 0.999 | 1,138,613 | 7,774 | 0.007 | 0.006 | 0.007 | 0.007 | **0.004** | 0.008 | 0.097 |
| dblp_100 | 680.8 | 0.629 | 5,252,073 | 3,115,460 | 0.975 | 0.145 | 0.975 | 0.369 | **0.136** | 0.952 | 0.201 |
| english_100 | 784.3 | 0.710 | 2,142,856 | 743,383 | 0.869 | **0.285** | 0.869 | 0.771 | 0.306 | 0.868 | 0.305 |
| rand_dblp | 680.8 | 0.629 | 42 | 20 | **0.956** | 0.971 | **0.956** | 4.872 | **0.956** | 0.952 | 0.991 |
| rand_english | 784.3 | 0.710 | 50 | 17 | **0.874** | 0.891 | **0.874** | 4.146 | **0.874** | 0.868 | 0.910 |

TABLE 1: Characteristics of the test files (left), compression ratios (right).

copy of the input bit-string $S$.

1. Initialize the dictionary with the phrases **0** and **1**.
2. Find the longest prefix $P$ of $R$ that is in the dictionary, and remove $P$ from $R$.
3. If the dictionary has less than $M$ phrases, remove $P$ from the dictionary and add the phrases $P\mathbf{0}$ and $P\mathbf{1}$ to the dictionary and go to 2.

The algorithm terminates when either $M$ phrases have been added, or (exceptionally) when no prefix of $R$ matches a phrase in the dictionary. Now we assign each of these $M$ phrases a code, and use this dictionary to parse $S$ in a second pass. Also, unlike the original LZW algorithm, the modified algorithm requires both the compressed string as well as the dictionary for decompression.

### 3.4. Empirical evaluation of the compressors

We now describe the compression performance of the above algorithms. When designing the RLE codewords, we created RLE codewords for runs of **0**s and **1**s of lengths $1, \ldots, \rho_0$ and $1, \ldots, \rho_1$, where $\rho_0$ and $\rho_1$ are two values such that $\rho_0 + \rho_1 \leq 2^\ell$, and the balance between $\rho_0$ and $\rho_1$ is determined by (a) the density of **0**s and **1**s and (b) the maximum length of a run of **0**s or **1**s in the input. The precise algorithm is as follows.

Let $p_1 = m/n$ be the density of **1**s in the input bit-string, and let $p_0 = 1 - p_1$, and let $R_0$ and $R_1$ be the lengths of the longest runs of **0**s and **1**s in the input.

1. If $R_0 + R_1 \leq 2^\ell$ then $\rho_i = R_i$ for $i = 0, 1$.
2. If not, suppose that $R_0 > R_1$. If $R_0 > 2^\ell$, we set $\rho_0 = p_0 \cdot 2^\ell$. Otherwise we set $\rho_0 = R_0$, and $\rho_1 = \min\{2^\ell - \rho_0, R_1\}$.
3. If $R_0 \leq R_1$ we apply the symmetric version of step (2).

Note that the above procedure states the maximum length of runs of **0**s and 1s in the pure RLE code. In Hybrid code, set aside $2^{\ell-1}$ codes for run-length codes, and $2^{\ell-1}$ codes for Khodak codes. The run-length codes for the Hybrid algorithm are determined as above, but using $2^{\ell-1}$ instead of $2^\ell$ to decide $\rho_0$ and $\rho_1$.

#### *3.4.1. Test files*

Table 1 summarizes the characteristics of the bit-strings we used in our experiments. factor9.6 and proteins are obtained by parsing two XML files, and outputting $\mathbf{0}^i\mathbf{1}$ when a text node of length $i$ is encountered [5]. Z-Accidents and Z-Pumsb2 are used in a data structure for mining frequent patterns from benchmark data sets [25]. dblp_100 and english_100 are the FM-indices [26] of english.100MB and dblp.xml.100MB in Pizza&Chili Corpus [27] respectively (with 25% of positions in suffix array are sampled). We use the implementation of FM-index from fm-index++ [28]. rand_dblp and rand_english are generated at random, but setting their length and density to be the same as dblp_100 and english_100, respectively. The test bit-strings can be classified into four types based on their properties. The bit-strings factor 9.6 and proteins are fairly sparse but have relatively short runs of **0**s and **1**s. The bit-strings Z-Accidents and Z-Pumsb2 are very sparse and have some very long runs of **0**s. While dblp_100 and english_100 are quite dense, they have long runs of **0**s and **1**s; obviously, their randomly generated analogues do not have such long runs.

Table 1 shows the compression ratio achieved by the compressors on the test bit-strings (with $\ell = 16$, so each dictionary has $2^{16} = 65536$ codewords). We also give the $H_0$ values of the bit-strings and their $Logsum$ value, defined as follows. If we divide a given bit-string $X$ of length $n$ into fixed-size blocks $B_i$, $i = 1 \ldots \lceil n/63 \rceil$ of size 63 and each $B_i$ has weight $m(i)$, $Logsum(X)$ is defined as $\frac{1}{n}\sum_{i=1}^{\lceil n/63 \rceil} \lceil \lg(\binom{63}{m(i)}) \rceil + 6)$. $Logsum$ is an estimate of the standard density-sensitive approach to compressed bitvectors used in [3] and predecessors (referred to as RRR in what follows), based on the implementation of [15], which is optimized for low redundancy. Roughly, the idea is to view the input bit-string as chunks of length 63 bits, and to store each chunk with weight $m(i)$ by viewing it as a subset of the universe $\{1, \ldots, 63\}$ and storing its position in some canonical listing of all subsets of this universe of size $m(i)$ (which takes $\lceil \lg(\binom{63}{m(i)}) \rceil$ bits) plus the value $m(i)$ (which takes 6 bits). We make the following observations:

- There is a negligible difference in compression ratio between the Tunstall and Khodak codes. While Tunstall/Khodak are sometimes better than $H_0$, the variation is small, as implied by Remark 2.
- $Logsum$ is sometimes significantly better than $H_0$, e.g. in dblp_100 and english_100. The reason is that all-**0** and all-**1** blocks (which occur frequently in these bit-strings) compress far better than would be suggested

by the overall density of these bit-strings. However, the additive overhead of 6 bits per block means that $Logsum$'s performance is poor on bit-strings such as `Z-Pumsb2` and `Z-Accidents`, as well as the random bit-strings.

- Among the enumerative codes, Hybrid uniformly performed the best, even easily outperforming RLE on very sparse files. It is also often the overall best performer, but it does perform poorly relative to LZW on the XML bit-strings. We speculate that this is because in XML files, identical elements may have similar-length text nodes under them (e.g., a `zipcode` element will usually contain a text string of length 5) and LZW is able to capture such long-range patterns.

## 4. A SIMPLE APPROACH TO BIT-VECTORS BASED ON V2F COMPRESSION

In this section we describe a simple approach to supporting rank and select on V2F compressed bit-strings. The approach we describe is completely standard and is in fact used in many existing implementations such as that of [15]. We describe an overview in Section 4.1. However, when applied to V2F compressed bit-vectors, some changes need to be made: specifically, the choice of two key parameters ($B$ and $LG$ below) is crucial. This is dicussed in Section 4.2.

### 4.1. Overview

rank/select$_1$ *index.* For rank we divide the bit-string into *rank blocks* of size $B$, where the $i$-th block consists of the bits numbered $iB$ through $(i+1)B-1$. For each block, we store the position of the first codeword that intersects the block, the weight at the start of that codeword, and the absolute position in the bit-string where that codeword begins.

For select$_1$, we use the standard "sample and scan" approach [1] used by most select implementations including [10, 15, 29]. We choose a sampling parameter $s$ and logically divide the raw bit-string into *select blocks*, where the $i$-th select block begins at the position of the $is$-th **1**, and scan this select block to answer select$_1(j)$ queries for $j = is+1, \ldots, (i+1)s-1$ (Type 0 blocks). Again, as in the case of rank blocks, the starts of the select blocks are adjusted to the nearest phrase boundary. For Type 0 blocks, as with rank blocks, we store codeword/phrase alignment information, and cumulative information.

This approach does not guarantee a good time bound if the **1**s are distributed non-uniformly: in the worst case, one may need to scan $\Theta(n)$ bit positions. To mitigate this effect, we choose a threshold $LG$, and whenever a select block is larger than $LG$, it is called a *long gap* (Type 1 block) and is treated differently [1], by storing the positions of the **1**s in the block explicitly. Even though the number of long gaps is at most $n/LG$, $LG$ must be relatively high, as the cost of a long gap—$s$ words to store all **1** positions in a long gap—is also quite high. Choosing $LG$ too high, however, slows down select operations on **1**s in between sampled positions that are separated by just under $LG$ positions (*borderline* long gaps in what follows).

### 4.2. Parameter selection

An important difference between the "sample and scan" approach when using V2F codes is the choice of the parameters $B$ and $LG$. We discuss this in an asymptotic way first, and then explain how it affects our implementation in practice.

*Long Gaps: a Theoretical View.* In this paragraph, we illustrate the potential asymptotic gains by using V2F codes in the "sample and scan" approach to select$_1$. This illustration makes a number of mappings from current practical parameter choices to asymptotic functions, which by its very nature involves a certain amount of guesswork: we do not hope to convince everybody of these mappings. For simplicity we consider the case of a bit-string with weight $m = \Theta(n/\lg n)$, i.e. one whose compressed size is $O(n \lg\lg n/\lg n)$ bits, and assume that we wish to achieve a redundancy of $O(n/\lg n)$ bits, and a running time of $O(\lg n)$ for rank and select$_1$.[8]

This can be achieved by using a rank block size of $B = \Theta((\lg n)^2)$ bits, accessing $O(1)$ random memory locations and scanning $O(\lg n)$ consecutive memory words, where each word comprises $\Theta(\lg n)$ bits. For select$_1$, a typical sampling factor would be $s = \lg n$, so that the cost of pointers to the sampled locations is $O((m/s)\lg n) = O(n/\lg n)$ bits. We would choose $LG = \Theta((\lg n)^3)$, so that the cost of storing the locations of the **1**s in the at most $O(n/LG)$ long gaps is $O((n/LG)s\lg n) = O(n/\lg n)$ bits. The time to scan a borderline long gap is therefore $O((\lg n)^2)$.

However, the key observation is that borderline long gaps are highly compressible, and V2F codes exploit this better than the standard approaches to compressed bit-vectors [2, 3, 15]. These standard approaches would compress a borderline long gap to $\Theta((\lg n)^2 \lg\lg n)$ bits. Using (say) Tunstall or Khodak coding, a bit-string with length $L = \Theta((\lg n)^3)$ with weight $s = O(\lg n)$ is compressed to $O((\lg n)^2)$ bits or $O(\lg n)$ codewords, which can be scanned in $O(\lg n)$ time. Indeed, in this case, the Tunstall or Khodak code is based on the *global* density and encodes each **0** using $\lg(n/(n-m)) = O(1/\lg n)$ bits and each **1** using $\lg(n/m) = O(\lg\lg n)$ bits, so the compressed representation of a borderline long gap would indeed take $\Theta((\lg n)^2)$ bits. However, one can do better. For example, using RLE, or an appropriate asympotic generalization of Hybrid codes, borderline long gap can be compressed to just $O(\lg n \lg\lg n)$ bits, and can be scanned even faster than a normal gap.

*Selecting parameters in practice.* In practical implementations, we need to choose $B$ according to the compress-

[8] As there is evidence that due to address translation, the cost of a random memory access is $O(\lg n)$ [30], we argue that $O(\lg n)$ time per operation is the best achievable for succinct data structures.

| File name | $B$ | $s$ | $LG$ | Average #codewords per single block |
|---|---|---|---|---|
| factor9.6 | 4096 | 512 | 271641 | 39 |
| proteins | 2048 | 512 | 31290 | 37 |
| Z-Accidents | 16384 | 256 | 543308 | 32 |
| Z-Pumsb2 | 131072 | 256 | 4202659 | 37 |
| dblp_100 | 4096 | 4096 | 2039256 | 35 |
| english_100 | 2048 | 2048 | 135501 | 40 |
| rand_dblp | 512 | 512 | 31863 | 31 |
| rand_english | 512 | 512 | 33874 | 28 |

TABLE 2: Size of rank block and number of codewords spanned by a single rank block for test files.

ibility of the bit-string, so that each rank block (on average) spans a moderate number (about $30 \sim 50$) of codewords (see Table 2). Note that this is necessary: a fixed block size would either be unreasonably large for relatively incompressible files such as english_100, making operations slow, or too small for highly compressible files such as Z-Pumsb2, making the redundancy overwhelm the compressed bit-vector size.

The select block size is chosen so that the *number* of select blocks is roughly the same as the number of rank blocks, i.e. $n/B \sim m/s$ (so on average both rank and select queries scan similar numbers of codewords). Our value of $LG$ is chosen conservatively so that the worst-case cost of handling essentially $sn/LG$ long gaps is comparable to the space cost of the rank and select indices. It is not hard to see that the value of $LG$ chosen this way will need to be significantly larger than the "average" select block size.

One can see the effect of compression of long gaps on the two files english_100 and rand_english in Figure 2. We plot the select block size (in bits) against the number of codewords spanned by the select block, for select blocks that were $< LG$ in size[9]. Recall that we choose $B$ differently for rand_english and english_100 (see Table 2). Since we choose $s$ based on $B$, we have different values of $s = 512$ and 2048 for rand_english and english_100 respectively. Our aim below is not to compare the absolute values of the select block sizes, but rather capture relevant patterns.

The different block sizes lead to different values of $LG$ :$\sim$33800 and $\sim$135000 for rand_english and english_100 respectively. Only 437 select blocks exceeded $LG$ in size in english_100, and none in rand_english. The average sizes of a select block that is not a long gap are 1770 and 5570 in rand_english and english_100 respectively. In rand_english, the select block sizes are in a narrow range (as expected), and the number of codewords per block grows roughly linearly with the size (as expected). In english_100, the range is much larger, and we can see that the larger blocks tend to compress well. Indeed, the upper envelope, which gives a kind of "worst-case", suggests a "sub-linear" growth (it's not clear how to interpret Figure 2 asymptotically, hence the quote marks).

[9]Due to rounding to phrase boundaries, these values are approximate.

### 4.3. Supporting Rank/Select Operations.

*Scanning a Rank/Select Block.* To perform a rank operation, we need to scan a rank/select block to find the codeword that contains position $i$. The key loop in scanning a block is to (a) read a codeword at a time from the compressed bit string, (b) obtain (and accumulate) the length of its phrase and its weight, and (c) determine both the codeword where position $i$ lies, and the offset of position $i$ within that codeword. If the codeword length $\ell$ is at most $c \lg n$ bits, for some $c < 1$, then this can be done by table lookup in $O(1)$ time using a table of $o(n)$ bits. A similar loop can be used for the $\mathsf{select}_1$ operation, which will identify the codeword containing the position to be returned, and also the position to be selected within the phrase of the codeword.

*Operations on Codewords.* After scanning a block, the codeword containing the index $i$ (for a rank operation) or containing the position to be returned (for a select operation) is identified, and the global rank/select operation is reduced to a rank/select operation on the phrase representing this codeword. In most standard "compressed" bit-vectors, this final step can also be done in $O(1)$ time through table lookup, by tabulating answers to all queries on this final step, using a table of negligible size. However, in V2F bit-vectors, in principle it is possible that the phrase represented by a codeword is $\Theta(n)$ bits long, and such a table could then take $\Omega(n)$ words of space.

A number of solutions are possible for this step, depending on the compressor. For example, [14], which uses grammar-compression, navigates the grammar for this final step, which is not guaranteed to take $O(1)$ time. We propose a default implementation of rank on a codeword, which concatenates all phrases into a bit-string similar to Corollary 3.1 and stores it in a bit-vector supporting rank, together with two words per codeword to allow rank on an individual phrase to be reduced to rank on the bit-vector. Assuming, wlog, that all possible codewords appear at least once in the compressed output, the concatenation of these phrases is at most $n$ bits, and we can support rank on a codeword in $O(1)$ time (select can be handled similarly). Clearly, for individual compressors (such as RLE) one can consider tailor-made solutions.

## 5. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

### 5.1. Implementation

We implemented the data structures described in Sections 2 and 4 in C++. The implementations are described below.

#### *5.1.1. Overview.*

Our implementation has been structured into two independent parts: a bit-vector class that contains the indices for

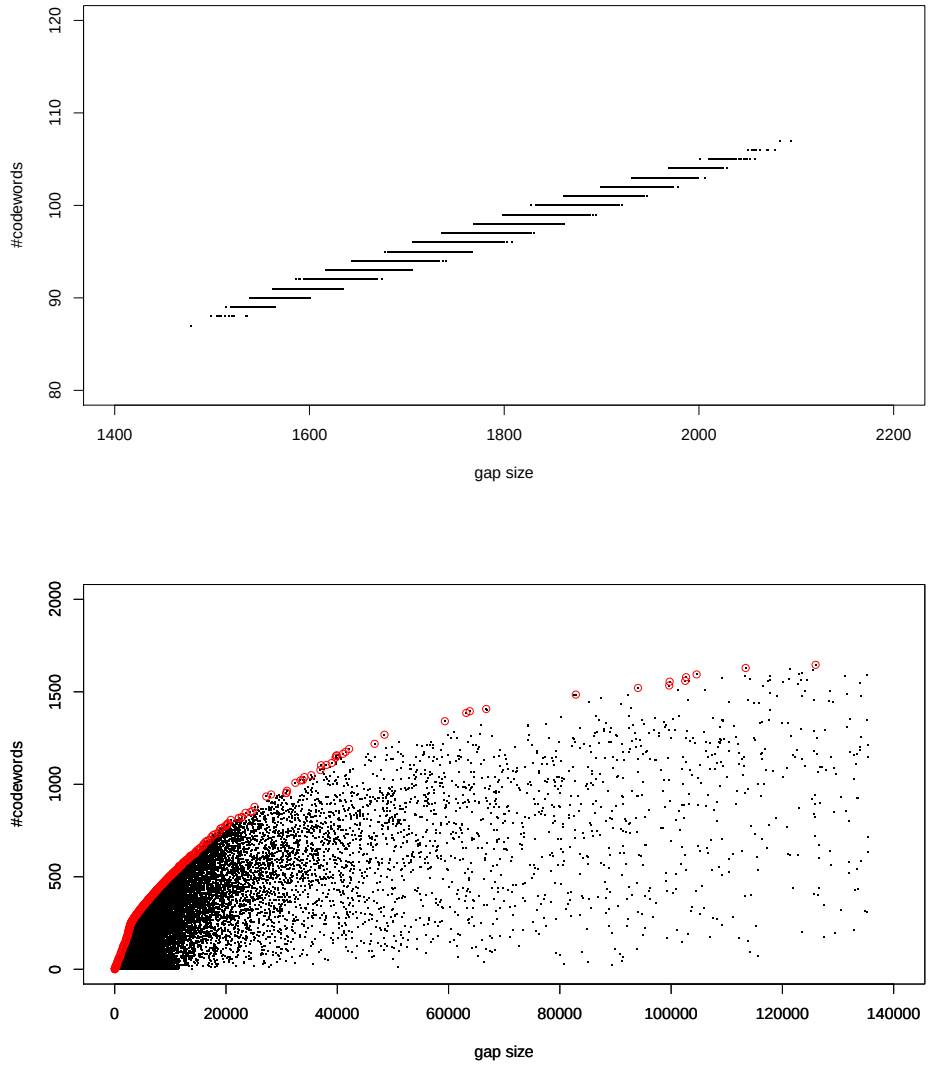



FIGURE 2: Scatter plot showing the size of select block size (gap size) for select blocks with size $< LG$, against the number of codewords spanned by the select block in `rand_english` (top) and `english_100`. In the latter plot, maximal points are circled in red. `rand_english` (`english_100`) uses $s = 512$ (2048) and $LG \sim 33800$ (135000).

rank and select, and a compressor-specific part that deals with individual codewords. The bit-vector class is (largely) independent of the compressor, and takes as input two files: one which contains the codewords and the phrases, and another which contains the sequence of codewords output by the compressor. The codeword class is responsible for reading these files and storing them in an array of the smallest "integer-like" type that will hold them. It also is responsible for creating the tables for scanning codewords in the approach of Section 4 and for supporting operations on individual codewords in both implementations.

*5.1.2. Implementation based on Section 2.*
Our implementation follows Theorem 2.1 closely. We implemented $OD$ and $PS$ using the compressed bit-vector implementations RRR and sdarray from the `sdsl-lite` library [29] and Okanohara's code [31] repectively, which have low redundancy on dense and sparse bit-strings respectively.

*5.1.3. Implementation based on Section 4.*
The implementation follows the description in Section 4. The bit-vector class has two main parts: arrays of rank/select$_1$ *blocks*, and a table for scanning codewords. We now describe each in detail. Each of the rank and select blocks uses 3 `long ints`, or 24 bytes. One of these is a pointer into the array of codewords and the other two contain cumulative information. For example, the rank block contains the number of **1**s up to the start of the codeword pointed to, as well as the total lengths of all the phrases of all the preceding codewords (stored as an offset).

As noted previously, the implementation of both opera-

| File name | Khodak | LZW | Enumerative code | |
|---|---|---|---|---|
| | | | Khodak | Hybrid |
| factor9.6 | 2.15% | 7.24% | 2.15% | 2.15% |
| proteins | 1.26% | 3.01% | 1.23% | 0.90% |
| Z-Accidents | 38.36% | 7.22% | 38.30% | 19.60% |
| Z-Pumsb2 | 668.61% | 200.14% | 667.98% | 73.25% |
| dblp_100 | 0.16% | 15.53% | 0.16% | 0.54% |
| english_100 | 0.17% | 52.20% | 0.17% | 0.22% |
| rand_dblp | 0.16% | 0.16% | 0.16% | 0.16% |
| rand_english | 0.17% | 0.16% | 0.17% | 0.17% |

TABLE 3: Total phrase length of test files (as % of compressed output), excluding RLE codewords

tions involves scanning the array of codewords, and accumulating the lengths of the phrases and their weights. This is done by table lookup, giving rise to the most important constraint on the size $\ell$ of a codeword. The table that contains this information has $2^\ell$ entries and must comfortably fit "into cache" (as the cache is likely to contain other data in real applications). On our machine, this suggests that $\ell \leq 16$. In practice, we choose $\ell = 16$ so the codewords are stored in an `unsigned short` array. The table then takes at most 512KB[10]. We have also considered $\ell = 8$ but the smaller cache footprint does not compensate for the greater number of codewords to be scanned, and $\ell = 8$ also typically has poorer compression.

*5.1.4. Implementation of Codeword Operations.* Having located the codeword containing the answer, we perform an appropriate rank/select on its phrase. The default implementation of rank on a codeword concatenates all phrases into a bit-string similar to Corollary 3.1 and stores it in a bit-vector supporting rank [29], together with two words per codeword to allow rank on an individual phrase to be reduced to rank on the bit-vector. select on each phrase is done by explicitly storing the positions of the **1**s in the phrase in an array. We estimate the *fixed* overhead to be about 4 `ints` per codeword, or 1MB overall. However, a potentially major variable overhead is the size of the rank phrase bit-vector and the phrase select array.

An obvious optimization is that for codes known to comprise runs of **0**s or **1**s, indicated by an additional *type* field stored in the length/weight table, we directly (and trivially) answer rank and select queries on the corresponding phrase. Table 3 shows the size of the resulting rank phrase bit-vector (the phrase select array is usually smaller). As suggested by Corollary 3.1, for Khodak codes, the size of the dictionary is negligible for relatively high-density bit-vectors. The overhead is much larger for the `Z-Accidents` and `Z-Pumsb2`, though Hybrid codes, which have many RLE codes, have smaller dictionaries than Khodak codes. Nevertheless, for very sparse bit-strings, it is clear that this naive approach is inappropriate.

[10]For the current compressors, no phrase can be longer than $2^\ell$ bits, so this could be reduced to 256KB.

### 5.2. Testing Methodology

The code was compiled with g++ 4.8.3 with optimization level 3, and tested on a 64-bit machine with 64GB RAM and an Intel Xeon E7450 6-core CPU clocked at 2.40GHz with $3 \times 3$ MB shared L2 caches and 12MB L3 cache, running Fedora Linux (kernel version 3.14.2). Tests were performed for the memory usage, and four tests for the speed of this structure, as follows.

*Memory Test.* To determine the true physical memory used by these data structures, we initialize them and then fork a process that allocates memory equal to the physical memory of the machine, which will result in all other processes' pages to be swapped out. Putting the forked process to sleep, we then perform rank and select operations and then measure the resident memory of the process.

We also measure the memory usage of our implementations by a self-reporting procedure which checks the total size of the main data structures using size reporting functions. Testing results shows that the measured memory size is larger than self-reported memory size because of the initial space used by OS and other variables in the program. But the difference between them does not exceed 10MB in all test files.

$\mathsf{rank}_1$ *Test.* To test the speed of rank, we perform $\mathsf{rank}_1(i)$ $n$ times, for random $i \in 1..n$.

*Random* $\mathsf{select}_1$ *Test.* Like the $\mathsf{rank}_1$ test, this test performs $\mathsf{select}_1(i)$ $n$ times, for $i$ selected randomly from $1..m$. Although "sample-and-scan" approach does not guarantee a good time bound, if the bit-string has long gaps, several implementations, including `RSDic`, do not guard against long gaps. However, their performance for random select tests on random bit-vectors (which typically don't have long gaps) is good. Vigna [10] proposed testing on pathological bit-strings to determine whether an implementation had good worst-case select performance. We note, however, that essentially *regardless* of the input bit-string, a random select test will not be able to distinguish between "sample-and-scan" bit-vectors, that deal with long gaps and those that don't. Specifically, observe that in any select block, the expected time taken to perform a select of one of the **1**s in this block, assuming a fairly even distribution of the **1**s within this block, is essentially *proportional to its length*. Since a random select accesses each select block with equal probability, it is not hard to see that the average running time of a random select is essentially *independent* of the distribution of select block lengths; i.e., a random select test is unlikely to distinguish between an easy bit-string and a pathological one. To address this issue, we propose a *hard* select test, described below.

*Hard* $\mathsf{select}_1$ *Test.* We perform $2^{19}$ random $\mathsf{rank}_1$ queries, and store the results in an array $Q$ of the same size (with repetitions). We then repeat the following, $n$ times: select

a random index $i$ in $Q$ and perform $\mathsf{select}_1(Q[i]+1)$. Doing this will select a **1** in a select block with probability proportional to the length of the select block (since the argument of the rank query falls in a select block with probability proportional to its length), and thus focusses on the harder select queries in a bit-string.

*Mixed Test.* We initialize an array $Q$ of size $2^{19}$ to values from random $\mathsf{rank}_1(i)$ as above. We cycle through the array and perform $\mathsf{select}_1(Q[i]+1)$ as above, but then do a $\mathsf{rank}_1(j)$ for a random index $j$, and store the result in $Q[i]$. Each such pair of rank and select operations is performed $n$ times.

### 5.3. Test setup

For our benchmarks, we choose the LZW code for XML bit-strings and Hybrid code for other bit-strings as V2F coders which gives the best compression ratio for their bit-strings. For comparison, we used Okanohara's rsdic code [32] (based on [15]), sdarray from Okanohara and Sadakane [13] and RRR from `sdsl-lite` [29]. We now describe the rsdic and sdarray bitvectors briefly.

The *Compressed Rank Select Dictionary*, rsdic is based on the structure proposed by Navarro and Providel [15]. It divides the bit-string $X$ into fixed-sized blocks of length $t = \lceil (\lg n)/2 \rceil$. The set of all possible blocks are divided into classes based on the number of 1's in the block. Hence, each block can be identified by a pair $(k, r)$, where $k$ is the class number which is simply the weight of the block, and $r$ is the index of the block in a table containing the set of all possible blocks in the class, in some canonical order, say, the lexicographic order. Thus, the representation of any block can be stored in $\lceil \lg(t+1) \rceil + \lceil \lg \binom{t}{k} \rceil$ bits. Also, one can rebuild a block "on-the-fly" using its representation, without storing any additional precomputed tables. For the sequence of blocks constituting the given bit-string $X$, it stores the first components (i.e., the weights of the blocks) in an array $K$, using fixed size entries of $\lceil \lg(t+1) \rceil$ bits each; the second components of all the blocks in the sequence are concatenated and stored as a bitvector $R$. To enable fast access into $R$, it first groups every $\lfloor \lg n \rfloor$ consecutive blocks into a superblock. For every superblock, it then stores a pointer into $R$ to point to the starting position of the representations corresponding to its blocks. In addition, we also store the rank up to the first bit in each superblock. To compute the rank for a given position, we first find the superblock containing the position, and do sequential search from the first block in the superblock. To support the select operation, we first perform a binary search to find the superblock containing the required position, and then scan the blocks within the superblock. The size of $R$ can be shown to be at most $nH_0(X) + o(n)$ bits, and the size of $K$ is $n\lceil \lg(t+1)/t \rceil = o(n)$ bits. Thus the space usage of rsdic is $nH_0(X) + o(n)$ bits.

Okanohara and Sadakane [13] proposed the sdarray which either stores an sarray when the given bit-string $X$ is *sparse*, or a darray when $X$ is *dense*. To describe the sarray, consider an array $x[0, \ldots, m-1]$ where $x[i]$ stores the position of the $(i+1)$-th 1-bit in $X$. We choose a parameter $t$, and store the lower $z = \lceil \lg t \rceil$ bits of each $x[i]$ in an array $L$ such that $L[i] = x[i] \bmod t$. The upper $w = \lfloor \lg(n/t) \rfloor$ bits of each $x[i]$ is encoded in unary to obtain a bit vector, $H$, of length $m+t$, along with auxiliary structures to support rank and select in $O(1)$ time on $H$. The operation select on $X$ can be supported in constant time by finding the upper bits using the select operation on $H$, and accessing the lower bits from the array $L$. To support $\mathsf{rank}(i)$ on $X$, we first find a smallest element whose position is greater than $\lceil i/2^w \rceil \cdot 2^w$ using select on $H$ and count number of ones sequentially from here using $H$ and $L$. By choosing $t = 1.44m$, the total size of sarray becomes $1.92m + m(\lg(n/m)) + o(m)$ bits.

The construction of darray first divides the given bit-string $X$ into blocks of $L$ ones each, and constructs an array $P[0, \ldots, n/(L-1)]$ such that $P[i]$ stores the position of $iL$-th one in $X$. These blocks are represented based on their length. If the length of a block is more than $(\lg n)^4$, it is represented by storing the positions of all the ones in it. Otherwise, its representation consists of the position of every $(\lg n)$-th one in the block, using $L(\lg L)/\lg n$ bits. To support $\mathsf{select}(i)$, we first find the block that contains the answer using $P$. If this block is longer than $(\lg n)^4$, we can read the answer from its representation. Otherwise, we use the representation of the block to find a sequence of $(\lg n)$ positions, one of which corresponds to the required answer, and scan the sequence to find the answer. The $rank$ operation is supported using an approach similar to that of rsdic. By choosing $L = (\lg n)^2$, the size of darray can be limited to $n + o(n)$ bits, including the bit-string $X$.

### 5.4. Results of Empirical Evaluation

*Memory test.* Practical implementation of bitvectors based on V2F used significantly less memory than the competition in most cases (see Fig. 4(a)); the exception is sdarray with `Z-Accidents` and `Z-Pumsb2` and RRR with `rand_dblp`, `rand_english` and `english_100`. In the former case (for `Z-Accidents` and `Z-Pumsb2` files), despite the V2F compressed bit-string being significantly less than $H_0$, the compressed size is so small that the fixed overhead of the phrase rank/select structure dominates. Also for latter three files, their V2F compression ratios are close to $Logsum$, and the overhead in the bitvector based on V2F implementations is more than that of RRR. Figure 3 shows the results of the memory test.

Although the redundancy in Theorem 2.1 is less than (little-oh of) the compressed output size in theory, for the implementation based on Theorem 2.1, the space overhead is $1 \sim 3$ times more than the compressed output size, in all the test files except `Z-Accidents` and `Z-Pumsb2` (for which the compressed output size is significantly smaller). This is because the $O(\lg(n/C))$ term in the redundancy can be larger than the codeword size even though the value of $\lg(n/C)$ in these files is $4 \sim 6$, which is less than the codeword size.

| File name | Compressed Bit-string size ratio | Overall space ratio | $\mathsf{rank}_1$ test ratio | Random $\mathsf{select}_1$ test ratio | Hard $\mathsf{select}_1$ test ratio | Mixed test ratio |
|---|---|---|---|---|---|---|
| factor9.6 | 189.34% | 85.94% | 129.76% | 208.89% | 220.93% | 223.63% |
| proteins | 341.95% | 65.69% | 115.79% | 210.81% | 220.59% | 200.93% |
| Z-Accidents | 865.76% | 166.42% | 351.05% | 208.33% | 555.56% | 480.93% |
| Z-Pumsb2 | 5822.67% | 366.41% | 495.73% | 214.29% | 229.17% | 263.24% |
| dblp_100 | 288.90% | 138.55% | 192.57% | 251.75% | 164.43% | 237.82% |
| rand_dblp | 161.84% | 103.10% | 114.65% | 166.67% | 198.39% | 182.06% |

TABLE 4: Comparison between bitvector based on V2F and bitvectors of Beskers and Fischer [33].

| File name | Space Usage ratio excluding select index | $\mathsf{rank}_1$ test ratio |
|---|---|---|
| factor9.6 | 156.01% | 15.46% |
| proteins | 204.24% | 15.24% |
| Z-Accidents | 226.70% | 56.99% |
| Z-Pumsb2 | 689.03% | 89.23% |
| dblp_100 | 126.24% | 24.40% |
| english_100 | 97.93% | 17.49% |
| rand_dblp | 82.28% | 9.73% |
| rand_english | 87.23% | 11.79% |

TABLE 5: Comparison between bitvector based on V2F and bitvectors of Kärkkäinen et al. [16].

$\mathsf{rank}_1$ *test.* Generally speaking, apart from sdarray, which is not optimized for rank, all others are comparably fast. However, sdarray does better than rsdic on the Z-vectors, possibly because it fits in cache due to its much lower memory usage, and the V2F bitvectors and RRR both do relatively poorly on the random bit-strings (see Fig. 4(a)).

*Random* $\mathsf{select}_1$ *test.* As expected, sdarray is generally the fastest, but loses out a little on the FM-index files, as it cannot compress them. The V2F bitvector is the second-best, and is very close to the best, in most cases, but performs slightly worse on the random files (see Fig. 4(b)). rsdic and RRR show significant weakness on the Z-vectors and XML bit-strings respectively.

*Hard* $\mathsf{select}_1$ *test.* rsdic is the only bit-vector that does not guard against long gaps, and performs very poorly (up to 20 times slower) on three of the input files (see Fig. 4(c)). V2F bitvectors do the hard $\mathsf{select}_1$ test at roughly the same speed as the random $\mathsf{select}_1$, and thus demonstrate their resilience.

*Mixed test.* The V2F bitvectors are the best overall performers in this test, since they show good performance for both the hard select test and the rank test.

Figure 4 shows the results for the above four tests.

### 5.5. Recent Work

For the sake of completeness, we have performed a preliminary comparison with the bitvectors of Kärkkäinen et al. [16] and Beskers and Fischer [33]. These works were performed independently of ours and both were published on or after the preliminary publication of our work in [34].

We first compare with Beskers and Fischer's work. First, in what concerns the space usage, they (conveniently) report the sizes of their compressed bit-string and the rankselect indices separately[11]. Broadly speaking the compression ratios of their compressed bit-string are not as good as ours. In particular, the compression ratio for the highly compressible Z-Accidents and Z-Pumsb2 are significantly worse. However, the overall space usage is a lot closer, between 60% of our usage and 140%, for most files; only in Z-Pumsb2 is their space usage significantly worse. Coming now to the running time, we note that their approach to select is to sample, use the samples to divide the bit-string into select blocks, and then perform binary search in the select block. This means that the performance is less affected by long gaps than a pure sample and scan approach. Nevertheless, we see that the performance is indeed affected. For rank as well, the speed is significantly slower, in particular, the large size of the compressed bit-string in Z-Pumsb2 results in rank being nearly 5 times slower.

Coming now to the work of Kärkkäinen et al. [16], there are significant differences in the two approaches, which render the comparison a little bit challenging. Firstly, Kärkkäinen et al. [16] favour speed and simplicity over compression. Secondly, they do not support select, and certain aspects of their implementation (such as the use of hardware pop-count) transfer less easily to select. On the other hand, we have consciously tried (in our parameter selection for example) to ensure rank and select are comparable in speeds. In Table 5 we add together the size of our compressed bit-string and the size of the rank index and compare to the reported memory usage of Kärkkäinen et al. As a result, the performance is very different. In terms of memory usage, our implementations (excluding any memory for select) are comparable on the files rand_dblp and dblp_100 (from 18% more memory on rand_dblp to 18% less memory on dblp_100). On the other files, we perform significantly better, from 36% less memory for factor9.6 to 85% less memory (i.e about 7 times smaller) for Z-Pumsb2. However, the speed of our rank ranges from comparable with theirs (for the highly compressible file Z-Pumsb2) to 9-11 times slower (for the incompressible files rand_dblp and rand_englsih).

[11] We observe that their code appears to interchange the sizes of the bit-string and the indices. We note this from the fact that the reported sizes of the indices would otherwise be several times larger than the reported sizes of the compressed bit-strings, and also that the random bit-strings rand_dblp and rand_english would compress well beyond their zero-th order entropy.

## 6. CONCLUDING REMARKS AND FUTURE WORK

This paper has, for the first time as far as we are aware, carefully investigated V2F compressors as a basis for bitvectors. We have shown how V2F bitvectors can lead to simple bitvectors with good redundancy. Empirical testing of an implementation, which albeit differs considerably from the theoretical proposals, shows that low memory usage and good, robust speed performance can be obtained via V2F compressors.

There is much room for further investigation. For instance, the naive approach to operations on individual phrases, as well as the relatively simple approach to supporting rank/select, leads to an overhead that is rather high for highly compressible bit-strings (admittedly, these are so sparse as to test the boundaries of our stated aim of targeting "moderately compressible" bit-strings). This could be overcome by adhering more closely to the theoretical result, and making greater use of on-the-fly decoding, both of which are very much in our plans in the near future. Apart from the Tunstall/Khodak/Enumerative codes, we have not explored V2F codes in any non-trivial way. Much more work is clearly possible along this axis as well.

**Acknowledgements.** A poster based on this work was presented at the DCC 2014 conference [34].

## REFERENCES

[1] Clark, D. R. and Munro, J. I. (1996) Efficient suffix trees on secondary storage (extended abstract). *Proceedings of 7th Annual Symposium on Discrete Algorithms (SODA), 28-30 January, Atlanta, Georgia.*, pp. 383–391. SIAM.

[2] Pagh, R. (2001) Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.*, **31**, 353–363.

[3] Raman, R., Raman, V., and Satti, S. R. (2007) Succinct indexable dictionaries with applications to encoding $k$-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, **3**, Article 43, 25pp.

[4] Gupta, A., Hon, W.-K., Shah, R., and Vitter, J. S. (2007) Compressed data structures: Dictionaries and data-aware measures. *Theor. Comput. Sci.*, **387**, 313–331.

[5] Delpratt, O., Rahman, N., and Raman, R. (2007) Compressed prefix sums. *Proceedings of 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), Harrachov, Czech Republic, January 20-26*, pp. 235–247. Springer.

[6] Delpratt, O., Rahman, N., and Raman, R. (2006) Engineering the louds succinct tree representation. *Proceedings of 5th International Workshop on Experimental Algorithms (WEA), Cala Galdana, Menorca, Spain, May 24-27*, pp. 134–145. Springer.

[7] Golynski, A., Grossi, R., Gupta, A., Raman, R., and Rao, S. S. (2007) On the size of succinct indices. *Proceedings of 15th Annual European Symposium on Algorithms ESA, Eilat, Israel, October 8-10*, pp. 371–382.

[8] Gonzáalez, R., Grabowski, S., Mäkinen, V., and Navarro., G. (2005) Practical implementation of rank and select queries. *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms*, pp. 27–38. CTI Press and Ellinika Grammata.

[9] Kim, D. K., Na, J. C., Kim, J. E., and Park, K. (2005) Efficient implementation of rank and select functions for succinct representation. *Proceedings of 4th International Workshop on Experimental Algorithms (WEA), Santorini Island, Greece, May 10-13*, pp. 315–327. Springer.

[10] Vigna, S. (2008) Broadword implementation of rank/select queries. *Proceedings of 7th International Workshop on Experimental Algorithms (WEA), Provincetown, MA, USA, May 30-June 1*, pp. 154–168. Springer.

[11] Patrascu, M. and Thorup, M. (2006) Time-space trade-offs for predecessor search. *Proceedings of 38th Annual ACM Symposium on Theory of Computing (STOC), Seattle, WA, USA, May 21-23*, pp. 232–240. SIAM.

[12] Geary, R. F., Rahman, N., Raman, R., and Raman, V. (2006) A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, **368**, 231–246.

[13] Okanohara, D. and Sadakane, K. (2007) Practical entropy-compressed rank/select dictionary. *Proceedings of 9th Workshop on Algorithm Engineering and Experiments (ALENEX), New Orleans, Louisiana, USA, January 6*. SIAM.

[14] Navarro, G., Puglisi, S. J., and Valenzuela, D. (2014) General document retrieval in compact space. *ACM Journal of Experimental Algorithmics*, **19**.

[15] Navarro, G. and Providel, E. (2012) Fast, small, simple rank/select on bitmaps. *Proceedings of 11th International Symposium Experimental Algorithms (SEA), Bordeaux, France, June 7-9*, pp. 295–306. Springer.

[16] Kärkkäinen, J., Kempa, D., and Puglisi, S. J. (2014) Hybrid compression of bitvectors for the FM-index. *Proceedings of Data Compression Conference, (DCC), Snowbird, UT, USA, 26-28 March*, pp. 302–311. IEEE.

[17] Cooper, D. and Lynch, M. F. (1982) Text compression using variable-to fixed-length encodings. *JASIS*, **33**, 18–31.

[18] Tunstall, B. P. (1967) Synthesis of noiseless compression codes. PhD thesis Georgia Tech.

[19] Khodak, G. L. (1969) Connection between redundancy and average delay of fixed-length coding. *All-Union Conf. Problems of Theoretical Cybernetics*. (in Russian).

[20] Drmota, M., Reznik, Y. A., and Szpankowski, W. (2010) Tunstall code, khodak variations, and random walks. *IEEE Transactions on Information Theory*, **56**, 2928–2937.

[21] Patrascu, M. (2008) Succincter. *Proceedings of 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS), October 25-28, Philadelphia, PA, USA*, pp. 305–313. IEEE Computer Society.

[22] Patrascu, M. and Viola, E. (2010) Cell-probe lower bounds for succinct partial sums. *Proceedings of 21st Annual Symposium on Discrete Algorithms (SODA), Austin, Texas, USA, January 17-19*, pp. 117–122. SIAM.

[23] Golynski, A., Orlandi, A., Raman, R., and Rao, S. S. (2014) Optimal indexes for sparse bit vectors. *Algorithmica*, **69**, 906–924.

[24] Ziv, J. and Lempel, A. (1978) Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, **24**, 530–536.

[25] Goethals, B. and Zaki, M. (2004). Fimi: Frequent itemset mining dataset repository. `http://fimi.ua.ac.be/`.

[26] Ferragina, P. and Manzini, G. (2001) An experimental study of an opportunistic index. *Proceedings of 12th Annual*

*Symposium on Discrete Algorithms (SODA), January 7-9, Washington, DC, USA.*, pp. 269–278. SIAM.

[27] Ferragina, P. and Navarro, G. Pizza&chili corpus. `http://pizzachili.dcc.uchile.cl/texts.html`.

[28] Tabei, Y. fmindex-plus-plus. `https://code.google.com/p/fmindex-plus-plus/`.

[29] Gog, S., Beller, T., Moffat, A., and Petri, M. (2013) From theory to practice: Plug and play with succinct data structures. *CoRR*, **abs/1311.1249**.

[30] Jurkiewicz, T. and Mehlhorn, K. (2013) The cost of address translation. *Proceedings of 15th Meeting on Algorithm Engineering and Experiments (ALENEX), New Orleans, Louisiana, USA, January 7*, pp. 148–162. SIAM.

[31] Okanohara, D. (2010). Succinct data structure for sparse arrays and integers. `https://code.google.com/p/sdarray/`.

[32] Okanohara, D. (2012). Compressed rank select dictionary. `https://code.google.com/p/rsdic/`. [Online; accessed 1-March-2013].

[33] Beskers, K. and Fischer, J. (2014) High-order entropy compressed bit vectors with rank/select. *Algorithms*, **7**, 608–620.

[34] Jo, S., Joannou, S., Okanohara, D., Raman, R., and Satti, S. R. (2014) Compressed bit vectors based on variable-to-fixed encodings. *Proceedings of Data Compression Conference, (DCC), Snowbird, UT, USA, 26-28 March* 409. IEEE.

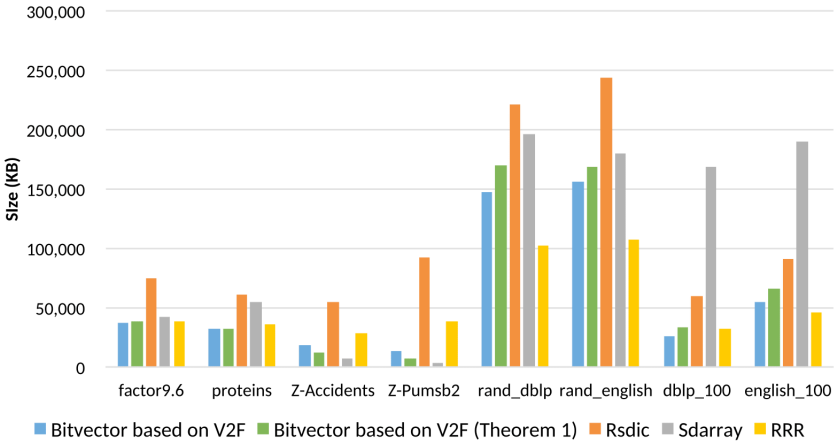
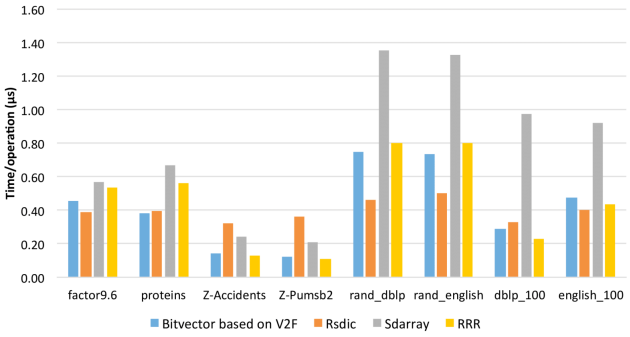
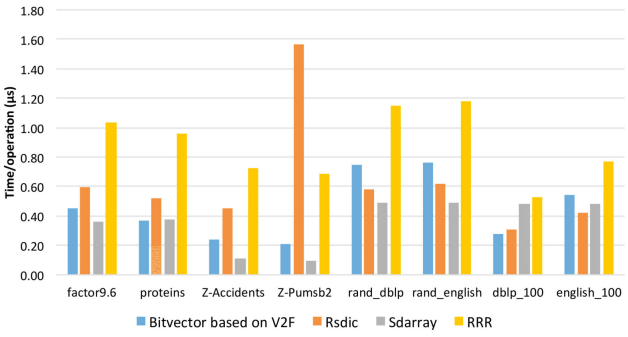


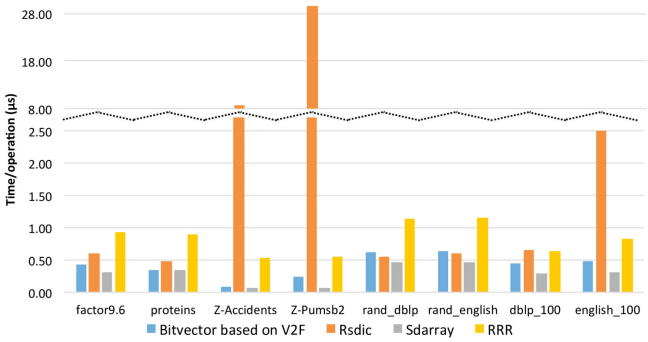

FIGURE 3: Memory test: Y-axis shows RSS size in KB



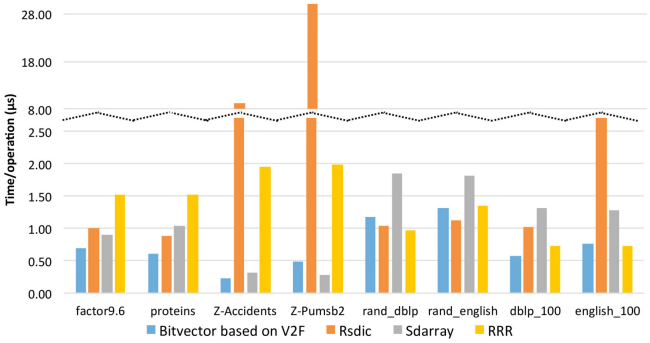

(a) $\mathsf{rank}_1$ test




(b) Random $\mathsf{select}_1$ test




(c) Hard $\mathsf{select}_1$ test




(d) Mixed test

FIGURE 4: (a) $\mathsf{rank}_1$ test, (b) Random $\mathsf{select}_1$ test, (c) Hard $\mathsf{select}_1$ test (d) Mixed test: Y-axis shows the running time per operation in microseconds. X-axis shows the test bit-strings in the order `factor9.6`, `proteins`, `Z-Accidents`, `Z-Pumsb2`, `rand_dblp`, `rand_english`, `dblp_100` and `english_100`. The order of bitvectors for each bit-string is bitvector based on V2F, rsdic, sdarray and RRR