Indexing Highly Repetitive Collections

Gonzalo Navarro

Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl

Abstract. The need to index and search huge highly repetitive sequence collections is rapidly arising in various fields, including computational biology, software repositories, versioned collections, and others. In this short survey we briefly describe the progress made along three research lines to address the problem: compressed suffix arrays, grammar compressed indexes, and Lempel-Ziv compressed indexes.

1 Introduction

After a years-long race to sequence the first human genome in the early 2000's, sequencing has become a routine activity that costs a few thousand dollars[1] and large sequencing companies are producing thousands of genomes per day[2]. Maintaining databases of millions of genomes will be a real possibility very soon. With a storage requirement of about 715 MB per human genome (about 3×10^9 bases using 2 bits each), storing, say, one million genomes is perfectly realistic (around 700 TB). However, sequence analysis tools require *indexed* access to the data, where one can carry out pattern searches and mining. Such indexes require at the very least 50 bits per base (one pointer), raising the 700 TB to more than 16 PB (petabytes). Those sizes, especially if we require fast indexed access to the data, exceed today's technological possibilities within reasonable cost.

What makes this challenge affordable is that most of those genomes (if we assume they belong to the same species, say human) are very similar to each other — 99.99% similar according to typical figures (although there is some debate about the exact value). If we were able to index such a collection within space proportional to the number of differences between the genomes, and not to their total size, then a one-million genome collection could be indexed within 1.6 TB, which is perfectly feasible. Yet, we still do not know how to do this.

Other scenarios where a set of very similar sequences is indexed and pattern searches are provided on them are software repositories (where versions form a tree or graph structure) and versioned document collections (such as Wikipedia or the Internet Wayback Machine, where versions have a linear structure).

In this short survey we will cover the results achieved on the challenge of storing and indexing those "highly repetitive" sequence collections. We will focus on the following scenario, looking for a balance of generality and simplicity that leads to both useful and algorithmically interesting research:

[1] *The Guardian* Jan 12, 2012, "Company announces low-cost DNA decoding machine".
[2] *The New York Times* Jan 12, 2011, "DNA sequencing caught in deluge of data".

1. The collection has d documents of total length n, where each document is an arbitrary string of symbols over an alphabet of size σ. This is general enough to consider DNA, proteins, source code, natural language, etc. Then the uncompressed data size is $n \log \sigma$ bits (our logarithms are in base 2) and a classical index requires $O(n \log n)$ bits.
2. The collection is *repetitive*, meaning that most documents can be covered by a few chunks that appear in other documents. This captures most cases, in particular those with known linear, tree, etc. version structures as well as those with unknown version structures like DNA collections. Yet, it leaves aside special cases such as reverse complemented repetitions that are frequent in DNA, which must be dealt with separately.
3. We wish to store and index the collection in a way that provides efficient *access*, that is, extracting any substring of any document, and *searches*, that is, locating the occurrences of string patterns in the collection. We ignore more complex searches such as approximate matching, complex pattern matching, sequence mining, etc. Yet this is challenging enough to leave aside methods like encoding the differences, which may support access but not searches.
4. We wish to use space proportional to the "repetitiveness" of the collection. While we seek for techniques that work on any collection without any explicit structure, we analyze them on a simplified case where there is one uncompressible *base* sequence of length ℓ and then $d-1$ other sequences identical to the base one, where s *edit operations* (single character insertions, deletions, and replacements) have been done on them. Ideally, we should achieve $\ell \log \sigma + O(s \log n)$ bits of space, as well as search time within $O((m + occ) \operatorname{polylog} n)$ to find the occ occurrences of a pattern of length m.

2 Compressed Suffix Arrays

The suffix array [18] is a classical structure to support pattern searches. Assume the d documents are concatenated into a single string $T[1, n]$, using a special symbol "\$" to mark the end of the documents. Each string $T[i, n]$ is called a *suffix* and is identified with its starting point i. The *suffix array* $A[1, n]$ is a permutation of $[1..n]$ where all the suffixes are listed in lexicographic order. The occurrences of $P[1, m]$ in T can be seen as the suffixes of T that start with P, and these form a contiguous interval of A that can be binary searched in time $O(m \log n)$, where m owes to the time needed to compare P with a suffix of A.

The suffix array uses $n \log n$ bits, but can be compressed into $O(n \log \sigma)$ bits by means of the Ψ *function* [13, 23], $\Psi(i) = A^{-1}[(A[i] \bmod n) + 1]$, which tells where the value $A[i] + 1$ appears in A. With a bitmap $D[1, n]$ that marks the points in A where the first letter of suffixes change, and a string $S[1, \sigma]$ noting the different symbols of T in order, we know that the first letter of suffix $T[A[i], n]$ is $S[rank(D, i)]$, where $rank(D, i)$ counts the number of 1s in $D[1, i]$. Moreover, the second letter is $S[rank(D, \Psi(i))]$, the third is $S[rank(D, \Psi^2(i))]$, and so on. With a preprocessing of D to solve $rank$ in constant time, we can do the binary search in $O(m \log n)$ time using Ψ, S and D, and without A or T.

In principle Ψ would also require $n \log n$ bits, but it is shown to be compressible as it is covered by σ increasing ranges. In fact, those ranges feature a much richer structure [12, 21], which allows one to represent the whole index within $nH_k(T) + o(n \log \sigma)$ bits for any $k < \log_\sigma n$ (roughly), where $H_k(T) \leq \log \sigma$ is the empirical k-th order entropy of T [19, 21]. This is a statistical compressibility measure sensitive to symbols distribution but blind to long-range repetitions: If we concatenate two copies of T, its k-th order entropy is $2nH_k(TT) \geq 2nH_k(T)$, so the space of such compressed indexes simply doubles [16].

What is most interesting for us is that, when the text contains long and frequent repetitions of a string, long *runs* of consecutive values appear on Ψ [10, 17]. In our model of s edits, it is shown that Ψ contains ℓ runs of length d when $s = 0$, and then each edit breaks, *on average* (assuming the base text and the edits are uniformly generated), $O(\log_\sigma \ell)$ runs. As a consequence, Ψ can be represented using $\ell \log \sigma + O(s \log_\sigma \ell \log n)$ bits on average.

A further problem is that, since we do not store A, knowing the interval of A given by the binary search is not sufficient to output the occurrence positions (which would be the content of the cells of A within the range). This can be done within $O(\log n)$ time per occurrence if we use $O(n)$ further bits of space for *sampling* the suffix array [23]. Such sampling also provides extraction of any substring of T of length t in time $O(t + \log n)$. While various attempts to reduce those $O(n)$ bits have been made [17], they have not been successful in practice. Further, they [17] show that, while LZ78-based compression [25] is poor on highly repetitive sequences, LZ77-based compression [24] is extremely promising.

3 Lempel-Ziv (LZ77) Compressed Indexes

The compression that most accurately reflects the kind of repetitiveness we wish to capture is Lempel-Ziv's, particularly the LZ77 parsing [24]. Here we advance in T, and at each step generate a new *phrase* by taking the longest prefix of the remaining text that has already appeared before in T, plus one further letter. Let us call z the number of phrases generated by such a parsing. It is not hard to see that, in our model, it holds $z \leq \ell / \log_\sigma \ell + s$.

Extracting an arbitrary substring of T is not easy in an LZ77 parsing. If we call $h \leq n$ the "height" of the parse, that is, the maximum number of times a single character is copied, then a substring of length t is extracted in time $O(t\,h)$ (variants that speed this up [16] may produce more phrases than edits).

Searching for patterns yields further complications. An occurrence of P in T may be split across phrases, that is, a prefix $P[1, i]$ may match a suffix of a phrase, and the corresponding suffix $P[i+1, m]$ may match a prefix of the concatenation of the following phrases. Such occurrences cut by a phrase boundary are called *primary*, whereas the others are called *secondary*. The strategy of LZ77-based indexes [15, 16] is to find both types of occurrences with different means.

Let $T = Z_1 \dots Z_z$ be the partition of T into phrases. For primary occurrences, we index all reverse phrases Z_j^{rev} and all suffixes starting at phrase boundaries, $Z_k Z_{k+1} \dots Z_z$. Both sets are connected in a grid where the rows correspond

to the lexicographically sorted Z_j^{rev} and the columns to the lexicographically sorted $Z_k Z_{k+1} \ldots Z_z$. Points in this grid connect reverse phrases Z_j^{rev} with the following suffix, $Z_{j+1} Z_{j+2} \ldots Z_z$. At query time, P is partitioned into $P[1,i]$ and $P[i+1,m]$ in the $m-1$ possible ways. We find the interval of the reverse phrases starting with $(P[1,i])^{rev}$ (i.e., phrases ending with $P[1,i]$) and the interval of the phrase-aligned suffixes starting with $P[i+1,m]$. Then each point in the grid in the intersection of the row and column ranges is precisely one primary occurrence.

For secondary occurrences, a structure to describe which portions of T are copied where, is used to track copies of areas where primary occurrences appear. Those copies are secondary occurrences, which must be recursively tracked for further secondary occurrences, until all are reported.

The most recent index of this family [16] uses $O(z \log n)$ bits of space and can search for P in time $O(m^2 h + (m + occ) \log^{\epsilon} z)$, for any constant $\epsilon > 0$, if we use the best grid representation that fits in this space [3]. The term m^2 owes to the search for all the partitions of P, and the h factor to the time to extract the phrase prefixes/suffixes when determining the intervals of rows and columns. The term $O(m \log^{\epsilon} z)$ refers to the $m-1$ range searches on the grid, and $O(occ \log^{\epsilon} z)$ to the time to report each point (i.e., primary occurrence) in the grid. Secondary occurrences take just constant time each. Note that the h factor is highly undesirable, as it is limited only by n in the worst case.

4 Grammar Compressed Indexes

Grammar compression is a successful approach to factor out repetitiveness in a text. It derives a context-free grammar that generates (only) T, and such a grammar can be small if T is repetitive. While finding the smallest grammar that generates T is NP-complete and popular methods offer poor approximation ratios [22, 5], an $O(\log n)$ approximation is easy to achieve by converting the LZ77 factorization, which lower-bounds the smallest grammar, into a balanced grammar [22]. A more sophisticated approximation [4] achieves ratio $O(\log(n/g^*))$, where g^* is the size of the smallest grammar. We will call g the size (total length of the rules) of our grammar that generates T, where it holds $z \leq g \leq z \log n$. In our model it is easy to obtain $g \leq \ell / \log_\sigma \ell + s \log n$. It has been shown that a grammar representation using $O(g \log n)$ bits provides access to any substring of T of length t in time $O(t + \log n)$ [2], much better than with LZ77 compression.

The concept of primary and secondary occurrences is useful here too. Given a rule $X \to ABC \ldots$, a prefix $P[1,i]$ may match a suffix of the string generated by A, and the corresponding suffix $P[i+1,m]$ may match a prefix of the string generated by $BC \ldots$, and thus P will have a primary occurrence inside the string generated by X. Occurrences of X elsewhere yield secondary occurrences of P.

In the most recent grammar-based compressed index [8], primary occurrences are handled very much as in LZ77 parsings, by indexing the reverse string generated by nonterminals and the strings generated by the following sequences of nonterminals on right hands of the grammar productions. Tracking secondary occurrences is done by using a pruned version of the parse tree of T, where all

Table 1. Simplified complexities for current approaches to index repetitive collections.

Approach	Space $+\ell \log \sigma$	Time $+O(occ \log n)$	Extract time
Suffix arrays [17] (avg. space)	$O(n + s \log_\sigma \ell \log n)$	$O(m \log n)$	$O(t + \log n)$
Grammar compression [2, 8]	$O(s \log^2 n)$	$O(m^2 + m \log n)$	$O(t + \log n)$
Lempel-Ziv compression [16]	$O(s \log n)$	$O(m^2 h + m \log n)$	$O(t\, h)$

the nodes labeled by a given nonterminal containing a primary occurrence are found. To find the text position of an occurrence at a node v, the grammar tree is traversed upwards from v to the root. Each label (i.e., nonterminal) found in this upward traversal is a source of further secondary occurrences, and as such is sought in the tree node labels. The grammar is transformed so that it is guaranteed that any nonterminal appears at least twice in the tree, and thus the upward traversal time is amortized with more secondary occurrences.

The resulting structure requires $O(g \log n)$ bits of space and supports searches in time $O(m^2 + (m + occ) \log^\epsilon g)$ for any constant $\epsilon > 0$ [8].

5 Conclusions

Table 1 summarizes the state of the art in very broad terms. While suffix arrays offer the ideal search time $O((m+occ) \log n)$, they are far from the ideal space of $\ell \log \sigma + O(s \log n)$, which is offered only by Lempel-Ziv indexes. These in turn are far from the ideal time complexity, both for searching and for extracting substrings. Grammar compression offers an intermediate space/time tradeoff that might be attractive. This theoretical picture coincides pretty well with practical experiences on biological and natural language sequence collections [6, 7].

The most obvious challenge ahead is to combine good space and time on repetitive sequences. For example, a recent development combining grammars and LZ77 parsing [11] achieves $O(m^2 + (m + occ) \log \log z)$ search time and $O(z \log n \log \log z)$ bits of space (i.e., slightly superlinear on the LZ77 compressed size). Other close relatives of LZ77 parsings may also yield interesting indexes in particular application scenarios [14, 20, 9]. A further challenge is to support more complex searches, for example suffix tree functionality [1].

References

1. A. Abeliuk and G. Navarro. Compressed suffix trees for repetitive texts. In *Proc. 19th SPIRE*, LNCS 7608, pages 30–41, 2012.
2. P. Bille, G. Landau, R. Raman, K. Sadakane, S. Rao Satti, and O. Weimann. Random access to grammar-compressed strings. In *Proc. 22nd SODA*, pages 373–389, 2011.
3. T. Chan, K. Larsen, and M. Patrascu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th SoCG*, pages 1–10, 2011.
4. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and A. Shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *Proc. 34th STOC*, pages 792–801, 2002.

5. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theo.*, 51(7):2554–2576, 2005.
6. F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. 10th BIBE*, pages 86–91, 2010.
7. F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Indexes for highly repetitive document collections. In *Proc. 20th CIKM*, pages 463–468, 2011.
8. F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th SPIRE*, LNCS 7608, pages 180–192, 2012.
9. H.-H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative Lempel-Ziv self-index for similar sequences. In *Proc. FAW-AAIM*, pages 291–302, 2012.
10. J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comp. Sci.*, 410(51):5354–5364, 2009.
11. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. Puglisi. A faster grammar-based self-index. In *Proc. 6th LATA*, pages 240–251, 2012.
12. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
13. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comp.*, 35(2):378–407, 2006.
14. S. Huang, T. W. Lam, W. K. Sung, S. L. Tam, and S. M. Yiu. Indexing similar DNA sequences. In *Proc. 6th AAIM*, pages 180–190, 2010.
15. J. Kärkkäinen. *Repetition-Based Text Indexing.* PhD thesis, Dept of Comp. Sci., Univ. of Helsinki, Finland, 1999.
16. S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theor. Comp. Sci.*, 2012. To appear. Earlier versions in *Proc. DCC'10* and *Proc. CPM'11.*
17. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comp. Biol.*, 17(3):281–308, 2010.
18. U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, pages 935–948, 1993.
19. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
20. S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. In *Proc. 18th SPIRE*, pages 398–409, 2011.
21. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.
22. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theo. Comp. Sci.*, 302(1-3):211–222, 2003.
23. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Alg.*, 48(2):294–313, 2003.
24. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theo.*, 23(3):337–343, 1977.
25. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theo.*, 24(5):530–536, 1978.