Practical Rank/Select Queries over Arbitrary Sequences *

Francisco Claude and Gonzalo Navarro

Department of Computer Science, Universidad de Chile
{fclaude,gnavarro}@dcc.uchile.cl

Abstract. We present a practical study on the compact representation of sequences supporting *rank*, *select*, and *access* queries. While there are several theoretical solutions to the problem, only a few have been tried out, and there is little idea on how the others would perform, especially in the case of sequences with very large alphabets. We first present a new practical implementation of the compressed representation for bit sequences proposed by Raman, Raman, and Rao [SODA 2002], that is competitive with the existing ones when the sequences are not too compressible. It also has nice local compression properties, and we show that this makes it an excellent tool for compressed text indexing in combination with the Burrows-Wheeler transform. This shows the practicality of a recent theoretical proposal [Mäkinen and Navarro, SPIRE 2007], achieving spaces never seen before. Second, for general sequences, we tune wavelet trees for the case of very large alphabets, by removing their pointer information. We show that this gives an excellent solution for representing a sequence within zero-order entropy space, in cases where the large alphabet poses a serious challenge to typical encoding methods. We also present the first implementation of Golynski et al.'s representation [SODA 2006], which offers another interesting time/space trade-off.

1 Introduction

During the past years, there has been an increasing interest in compressed data structures, since they allow one to manipulate more data in main memory, waiving the painful overcost of accessing the disk. Apart from saving space, this largely improves the execution time of an algorithm even when the compressed version makes several times more operations than the uncompressed counterpart. This applies, albeit less sharply, to all levels of memory hierarchy.

Probably the most basic tool, used in virtually all compressed data structures, is the sequence of symbols supporting *rank*, *select* and *access*. *Rank(a,i)* counts the number of as until position i. *Select(a,i)* finds the position of the i-th occurrence of a in the sequence. *Access(i)* returns the symbol at position i in the sequence. The most basic case is when the sequence is drawn from a binary alphabet. Theoretically and practically appealing solutions have been proposed

* Partially funded by Fondecyt Grant 1-080019 (Chile).

for this case, achieving space close to the zero-order entropy of the sequence and good time performance.

The general case, when the alphabet has size $\sigma > 2$, has many applications to compressed representation of texts [9, 8, 14], trees [18, 1], graphs [5], binary relations [1], etc. For example, it has been shown [7, 14] that a compressed representation of a sequence that supports *rank* and *access* suffices to build a compressed full-text index if combined with the Burrows-Wheeler transform (BWT) [3]. Many solutions for general sequences have been proposed [9, 8], but as far as we know only some implementations of wavelet trees [11, 6] have been tried out.

In this paper we propose and study practical implementations of sequences. Our first contribution is a compressed representation of binary sequences based on Raman, Raman, and Rao's (RRR) [18] theoretical proposal. We combine faithful implementation of the theory with commonsense decisions. The result is compared, on uniformly distributed bitmaps, with a number of very well-engineered implementations for compressible binary sequences [16], and found to be competitive when the sequence is not too compressible, that is, when the fraction of 1s raises over 10%.

Still this result does not serve to illustrate the local compressibility property of RRR data structure, that is, it adapts well to local variations in the sequence. Mäkinen and Navarro [12] showed that the theoretical properties of RRR structure makes it an excellent alternative for full-text indexing: By combining it with the BWT, a high-order compressed self-index is immediately obtained, without all the extra sophistications used up to then [14]. In this paper we show experimentally that the proposed combination does work well in practice, achieving (sometimes significantly) better space than any other existing self-index, with moderate or no slowdown. The other compressed bitmap representations do not achieve this result: the bitmaps are globally balanced, but they exhibit long runs of 0s or 1s that only the RRR technique exploits so efficiently.

We then turn our attention to representing sequences over larger alphabets. Huffman-shaped wavelet trees have been used to approach zero-order compression of sequences [11, 6]. This requires $O(\sigma \log n)$ bits for the symbol table and the tree pointers, where n is the sequence length. On large alphabets, this factor can be prohibitive in space and ruin the compression ratios. We propose an alternative representation that uses no (or just $\log \sigma$) pointers, and concatenates all the bitmaps of the wavelet tree levelwise. As far as we know, no previous direct solution to *select* over this representation existed. Combined with our compressed bitmap representation, the result is an extremely useful tool to represent a sequence up to its zero-order entropy, disregarding any problem related to alphabet size. We illustrate this point by improving an existing result on graph compression [5], in a case where no other considered technique succeeds.

Finally, we present the (as far as we know) first implementation of Golynski et al.'s data structure for sequences [9], again combining faithful implementation of the theory with common sense. The result is a representation that does not compress the sequence, yet it answers queries very fast without using too much extra space. In particular, its performance over a sequence of word identifiers

provides a sequence representation that uses about 70% of the original space of the text (in character form) and gives the same functionality of an inverted index. It might become an interesting alternative to recent wavelet-tree-based proposals for representing text collections [2], and to inverted indexes in general.

2 Related Work

We divide the related work into two subsections, the first one covering *rank*, *select* and *access* for binary sequences, and the second covering the case of larger alphabets. We omit the base of logarithms when it is 2. We make heavy use of the definition of zero-order empirical entropy for a sequence S of length n drawn from an alphabet Σ of size σ: $H_0(S) = \sum_{a \in \Sigma} \frac{n_a}{n} \log \frac{n}{n_a}$, where n_a is the number of occurrences of symbol a in S. In the case where $\Sigma = \{0, 1\}$ and $n_1 = m << n$, it is interesting to write $H_0 = m \log \frac{n}{m} + O(m)$.

2.1 Binary Sequences

Many solutions have been proposed for the case of binary sequences. Consider a bitmap $B[1, n]$ with m ones. The first compact solution to this problem is capable of answering the queries in constant time and uses $n + o(n)$ bits [4] (i.e., B itself plus $o(n)$ extra space); the solution is straightforward to implement [10]. This was later improved by Raman, Raman and Rao (RRR) [18] achieving $nH_0(B) + o(n)$ bits while answering the queries in constant time, but the technique is not anymore simple to implement. Several practical alternatives achieving very close results have been proposed by Sadakane and Okanohara [16], tailored to the case of small m: `esp`, `recrank`, `vcode`, `sdarray`, and `darray`. Most of them are very good for *select* queries, yet *rank* queries are slower. The variant `esp` is indeed a practical implementation of RRR structure that saves space by replacing some pointers by estimations based on entropy.

In this work we implement the RRR data structure [18]. It divides the sequence into blocks of length $u = \frac{\log n}{2}$ and every block is represented as a tuple (c_i, o_i). The first component, c_i, represents the *class* of the block, which corresponds to its number of 1s. The second, o_i, represents the *offset* of that block inside a list of all the posible blocks in class c_i. Three tables are defined: E, R and S. Table E stores every possible combination of u bits, sorted by class, and by offset within each class. It also stores all answers for *rank* at every position of each combination. Table R corresponds to the concatenation of all the c_i's, using $\lceil \log(u+1) \rceil$ bits per field. Table S stores the concatenation of the o_i's using $\left\lceil \log \binom{u}{c_i} \right\rceil$ bits per field. This structure also needs two partial sum structures [17], one for R and the other for the length of the o_i's in S, $posS$. For answering *rank* until position i we first compute $sum(R, \lfloor i/u \rfloor) = \sum_{j=0}^{\lfloor i/u \rfloor} R_j$, the number of 1s before the beginning of i's block, and then *rank* inside the block until position i using table E. For this we need to find o_i: using $sum(posS, \lfloor i/u \rfloor)$ we determine the starting position of o_i in S, and with c_i and u we know how many bits we

need to read. For *select* queries, they store the same extra information as Clark [4], but no practical implementation for this extra structure has been shown. *Access* can be answered with two *rank*s, $access(i) = rank(1, i) - rank(1, i-1)$.

2.2 Arbitrary Sequences

Rank, *select* and *access* operations can be extended to arbitrary sequences drawn from an alphabet Σ of size σ. The two most prominent data structures that solve this problem are reviewed next.

Wavelet Trees [11, 8, 15] are perfectly balanced trees that store a bitmap of length n in the root; every position in the bitmap is either 0 or 1 depending on the value of the most significant bit of the symbol in that position in the sequence.[1] A symbol with a 0 goes to the left subtree and a symbol with a 1 goes to the right subtree. This decomposition continues recursively with the next highest bit, and so on. The tree has σ leaves and requires $n\lceil \log \sigma \rceil$ bits, n bits per level. Every bitmap in the tree answers *access*, *rank* and *select* queries.

The *access* query for position i can be answered by following the path described for position i. At the root, if the bitmap at position i has a 0/1, we descend to the left/right child, switching to the bitmap position $rank(0/1, i)$ in the left/right subtree. This continues recursively until reaching the last level, when we finish forming the binary representation of the symbol.

Query *rank* for symbol a until position i can be answered in a similar way as *access*, the difference being that instead of considering the bit at position i in the first level, we consider the most significant bit of a; for the second level we consider the second highest bit, and so on. We update the position for the next subtree with $rank(b, i)$, where b is the bit of a considered at this level. At the leaves, the final bitmap position corresponds to the answer to $rank(a, i)$ in S.

The *select* query does a similar process as *rank*, but upwards. To select the i-th occurrence of character a, we start at the leaf where a is represented and do $select(b, i)$ where, as before, b is the bit of a corresponding to this level. Using the position obtained by the binary *select* query we move to the parent, querying for this new position. At the root, the position is the final result.

The cost of the operations is $O(\log \sigma)$ assuming constant-time *rank*, *select* and *access* over bitmaps. A practical variant to achieve $n(H_0(S) + 1)$ bits of space is to give the wavelet tree the shape of the Huffman tree of S [11, 15].

Golynski et al. [9] proposed a data structure capable of answering *rank*, *select* and *access* in time $O(\log \log \sigma)$ using $n \log \sigma + n\, o(\log \sigma)$ bits of space. The main idea is to reduce the problem over one sequence to n/σ *chunks* of length σ. For each symbol they concatenate, in unary, the number of its occurrences in each *chunk*, and then concatenate those sequences for all the symbols in a bitmap B. Armed with *rank* and *select* structures, B's total length is $2n + o(n)$ bits.

[1] In general wavelet trees are described as dividing alphabet segments into halves. The description we give here, based on the binary decomposition of alphabet symbols, is more convenient for the solutions shown in this paper.

With B it is possible to answer *rank* and *select* queries up to *chunk* granularity. Every *chunk* stores σ text symbols using a bitmap X and a permutation π. X stores the cardinality of every symbol of the alphabet in the *chunk* using the same encoding as B. π stores the permutation obtained by stably sorting the sequence represented by the *chunk*, and uses a data structure that allows computation of π^{-1} in $O(\log\log\sigma)$ time [13]. This adds up to $2n + n\log\sigma + n\,o(\log\sigma)$ bits, and permits completing all queries in constant time for *select*, and $O(\log\log\sigma)$ time for *rank* and *access*. The latter complexity needs a Y-Fast trie within each chunk to search π for the position of interest, among those corresponding to the occurrences of a single symbol within the chunk.

We note that the $n\,o(\log\sigma)$ extra term does not vanish asymptotically with n but with σ. This suggests, as we verify experimentally later, that the structure performs well only on large alphabets.

3 Practical Implementations

3.1 Raman, Raman and Rao's Structure

We fix $u = 15$ so that the c_i's need 4 bits to represent the class $(0-15)$. We store table E using 16-bit integers for the bitstring contents, and for the pointers to the beginning of each class in E. The answers to *rank* are not stored but computed on the fly from the bitstrings, so E uses just 64 KB. Table R is represented by a compact array using 4 bits per field, achieving fast extraction. Table S stores each offset using $\left\lceil \log \binom{u}{c_i} \right\rceil$ bits.

The partial sums are represented by a one-level sampling. For table R we sample the sum every k values, and store these values in a new table $sumR$ using $\lceil \log m \rceil$ bits per field, where m is the number of ones. To obtain the partial sum until position i we compute $sumR[j] + \sum_{p=jk}^{i} c_p$ where $j = \lfloor i/k \rfloor$, and the summation of the c_p's is done sequentially over the R entries. The positions in S are represented the same way: We store the sampled sums in a new table called $posS$ using $\lceil \log(\sum_{i=1}^{n/u} \lceil \log \binom{u}{c_i} \rceil) \rceil$ bits per field. We compute the position for block i as $posS[j] + \sum_{p=jk}^{i} \lceil \log \binom{u}{c_p} \rceil$. We precompute the 16 possible $\lceil \log \binom{u}{c_p} \rceil$ values in order to speed up this last sequential summation.

With this support, we answer *rank* queries by using the same RRR procedure. Yet, $select(1, i)$ queries are implemented in a simpler and more practical way. We use a binary search over $sumR$, finding the rightmost sampled block for which $sumR[k] \leq i$. Then we traverse table R looking for the block in which we expect to find the i-th bit set (i.e., adding up c_p's until we exceed i). Finally we access this block in table E and traverse it bit by bit until finding the i-th 1. $Select(0, i)$ can be implemented analogously.

3.2 Wavelet Trees without Pointers

There exist already Huffman-shaped wavelet tree implementations that achieve close to zero-order entropy space. Yet, those solutions are not efficient

when the alphabet is very large: The overhead of storing the Huffman symbol assignment and the wavelet tree pointers, $O(\sigma \log n)$, ruins the compression if σ is large. In this section we present an alternative implementation that achieves zero-order entropy with a very mild dependence on σ (i.e. $O(\log \sigma \log n)$ bits of space), thus extending the existing results to the case of very large alphabets. We use two bitmaps: DA and Occ. $DA[1, \sigma]$ stores which symbols appear in the sequence, $DA[i] = 1$ if symbol i appears in S. This allows us to remap the sequence in order to get a contigous alphabet; using *rank* and *select* over DA we can map in both directions (we only use it if the alphabet is not contiguous). $Occ[1, n]$ records the number of occurrences of symbols $i \leq k$ by placing a one at $\sum_{i=1}^{k} n_i$. For example, the sequence 113213323 would generate $Occ = 001010001$.

Our implementation of the wavelet tree stores $\lceil \log \sigma \rceil$ bitmaps of length n. The tree is mapped to these bitmaps levelwise: the first bitmap corresponds to the root, the next one corresponds to the concatenation of left and right children of the root, and so on. In this set of bitmaps we must be able to calculate the interval $[s, e]$ corresponding to the bitmap of a node, and to obtain the new interval $[s', e']$ upon a child or parent operation. Assume the current node is at level l ($l = 1$ at the leaves) on a tree of h levels. Further, assume that a is the symbol related to the query, that $\Sigma = \{0, \ldots, \sigma-1\}$, and that $select_{Occ}(1, 0) = 0$.

We compute the left child as $s' = s$ and $e' = e - rank(1, e) + rank(1, s-1)$, and the right child as $s' = e + 1 - rank(1, e) + rank(1, s-1)$ and $e' = e$. Let us explain the left child formula. In the next level, the current bitmap segment is partitioned into a left child and right child parts. The left child starts at the same position of the current segment in this level, so $s' = s$. To know the end of its part, we must add the number of 0s in the current segment, $e' = s + rank(0, e) - rank(0, s-1) - 1 = s + (e - rank(1, e)) - ((s-1) - rank(1, s-1)) - 1$.

The formula to compute the parent is $s' = select_{Occ}(1, \lfloor a/2^l \rfloor \cdot 2^l) + 1$ and $e' = select_{Occ}(1, (\lfloor a/2^l \rfloor + 1) \cdot 2^l)$. The idea is to consider the binary representation of a, as this is the way our wavelet tree is structured. A node at level l should contain all the combinations of the l lowest bits of a. For example, if $l = 1$ and $a = 5 = (101)_2$, its parent is the node at level $l = 2$ comprising the symbols $4 = (100)_2$ to $5 = (101)_2$. The parent of this node, at level $l = 3$, comprises the symbols $4 = (100)_2$ to $7 = (111)_2$. We blur the last l bits of a and use $select_{Occ}$ to find the right segments at any level corresponding to the symbol intervals.

To achieve compression we represent the bitmaps of each level (as well as Occ) using RRR, whose sampling yields a time/space trade-off for the structure.

3.3 Golynski's Structure

We implement Golynski et al.'s proposal rather faithfully, except that we replace the Y-Fast trie by a binary search over the positions for the *rank* query. In practice, this yields a gain in space and time except for large σ values and biased symbol distribution within the *chunk* (remind that we must search within the range of occurrences of a symbol of Σ in a *chunk* of size σ, i.e. the range is $O(1)$ size on average). Hence the time for *rank* is $O(\log \sigma)$ worst case, and $O(1)$ on average. The version used for permutations [13] requires $(1 + \epsilon)n\lceil \log n \rceil$ bits for

n elements and computes π^{-1} in $O(1/\epsilon)$ worst-case time (code by D. Arroyuelo). This gives us a space/time tradeoff parameter for this data structure.

In the case when $n \approx \sigma$ we also experiment with using only one *chunk* to represent the structure. This speeds up all the operations since we do not need to compute which *chunk* should we query, and all the operations over B become unnecessary, as well as storing B itself.

4 Experimental Results

We first test the data structures for binary sequences, on random data and on the BWT of real texts, showing that RRR is an attractive option. Second, we compare the data structures for general sequences on various types of large-alphabet texts, obtaining several interesting results. Finally, we apply our machinery to obtain the best results so far on compressed text indexing.

The machine is a Pentium IV 3.0 GHz with 4GB of RAM using Gentoo GNU/Linux with kernel 2.6.13 and `g++` with `-O9` and `-DNDEBUG` options.

4.1 Binary Sequences

We generated three random uniformly and independently distributed bitmaps of length $n = 10^8$, with densities (fraction of 1s) of $5\%, 10\%$ and 20%. Fig. 1 compares our RRR implementation against the best practical ones in previous work [16], considering operations *rank* and *select* (*access* can be implemented as the difference of two *rank*'s, and in some cases it can be done slightly better, yet only some of the structures in [16] support it). As control data we include a fast uncompressed implementation [10], which is insensitive to the bitmap density.

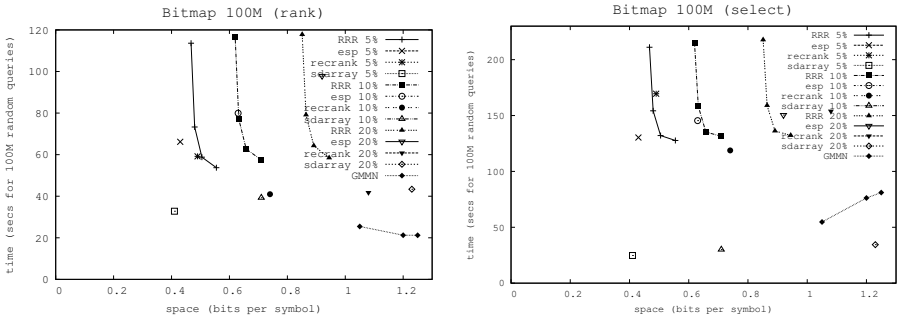

Fig. 1. Space in bits per symbol and time in seconds for answering 10^8 random queries over a bitmap of 10^8 bits.

Our RRR implementation is far from competitive for very low densities (5%), where it is totally dominated by `sdarray`, for example. For 10% density it is already competitive with `esp`, its equivalent implementation [16], while offering

Variant	Size	*Rank* time
`sdarray`	2.05	> uncompressed
`recrank`	1.25	> uncompressed
`esp`	0.50	0.594
RRR (ours)	0.48	0.494
uncompressed	1.05	0.254

Table 1. Space (as a fraction of bitmap size) and *rank* time (in μsec/query) achieved by various data structures on the wavelet tree bitmaps of a BWT-transformed text.

more space/time tradeoffs and achieving the best space. For 20% density, RRR is unparalleled in space usage, and alternative implementations need significantly more space to beat its time performance. We remark that RRR implements $select(0, i)$ in the same time of $select(1, i)$ with no extra space, where competing structures would have to double the space they devote to *select*.

A property of RRR that is not apparent over uniformly distributed bitmaps, but it becomes very relevant for implementing the wavelet tree of a BWT-transformed text, is its ability to exploit local regularities in the bit sequence. To test this, we extracted the 50MB `English` text from *Pizza&Chili* (`http://pizzachili.dcc.uchile.cl`), computed the balanced wavelet tree of its BWT, and concatenated all the bitmaps of the wavelet tree levelwise. Table 1 shows the compression achieved by different methods. Global methods like `sdarray` and `recrank` fail, taking more space than an uncompressed implementation. RRR stands out as the clear choice for this problem, followed by `esp` (which is based on the same principle). The bitmap density is around 40%, yet RRR achieves space similar to 5% uniformly distributed density.

4.2 General Sequences

We compared our implementations of Golynski et al.'s and different variants of wavelet trees. We consider three alphabet sizes. The smaller one is byte-size: We consider our plain text sequences `English` and `DNA`, seeing them as character sequences. Next, we consider a large alphabet, yet not large enough to compete with the sequence size: We take the 200MB `English` text from *Pizza&Chili* and regard this as a sequence of *words* The result is a sequence of $46,582,195$ words over an alphabet of size $270,096$. Providing *access* and *select* over this sequence mimics a word-addressing inverted index functionality [2]. Finally, we consider a case where the alphabet is nearly as large as the text. The sequence corresponds to graph `Indochina` after applying *Re-Pair* compression on its adjacency list [5]. The result is a sequence of length $15,568,253$ over an alphabet of size $13,502,874$. The result of *Re-Pair* can still be compressed with a zero-order compressor, but the size of the alphabet challenges all of the traditional methods. Our techniques can achieve such compression and in addition provide *rank*/*select* functionality, which permits implementing backward traversal on the graph for free.

We consider full and 1-chunk variants of Golynski. On wavelet trees, variant *DA* maps the alphabet to a contiguous range, *RRR* compresses the bitmaps with RRR, *Ptrs* uses the standard version with pointers, and *Huff* gives Huffman tree shape to the pointer-based wavelet tree. We show several combinations of these.

Fig. 2 shows the results for byte alphabets. Golynski's structure is not competitive here. This shows that their $o(\log \sigma)$ term is not yet negligible for $\sigma = 256$. On wavelet trees, the Ptrs+Huff variant excells in space and in time. Adding RRR reduces the space very slightly in some cases; in others keeping balanced shape gives better time in exchange for much more space. We also included Naive, an implementation for byte sequences that stores the plain sequence plus regularly sampled *rank* values for all symbols [2]. The results show that we could improve their wavelet trees on words by replacing their Naive method by ours.

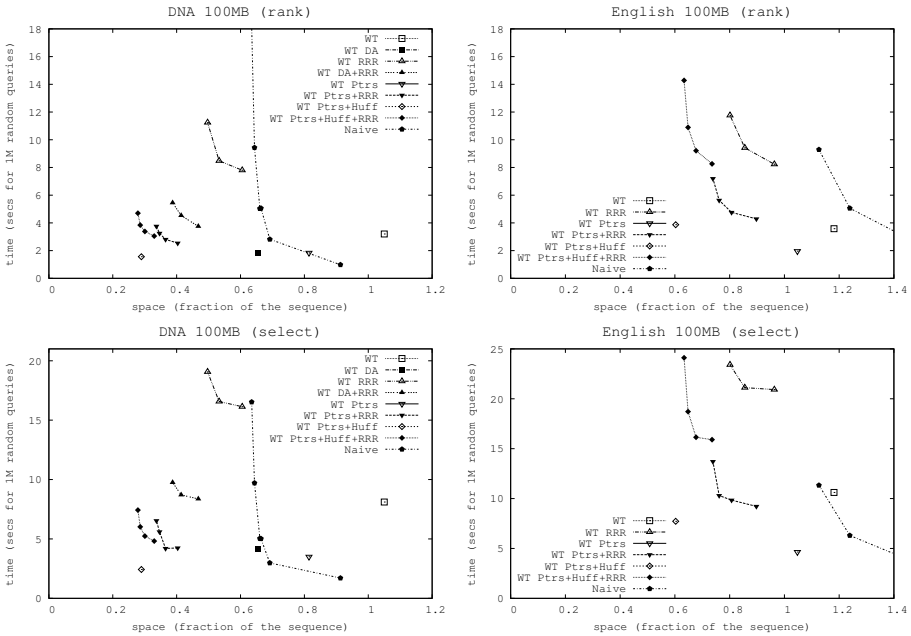

Fig. 2. Results for byte alphabets. Space is measured as a fraction of the sequence size (assuming one byte per symbol).

Fig. 3 shows experiments on larger alphabets. Here Golynski et al.'s structure becomes relevant. On the sequence of words it adds to the previous scenario a third space/time tradeoff point, offering much faster operations (especially *select*) in exchange for significantly more space. Yet, this extra space is acceptable for the sequence of word identifiers, as overall it requires 70% *of the original text*. As such it competes with a word-addressing inverted index, following a recent trend of replacing inverted indexes by a succinct data structure for representing

sequences of word identifiers [2], which can retrieve the text using *access* but also find the consecutive positions of a word using *select*.

Again, adding RRR reduces the space of Ptrs+Huff, yet this time the reduction is more interesting, and might have to do with some locality in the usage of words across a text collection.

For the case of graphs, where the alphabet size is close to the text length, the option of using just one chunk in Golynski et al.'s structure becomes extremely relevant. It does not help to further compress the Re-Pair output, but for 20% extra space it provides very efficient backward graph traversal. On the other hand, the wavelet trees with DA+RRR offer further compression up to 70% of the Re-Pair output size, which is remarkable for a technique that already achieved excellent compression results [5]. The price is much higher access time. An interesting point here is that the versions with pointers are not applicable here, as the alphabet overhead drives their space over 3 times the sequence size. Hence exploring versions that do not use pointers pays off.

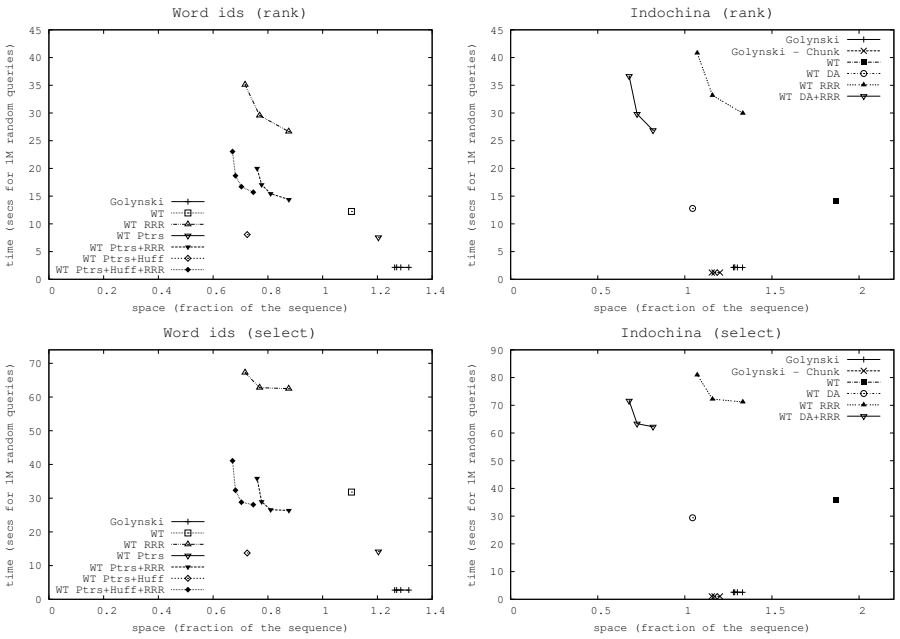

Fig. 3. Results for word identifiers (left) and a graph compressed with Re-Pair (right). Space is measured as a fraction of the sequence (using $\lceil \log \sigma \rceil$ bits per symbol).

4.3 Compressed Full-Text Self-Indexes

It was recently proved [12] that the wavelet tree of the Burrows-Wheeler transform (BWT) of a text (the key ingredient of the successful FM-index family

of text self-indexes [15]), can achieve high-entropy space without any further sophistication, provided the bitmaps of the wavelet tree are represented using RRR structure [18]. Hence a simple and efficient self-index emerges, at least in theory. In Section 4.1 we showed that RRR indeed takes unique advantage from the varying densities along the bitmap typical of the BWT transform. We can now show that this proposal [12] has much practical value.

Fig. 4 compares the best suffix-array based indexes from *Pizza&Chili*: SSA, AFFM-Index, RLFM-Index (the three based on the BWT) and CSA. We combine SSA with our most promising versions for this setup, WT Ptrs+RRR and WT Ptrs+Huff+RRR. All the spaces are optimized for the *count* query, which is the key one in these self-indexes

We built the index over the 100 MB texts `English`, `DNA`, `Proteins`, and `Sources` provided in *Pizza&Chili*. We chose 10^5 random patterns of length 20 from the same texts and ran the *count* query on each.

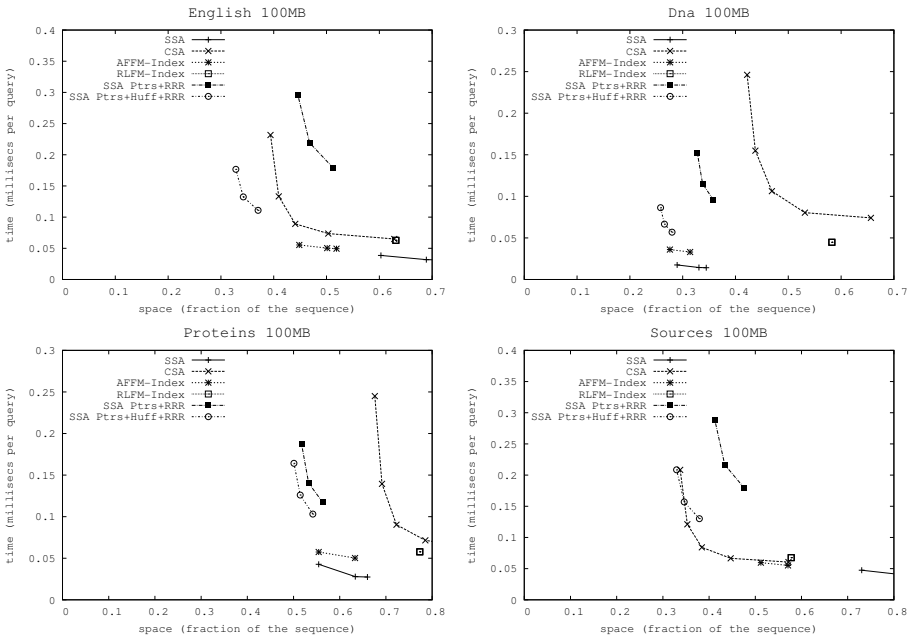

Fig. 4. Times for counting, averaged over 10^5 repetitions, for patterns of length 20.

As can be seen, our new implementation is extremely space-efficient, achieving a space performance *never seen before* in compressed full-text indexing, and in some cases without a time penalty.

The reason why combining RRR with Huffman shape is better than RRR alone, when RRR by itself should in principle exploit all of the compressibility captured by Huffman, is in the c component of the (c, o) pairs of RRR. These

pose a fixed overhead per symbol which is not captured by the entropy. Indeed, we measured the length of the o components (table S) in both cases (for English) and the difference was 0.02%. The Huffman shape helps reduce the total number of symbols to be indexed, and hence it reduces the overhead due to the c components.

Several lines of work are open to future work, in particular implementing an efficient version of RRR combined with run-length encoding, which should give even better spaces. Another line is to pursue on the idea of an inverted-index-like capability by encoding the sequence of words of a natural language text, or a word-based self-index.

References

1. J. Barbay, M. He, I. Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *18th SODA*, pages 680–689, 2007.
2. N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *SIGIR*, 2008. To appear.
3. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Tech.Rep. 124, DEC, 1994.
4. D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
5. F. Claude and G. Navarro. A fast and compact Web graph representation. In *SPIRE*, pages 105–116, 2007.
6. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice! Manuscript. `http://pizzachili.dcc.uchile.cl`, 2007.
7. P. Ferragina and G. Manzini. Indexing compressed texts. *J.ACM*, 52(4):552–581, 2005.
8. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM TALG*, 3(2):article 20, 2007.
9. A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA*, pages 368–373, 2006.
10. R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Posters WEA*, pages 27–38, 2005.
11. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.
12. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *SPIRE*, pages 214–226, 2007.
13. I. Munro, R. Raman, V. Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *ICALP*, pages 345–356, 2003.
14. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp.Surv.*, 39(1):article 2, 2007.
15. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp.Surv.*, 39(1):article 2, 2007.
16. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*, 2007.
17. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *WADS*, pages 426–437, 2001.
18. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.