Compressed q-gram Indexing for Highly Repetitive Biological Sequences

Francisco Claude
University of Waterloo
fclaude@cs.uwaterloo.ca

Antonio Fariña
University of A Coruña
fari@udc.es

Miguel A. Martínez-Prieto
University of Valladolid
migumar2@infor.uva.es

Gonzalo Navarro
University of Chile
gnavarro@dcc.uchile.cl

***Abstract*—The study of compressed storage schemes for highly repetitive sequence collections has been recently boosted by the availability of cheaper sequencing technologies and the flood of data they promise to generate. Such a storage scheme may range from the simple goal of retrieving whole individual sequences to the more advanced one of providing fast searches in the collection. In this paper we study alternatives to implement a particularly popular index, namely, the one able of finding all the positions in the collection of substrings of fixed length (q-grams). We introduce two novel techniques and show they constitute practical alternatives to handle this scenario. They excell particularly in two cases: when q is small (up to 6), and when the collection is extremely repetitive (less than 0.01% mutations).**

I. Introduction and Related Work

The sequencing of the whole Human Genome was a celebrated breakthrough. The goal was to obtain a *consensus* sequence accounting for the common parts of the genomes of all humans. Less than one decade later, DNA sequencing technologies have become so fast and cost-effective that sequencing *individual* genomes will soon become a feasible and common task [1], [2], [3], and huge collections of DNA data are at the next corner.

The computational challenges posed by handling collections of thousands of genomes are formidable, even on disk storage. In general, DNA and proteins are not compressible (beyond exploiting their small alphabet size) using classical methods [4]. Compression methods tailored for DNA, such as GenCompress [5], Biocompress [6], Fact [7], and GSCompress [8], have only moderate success. Fortunately, massive collections of *related* DNA sequences have a fundamental property that makes them more tractable: they are *highly repetitive*, for obvious biological reasons (e.g., two human genomes are more than 99.9% similar).

Compressing and indexing those highly repetitive sequence collections, however, poses many challenges, as most of the current compression technology is not well prepared to deal with them. In practice, methods like those in `ppmdi, gzip` and `bzip2` will not take advantage of the repetitiveness because they search for repetitions in a bounded window of the text. Another less known member of the family, `p7zip`, uses a larger buffer and is very successful, yet unable of decompressing individual sequences. Grammar-based compressors have a good chance to succeed, as long as the compression does not proceed by limited chunks. This is well illustrated by software `comrad`[1], which achieves good compression ratios and efficient extraction of individual sequences [9].

On the other hand, recent research [10] has shown that the current compressed-indexing technology is also not properly prepared to handle this high repetitiveness. However, they show that the BWT of highly repetitive collections contains long runs of identical symbols, which they successfully exploit to build extremely compact indices able of not only retrieving the individual sequences, but also of extracting arbitrary subsequences of them, and of counting and locating the number of occurrences of arbitrary strings in the collection.

A weak point of these BWT-based indices is that, although counting the number of occurrences of pattern substrings is fast, retrieving their positions in the collection is relatively slow. This is particularly noticeable when searching for short sequences, which have many occurrences. In turn, this is particularly unfortunate because the search for short sequences is an extremely popular task in Computational Biology. For example, all the techniques that build on analyzing K-mers require that kind of search. Another example is the approximate search of sequences for conserved regions, that is, subsequences that are sufficiently similar to a given pattern sequence. Such searches usually start by identifying areas where short pattern strings appear, then they extract the sequence around those occurrences, and finally run a sequential approximate pattern matching algorithm on the spotted areas. This is how the popular BLAST tool[2] proceeds, for example.

In this paper we focus on providing compressed storage schemes for highly repetitive sequence collections, while providing efficient indexed search for substrings of a fixed length q. We test two widely differerent mechanisms and study their effectiveness.

The first method is inspired in a result [11] showing that a compressed q-gram index is more effective than the BWT-based self-indexes when searching for short patterns. We compress the sequences using a grammar-based compressor (Re-Pair [12]) and compress the lists of occurrences of each q-gram using Lempel-Ziv compression (more precisely, LZ77 [13]) after differentially encoding the positions. The rationale is that highly repetitive sequences will induce also repetitiveness in the sequences of relative positions where the q-grams occur.

[1] `http://www.cs.mu.oz.au/~kuruppu/comrad`
[2] `http://blast.ncbi.nlm.nih.gov/Blast.cgi`

The second method is inspired in a recent technique [14] to turn a grammar-based compressor into an index able of not only retrieving any part of the sequence, but also locating patterns in the collection. We explain how to adapt the method to Re-Pair compression [12], and use it to search for the desired q-grams. The index is able of searching for pattern strings of any length, and particularly effective on short ones.

Our experimental results show that both techniques are competitive with the state of the art in this particular scenario, where they offer practical alternatives to compress and index highly repetitive sequence collections. The q-gram index is unrivalled for $q \leq 6$, whereas the grammar-based index is unbeatable for collections with less than 0.01% of differences.

From now on, we use T to denote the text collection, which we represent as the concatenation of all the texts in the collection (using proper separators). We call N the total length, in symbols, of T. In general we will explain search operations for q-grams, yet we will also refer to them as a pattern P. We call occ the number of times the q-gram or P occurs in T.

II. A Lempel-Ziv Compressed q-Gram Index

A classical q-gram index is formed by a *vocabulary* comprising all the different substrings of length q in T, and for each of those q-grams, a *posting list* recording the positions in T where they appear, in increasing order. To reduce space, *block addressing* is used, that is, the text is divided into blocks of b characters and the posting lists mention only the distinct blocks where their q-gram appears.

A search for all the positions where a given q-gram appears proceeds as follows. First, the query is looked up in the vocabulary and its posting list is retrieved. Second, each block in the list is scanned using a sequential string matching algorithm, and the exact occurrences of the q-gram are reported. To ensure that q-grams overlapping blocks are correctly found, it is necessary to replicate the first $q-1$ symbols of each block at the end of the previous block.

We choose binary search for the vocabulary lookup, and Horspool algorithm [15] for the sequential search of blocks, as it is usually a good choice for the typical range of pattern lengths q to search (4 to 12) [16].

In order to further reduce the space, we compress both the q-gram index and the text with techniques tailored for highly repetitive sequences. As both decompress in linear time, we can already give the expected search time for a q-gram. The binary search time can be upper bounded by $O(q \log N)$, the decompression time for the posting list by $O(occ)$, and decompression and scanning of a block by $O(b)$. The average number of blocks, out of $n = \lceil N/b \rceil$, where any of the occ occurrences appear, is $(N/b)(1-(1-b/N)^{occ})$. Multiplied by the processing cost $O(b)$, we have the average time $O(N(1-(1-b/N)^{occ}))$, which in the worst case is bounded by $O(\min(b \cdot occ, N))$.

A. Compressing the Index

A customary way of compressing inverted indexes [17], which also applies to q-gram indexes, starts by differentially encoding the posting lists. Let $L_t = p_1 p_2 \ldots p_\ell$ the list of blocks where q-gram t appears. Its differentially encoded version is $D_t = d_1 d_2 \ldots d_\ell$, where $d_1 = p_1$ and $d_{i+1} = p_{i+1} - p_i$ for $1 \leq i < \ell$. Because $1 \leq p_i \leq n$, the numbers d_i are smaller when ℓ is larger, that is, when the lists are longer. Hence a variable-length encoding technique that assigns shorter codewords to smaller numbers achieves compression especially on the longer lists.

In Natural Language, this technique takes advantage of the fact that some words are much more frequent than others [18], but not of the fact that a collection might be repetitive. We now modify the scheme to account for that scenario, which is the one of interest for us. Consider a sequence SXS, where string S appears twice. Let $s_1, s_2, \ldots, s_a$ be the occurrences of a given q-gram t in S, and $x_1, x_2, \ldots, x_b$ its occurrences in X. Then, a repetition of the sequence of numbers $(s_2 - s_1), (s_3 - s_2), \ldots, (s_a - s_{a-1})$ appears in D_t thanks to the differential encoding. Because we wish to retrieve the whole list L_t from the beginning, any compression algorithm able of taking advantage of long repetitions can be used. We choose LZ77 [13], as it is fast to decompress and can detect any repeated sequence seen in the past in order to compress the upcoming text. In particular, LZ77 would spend just $O(\log(a+b))$ bits to encode the second occurrence of $(s_2 - s_1), (s_3 - s_2), \ldots, (s_a - s_{a-1})$ in D_t.

Several variable-length encoders emit codes of arbitrary numbers of bits. We opt instead for Vbyte encoding [19], which uses a variable number of whole bytes. Its advantages are that it decompresses fast and that the repetitions in D_t will show up as repeated sequences of bytes, which can be spotted by available LZ77 implementations.

Therefore, we transform each list L_t into D_t by writing the differences, then encode the numbers d_i using Vbyte encoding, and finally use LZ77 compression on the resulting list. Each posting list is compressed separately, so that it can be decompressed independently. As our LZ77 compressor we use `lzma` from the `p7zip` distribution (`www.p7zip.org`). If the resulting list compressed with `lzma` is longer than the uncompressed version, we represent the list using only Vbyte encoding. A bitmap marks which lists are compressed.

B. Compressing the Text

The text itself is also compressed, in such a way that repetitions are exploited and individual blocks can be efficiently decompressed. LZ77 compression is not suitable because we should compress each block separately in order to be able of independently decompressing it.

We opt for a grammar-based compression algorithm, as in previous work [9]. These are strong enough to detect long-range repetitions in the text, and are also able of fast local decompression. In particular, we choose Re-Pair [12], which compresses and decompresses in linear time and offers good compression ratios on highly repetitive sequences.

Re-Pair operates as follows. (1) It finds the most common pair ab of characters in the sequence; (2) It creates a rule $A \rightarrow ab$, where A is a new (nonterminal) symbol; (3) It replaces all

the occurrences of ab in the sequence by A; (4) It iterates from step (1) until no pair appears twice. Note that newly created symbols A can be further replaced by other rules. The outcome of the algorithm is a set of rules $\mathcal{D}$ and the final compressed stream $\mathcal{C}$ where no repeated pairs appear. Note that $\mathcal{C}$ is a sequence of integers, not characters, as many nonterminals can be created. Compression, somewhat surprisingly, can be executed in linear time. Decompression is linear-time and very efficient in practice if we want to decompress a subsequence of $\mathcal{C}$: we recursively unroll each nonterminal of $\mathcal{C}$ until reaching the terminals (i.e., the original characters).

In order to detect long-range repetitions and at the same time allow for isolated block decompression in an efficient way, we insert the special integer symbol $-i$ after the i-th block. As this symbol is unique, it cannot participate in any pair that appears twice, and thus Re-Pair will not create nonterminals that cross a block border. Those symbols will remain in sequence $\mathcal{C}$ and will mark the limits of blocks. After executing Re-Pair, symbols $-i$ are removed from the sequence, and we set up an array of pointers to the beginning of each block in $\mathcal{C}$. These pointers are used to identify the area of $\mathcal{C}$ that must be decompressed and then searched for the pattern.

III. A Grammar-Compressed Self-Index

Instead of combining a compressed index with a text compression mechanism, we can opt for a *self-index*, which is a compressed index that is able of reproducing any text passage, and thus it replaces the text as well [20]. However, traditional self-indexes, as explained in the Introduction, are not well suited to compressing highly repetitive sequences.

Recent theoretical work [14] proposes a self-index technique for straight-line programs (SLPs), a restricted kind of grammar. This type of compression is very promising for highly repetitive sequences, and such a self-index could be competitive to retrieve q-grams from the collection.

Finding the smallest grammar that generates a given sequence is an NP-hard problem, so one must resort to good heuristics. We choose Re-Pair as a concrete instance of grammar-based compressor to apply, as it compresses in linear time and yields good results. Re-Pair does not generate exactly an SLP, so in the sequel we explain the self-indexing technique used for Re-Pair compression, including the practical decisions made during the implementation.

Recall that $\mathcal{C}$ is the sequence resulting from applying Re-Pair to T, and $\mathcal{D}$ is the set of n rules created during the process. We can regard every terminal symbol as a rule that generates itself. All the rules generated by Re-Pair are of the form $X_i \rightarrow X_l X_r$, and no loops are allowed (which is basically the property required to be an SLP). We call $\mathcal{F}(X)$ the expansion of X into terminals, and $\mathcal{F}^{rev}(X)$ the corresponding reversed string (read backwards, not complemented).

A. Representing the Set of Rules

For representing the rules generated by Re-Pair, we use a *labeled binary relation* data structure (LBR) [14]. The LBR represents a binary relation $\mathcal{R}$ between sets $A = [1, n]$ and $B = [1, n]$, where the labels are from the set $\mathcal{L} = [1, n]$[3]. We refer to A as the rows of the binary relation and B as the columns. The operations we require in this work are the following, all of which are supported in $O(\log n)$ per datum retrieved: $\mathcal{L}(a \in A, b \in B)$ returns the label associated to the pair $(a, b) \in \mathcal{R}$ or $\perp$ if a is not related to b; $R(a_1, a_2, b_1, b_2)$ retirves the set of elements $(a, b) \in \mathcal{R}$ such that $a_1 \leq a \leq a_2,\ b_1 \leq b \leq b_2$; $\mathcal{L}(s)$ computes the set of pairs (a, b) related through label s.

The rules $X \rightarrow X_\ell X_r$ are seen as "X_ℓ relates to X_r through label X". The rows are sorted by lexicographic order of $\mathcal{F}^{rev}$ and the columns by lexicographic order of $\mathcal{F}$. We store the permutation π that maps from rows to columns, and also support mapping from columns to rows in $O(\log n)$ time [21]. The operations allow direct and reverse access to the rules in $O(\log n)$ time per element retrieved: $\mathcal{L}(l, r)$ returns j such that $X_j \rightarrow X_l X_r$ if any; $R(l_1, l_2, r_1, r_2)$ returns the set of right-hands $X_l X_r$ where $l_1 \leq l \leq l_2$ and $r_1 \leq r \leq r_2$; $\mathcal{L}(s)$ retrieves the pair (l, r) such that $X_s \rightarrow X_l X_r$.

1) Searching for Primary Occurrences.: We define the primary occurrences of a pattern $P = p_1 p_2 \ldots p_q$ as those symbols $X_i \rightarrow X_l X_r$ that contain P and its occurrence spans from X_l to X_r. This means $\mathcal{F}(X_l) = \ldots p_1 p_2 \ldots p_j$ and $\mathcal{F}(X_r) = p_{j+1} \ldots p_q \ldots$ Once we find the primary occurrences, all the occurrences of the pattern can be retrieved by obtaining symbols that contain the primary occurrences. In other words, we track each primary occurrence inside X_i upwards through the parse forest[4] defined by the rules, by recursively using $R(i, i, 1, n)$ and $R(1, n, \pi(i), \pi(i))$ and then $\mathcal{L}(a, b)$ on the resulting pairs (a, b). We simultaneously track the position of the occurrence as we find it inside other nonterminals. This can be done in time $O(h \log n)$ per occurrence, at worst, where h is the height of the parse forest.

The search for the primary occurrences of a pattern P proceeds as follows. For every partition $P = P_l P_r$, we search P_l^{rev} in the rows and P_r in the columns using binary search (on $\mathcal{F}^{rev}$ and $\mathcal{F}$, respectively; these strings are extracted on the fly for the comparisons). That determines the range of phrases ending with P_l and the ones starting with P_r. Using the general R query we retrieve the elements that combine those phrases, and thus contain the pattern as a primary occurrence. This takes $O((q + h) \log^2 n)$ time per partition, and has to be repeated $q-1$ times, once for each possible partition of P. The space required by the structure is $3n \log n + n \log N$ bits plus lower-order terms: $2n \log n$ for the binary relation, $n \log n$ for π, and $n \log N$ for the length of the phrases $\mathcal{F}$.

B. Locating the Occurrences in $\mathcal{C}$

Once we find all the occurrences inside every nonterminal, we have to track were each such symbol appears in $\mathcal{C}$, in order to obtain the actual occurrences in T.

[3]This is a particular case of the original structure that serves the purposes of this paper.

[4]Recall that the rules generated by Re-Pair do not constitute a complete grammar that generates T, but we have to expand each symbol of $\mathcal{C}$.

In order to locate the occurrences of each relevant nonterminal we use *select* queries on $\mathcal{C}$: $select_{\mathcal{C}}(X, i)$ retrieves the i-th occurrence of X in $\mathcal{C}$. During our upward traversal carried out to propagate each primary occurrence, we call $select_{\mathcal{C}}(X, i)$ for $i = 1, 2 \ldots$ until we retrieve all the occurrences of the current nonterminal X, and then we continue with its parents. By representing $\mathcal{C}$, as a wavelet tree without pointers, operation $select$ takes time $O(\log c \log n)$, where $c = |\mathcal{C}|$.

The only kind of occurrences remaining are those that appear when expanding more than one consecutive symbol in $\mathcal{C}$. For solving this, we create another binary relation, relating the n rules with the c positions in $\mathcal{C}$. If $\mathcal{C} = s_1 s_2 \ldots s_c$, then we relate each s_i (sorted in lexicographic order of $\mathcal{F}^{rev}(s_i)$) with suffix $s_{i+1} \ldots$ (sorted in lexicographic order of $\mathcal{F}(s_{i+1} \ldots)$), with label i. Any occurrence of a pattern P starting at s_i and continuing at $s_{i+1} \ldots$ can be obtained with a mechanism similar as before: first binary search for the range of rules whose string finishes with $p_1 \ldots p_j$, then binary search for the range of suffixes of $\mathcal{C}$ which string start with $p_{j+1} \ldots p_q$, and then retrieve all the labels i from the intersection of both ranges in the binary relation. Those are the positions in $\mathcal{C}$ where P occurs, with a displacement of j. This is repeated for each j in $[1, q-1]$.

A simple sampling records the original position of every symbol $s_{i\ell}$, for some sampling step ℓ. This permits converting from positions in $\mathcal{C}$ to positions in the original sequence.

C. The Resulting Index

The resulting self-index requires $c(\log n + \log c + (\log N)/s) + n(3 \log n + \log N)$ bits plus sublinear terms. It finds the occ occurrences of any pattern (q-gram in this case) in time $O(q(q+h) \log(c+n) \log n + occ \log n(h + \log c))$. We ensure $h = O(\log N)$ by running a *balanced* Re-Pair, where compression proceeds in rounds, such that symbols generated in a round can only be used in subsequent rounds [22]. A simplified running time upper bound is thus $O((q(q+\log N)+ occ) \log N \log n)$.

Note that such a self-index is not built for any particular q, and thus it is able of searching for any string. As we see in the experimental results, however, it performs best with small q values.

IV. Experimental Results

We compared the space and time performance of our compressed indexing proposals against the state of the art, over several real-life and synthetic repetitive biological sequences.

Our machine is an Intel Core2Duo E6420@2.13Ghz, with 4GB of DDR2-800 RAM. It runs `GNU/Linux`, 64-bit `Ubuntu 8.04` with kernel 2.6.24-24-generic. We compiled with `gcc` 4.2.4 and the option `-O9`.

The experimental setup is divided into three parts. First, we compare the best available compressors for repetitive DNA sequences. Second, we experiment on several alternative designs for the q-gram index, to justify our choice. Third, we compare both compressed indexes and RLCSA [10], which is the state of the art. We use four real-life collections for the experiments. We removed the `'N'` symbols in order to compare against compressors that assume that the input sequence does not contain that symbol.

The datasets we used are: `hemo`[5] with 7,282,339 bases; `influ`[6] composed of 78,041 sequences of H. Influenzae containing 112,640,397 bases; `para`[7] composed of 27 sequences of S. Paradoxus containing 412,279,603 bases; and `cere`[7] that contains 37 sequences of S. Cerevisiae containing 428,118,842 bases.

In addition, we use three synthetic 100MB collections, formed by taking the first 1,048,576 symbols of the DNA file in *PizzaChili* (`http://pizzachili.dcc.uchile.cl`), and replicating it 100 times. These are sequences over {`A,C,G,T,N`}. Collections `s0001`, `s001`, and `s01`, are obtained by mutating a fraction 0.01%, 0.1%, and 1%, respectively, of the symbols. A mutation chooses a random position and changes it to the value of another random position of the initial sequence.

We express the compression ratio as the percentage of the compressed over uncompressed file size, assuming the original file uses one byte per base. This means that 25% compression can be achieved over {`A,C,G,T`} files by simply using 2 bits per symbol. As seen from the real-life examples given, these four symbols are usually predominant, so it is not hard to get very close to 25% on general DNA sequences as well.

A. Plain Compressors

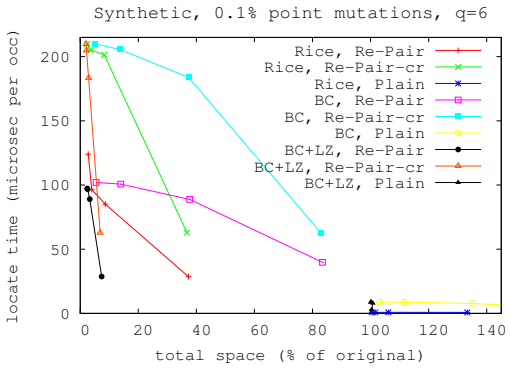
We test the compression ratio achieved by plain DNA compressors, using the 4-symbol reduced sequences, as most of these compressors pose this restriction. As explained in the Introduction, most DNA compressors are only mildly successful in improving compression further than the 2-bits-per-symbol barrier. There are, however, some that perform well on highly repetitive collections: `Comrad` [9], `XM` [23] and `dna2` [24]

Another renowned compressor, `DNACompress`, has not ben included because it needs proprietary software `PatternHunter`. Others, such as `BioCompress` and `GenComp`, did not run on our long sequences. We have considered also general-purpose compressors that are well suited to highly repetitive sequences, such as `p7zip` (`http://www.p7zip.org`), a LZ77-based compressor; and our own implementation of `Re-Pair`, and an implementation using a more compact representation for the dictionary, `Re-Pair-cr` [25]. We believe that our coverage of compressors is sufficient to give a good perspective of the state of the art.

The table inside Figure 3 gives the results. As expected, `dna2` is the fastest in this respect. On the other hand, `XM` achieves the best compression ratios, followed by `p7zip` and `dna2`. `Comrad` and `Re-Pair` compress less, arguably in exchange of their random-access ability (`Re-Pair` performs better on the more compressible sequences). We note that

[5] `http://srs.ebi.ac.uk/srsbin/cgi-bin/wgetz`
[6] `ftp://ftp.ncbi.nih.gov/genomes/INFLUENZA/`
[7] `http://www.sanger.ac.uk/Teams/Team71/durbin/sgrp`

Fig. 1. Space-time tradeoffs achieved by different implementations of the q-gram index.



`Re-Pair` is the compressor we use for our compressed q-gram index, so it gives the base space consumption on top of which we add the compressed inverted lists. In the last column we include `SLP`, the size of our Re-Pair-based self-index. Of course it takes much more space due to its ability of performing indexed searches.

The table confirms that the collections are highly compressible because they are highly repetitive. The least compressible is `hemo`, yet it is still well below the 25% barrier.

B. Study of the q-gram Index

Compared to previous work on compressed inverted indexes [17] and compressed q-gram indexes [11], our compressed q-gram index has two novelties: (1) It compresses the text with block-aligned Re-Pair, so that the compression uses global repetitions but can decompress individual blocks (whereas in natural language one can use natural-language-oriented compression [17], and on general sequences previous work simply does not compress the sequence [11]). (2) It compresses the differential lists with LZ77, which exploits high repetitiveness in the sequences (whereas previous work uses only variable-length coding of the differences [17], [11]).

We consider the 6 combinations of the alternatives of compressing or not compressing the sequences; and of using Rice codes (one of the best variable-length encoders [17]), VByte codes [19] (BC, which compress less but decompresses faster), and VByte codes followed by `lzma` compressor. We note `lzma` spots only byte-aligned repetitions, so it cannot be successfully combined with Rice codes.

Figure 1 shows the results on the synthetic sequence `s001`. We show the case of $q = 6$. The tradeoffs are achieved by using blocks of 1KB, 8KB, 32KB, and 128KB. We generate a unique set of search patterns per collection, which is used consistently for all the indexes. This is obtained by extracting 100 random substrings of length q from the collection. The search time given is the average time over all the occurrences of all the patterns.

C. Comparison of Compressed Indexes

Finally, we compare our q-gram index of Section II (`II`, for "inverted index") using the variant `BC+LZ,Re-Pair`, our grammar-based index of Section III (`SLP`, for "straight-line program"), and the best representative of previous work based on run-length compressed suffix arrays (`RLCSA` [10]). The latter offers a space-time tradeoff given by a sampling parameter on the text.

Figure 2 shows the result over the synthetic collections. (the missing lines of `II` fall well outside the plots, we use `BV+LZ,repair`). It can be seen that `II` is extremely competitive for small $q = 4$ and $q = 6$, while for $q = 8$ both its space and time performance degrades sharply. For very repetitive sequences (`s0001`), the `SLP` alternative stands out as an excellent choice, offering good space performance for all q (indeed, the index does not depend on q), and good time performance for medium $q \leq 8$. This deteriorates as we look for longer q-grams. The performance of `RLCSA`, instead, is largely independent of q, both in space (where, again, the index does not depend on q), and in time. When the sequences become less repetitive (`s001` and `s01`), `RLCSA` takes over `SLP` in both space and time.

Figure 3 gives the result on our real-life collections. The results are roughly similar to those for `s001` and `s01`: `II` is competitive for small $q \leq 6$, whereas `RLCSA` dominates `SLP`.

V. Conclusions

We have proposed two new compressed indexes specialized on searching short substrings (q-grams) on highly repetitive sequences. A first one (`II`) is particularly relevant for small $q \leq 6$, whereas the other (`SLP`) stands out on very highly repetitive collections (as shown on synthetic data). The real repetitive collections we have found are not yet sufficiently repetitive for `SLP` to be competitive on those, but we expect its properties to become very relevant for the massive repetitive collections that are expected to appear in the next years. An existing approach [10] is still unbeatable when searching for longer strings.

Acknowledgement: We thank Jouni Sirén for his help on using `RLCSA` software.

References

[1] G. M. Church, "Genomes for ALL," *Scientific American*, vol. 294, no. 1, pp. 47–54, 2006.

[2] N. Hall, "Advanced sequencing technologies and their wider impact in microbiology," *The Journal of Experimental Biology*, vol. 209, pp. 1518–1525, 2007.

[3] E. Pennisi, "Breakthrough of the year: Human genetic variation," *Science*, vol. 21, pp. 1842–1843, Dec. 2007.

[4] C. G. Nevill-Manning and I. H. Witten, "Protein is incompressible," in *Proc. 9th DCC*. Washington, DC, USA: IEEE Computer Society, 1999, p. 257.

[5] X. Chen, S. Kwong, and M. Li, "A compression algorithm for DNA sequences and its applications in genome comparison," in *Proc. 4th RECOMB*. New York, NY, USA: ACM, 2000, p. 107.

[6] S. Grümbach and F. Tahi, "A new challenge for compression algorithms: Genetic sequences," *Information Processing and Management*, vol. 30, no. 6, pp. 875–886, 1994.

[7] E. Rivals, J.-P. Delahaye, M. Dauchet, and . Delgrange, "A guaranteed compression scheme for repetitive DNA sequences," in *Proc. 6th DCC*, 1996, p. 453.

[8] H. Sato, T. Yoshioka, A. Konagaya, and T. Toyoda, "DNA data compression in the post genome era," *Genome Informatics*, vol. 12, pp. 512–514, 2001.

Fig. 2. Comparison of compressed indexes over the synthetic collections.

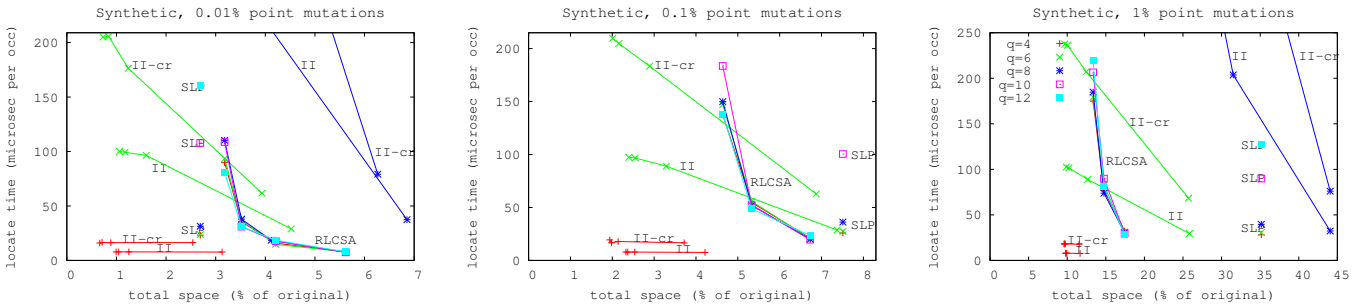
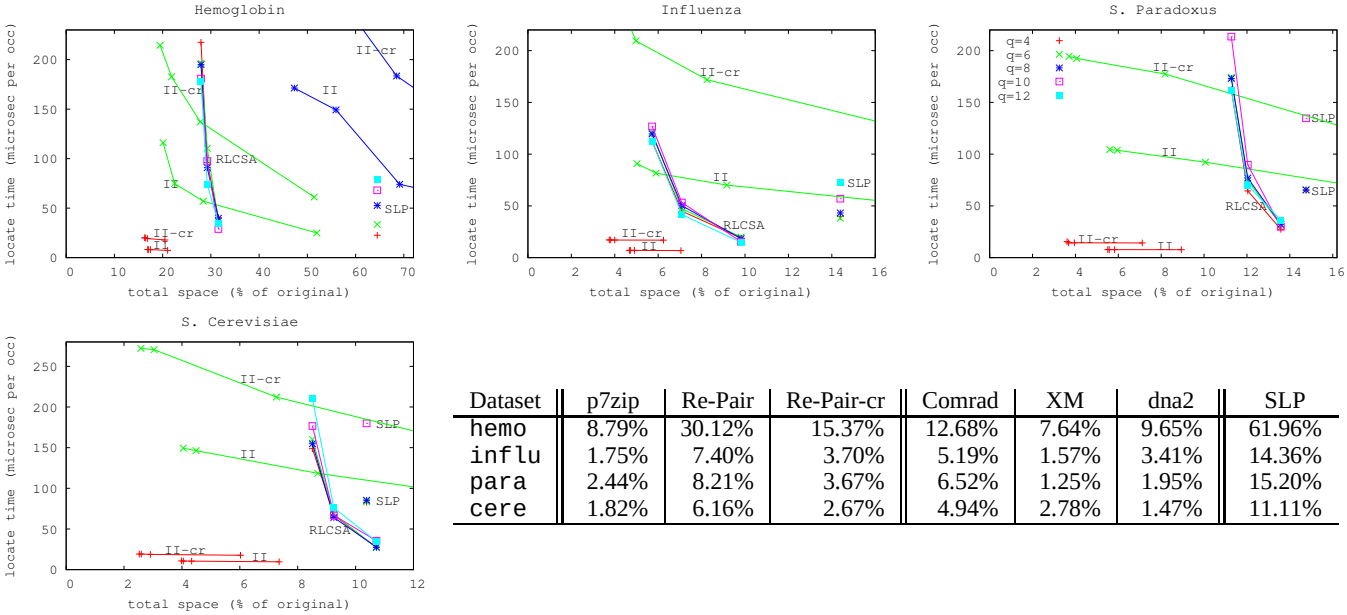


Fig. 3. Comparison of compressed indexes over the real collections. The table shows compression ratios on the 4-letter sequences. The groups are general-purpose compressors, DNA-specific compressors, and our self-index.



Dataset	p7zip	Re-Pair	Re-Pair-cr	Comrad	XM	dna2	SLP
hemo	8.79%	30.12%	15.37%	12.68%	7.64%	9.65%	61.96%
influ	1.75%	7.40%	3.70%	5.19%	1.57%	3.41%	14.36%
para	2.44%	8.21%	3.67%	6.52%	1.25%	1.95%	15.20%
cere	1.82%	6.16%	2.67%	4.94%	2.78%	1.47%	11.11%

[9] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel, "Repetition-based compression of large DNA datasets," in *Proc. 13th RECOMB*, 2009, poster.

[10] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, "Storage and retrieval of individual genomes," in *Proc. 13th RECOMB*, ser. LNCS 5541, 2009, pp. 121–137.

[11] S. Puglisi, W. Smyth, and A. Turpin, "Inverted files versus suffix arrays for locating patterns in primary memory," in *Proc. 13th SPIRE*, ser. LNCS 4209, 2006, pp. 122–133.

[12] J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proc. of the IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.

[13] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[14] F. Claude and G. Navarro, "Self-indexed text compression using straight-line programs," in *Proc. 34th MFCS*, ser. LNCS 5734, 2009, pp. 235–246.

[15] R. N. Horspool, "Practical fast searching in strings," *Software Practice and Experience*, vol. 10, no. 6, pp. 501–506, 1980.

[16] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[17] I. Witten, A. Moffat, and T. Bell, *Managing Gigabytes*, 2nd ed. Morgan Kaufmann Publishers, 1999.

[18] G. Zipf, *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.

[19] H. Williams and J. Zobel, "Compressing integers for fast file access," *The Computer Journal*, vol. 42, no. 3, pp. 193–201, 1999.

[20] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys*, vol. 39, no. 1, p. article 2, 2007.

[21] J. Munro, R. Raman, V. Raman, and S. S. Rao, "Succinct representations of permutations," in *Proc. 30th ICALP*, ser. LNCS 2719, 2003, pp. 345–356.

[22] H. Sakamoto, "A fully linear-time approximation algorithm for grammar-based compression," *J. Discrete Algorithms*, vol. 3, pp. 416–430, 2005.

[23] M. Cao, T. Dix, L. Allison, and C. Mears, "A simple statistical algorithm for biological sequence compression," in *Proc. 29th DCC*, 2007, pp. 43–52.

[24] G. Manzini and M. Rastrero, "A simple and fast DNA compression algorithm," *Software Practice and Experience*, vol. 34, pp. 1397–1411, 2004.

[25] R. González and G. Navarro, "Compressed text indexes with fast locate," in *Proc. 18th CPM*, ser. LNCS 4580, 2007, pp. 216–227.