Advances in DNA sequencing mean that databases of thousands of human genomes will soon be commonplace. In this paper, we introduce a simple technique for reducing the size of conventional indexes on such highly repetitive texts. Given upper bounds on pattern lengths and edit distances, we preprocess the text with the lossless data compression algorithm LZ77 to obtain a filtered text, for which we store a conventional index. Later, given a query, we find all matches in the filtered text, then use their positions and the structure of the LZ77 parse to find all matches in the original text. Our experiments show that this also significantly reduces query times.

1. Introduction

The British government recently announced plans to sequence the genomes of up to one hundred thousand citizens [1]. Although sequencing so many people is still a challenge, the cost has dropped sharply in the past few years, and the decline is expected to continue. Now, scientists must consider how to store such a massive genomic database in a useful form.

Because human genomes are very similar, this database will be highly repetitive and easy to compress well with, for example, the lossless data compression algorithm LZ77 [2]. Unfortunately, conventional indexes [3–5] for approximate pattern matching—a basic operation in bioinformatics—are based on FM-indexes [6] or other technologies that do not take good advantage of repetitive structure. Therefore, these indexes quickly outgrow internal memory when applied to many genomes and must then reside on disk, which slows them down by orders of magnitude.
There are already some experimental LZ- and grammar-based compressed indexes for exact pattern matching [7–13] and these could eventually provide a basis for approximate pattern matching as well (see [14]), but science cannot wait. In this paper, we introduce a simple technique, hybrid indexing, for reducing the size of conventional indexes on highly repetitive texts while preserving most or all of their functionality.

Given upper bounds on pattern lengths and edit distances, we pre-process the text with LZ77 to obtain a filtered text, for which we store a conventional index. Later, given a query, we find all matches in the filtered text, then use their positions and the structure of the LZ77 parse to find all matches in the original text.

We describe our hybrid index in more detail in §2; this includes details of our implementation.1 In §3, we present experimental results, which show that our technique also significantly reduces query times.

2. Hybrid indexing

Suppose we are given upper bounds on pattern lengths and edit distances, and asked to index a text \( T[1..n] \). A query to the index consists of a pattern and an edit distance (which is 0 in the case of exact matching). We store different data structures to be able to find queries’ primary matches and secondary matches. A match for a query is a substring within the given edit distance of the given pattern; it is primary if it contains the first occurrence of a distinct character in \( T \) or crosses a boundary between phrases in the LZ77 parse of \( T \), and secondary otherwise.

To be able to find primary matches, we pre-process the text with LZ77 to obtain a filtered text, which is essentially the subsequence of \( T \) containing characters close enough to phrase boundaries in \( T \)’s LZ77 parse, where ‘close enough’ depends on the given upper bounds. We store a conventional index on this filtered text, and a mapping from it to \( T \). To be able to find secondary matches, we store a data structure by Kärkkäinen & Ukkonen [15].

Later, given a query, we use the conventional index to find all matches in the filtered text; use the mapping to determine which of those matches correspond to primary matches in \( T \); use the mapping again to find those primary matches’ positions in \( T \); and apply Kärkkäinen & Ukkonen’s data structure.

We briefly describe LZ77 in §2a. In §2b, we give more details about the filtered text and how we find primary matches. We describe Kärkkäinen & Ukkonen’s data structure in §2c. Finally, in §2d, we describe details of our implementation.

(a) LZ77

We use the variant of LZ77 according to which, for each phrase \( T[i..j] \) in the parse of \( T \), either \( i = j \) and \( T[i] \) is the first occurrence of that distinct character, or \( T[i..j] \) occurs in \( T[1..j - 1] \) but \( T[i..j + 1] \) does not occur in \( T[1..j] \). In the first case, \( T[i] \) is encoded as itself. In the second case, \( T[i..j] \) is encoded as the pair \( (i', j - i + 1) \), where \( i' \) is the starting point of the leftmost occurrence of \( T[i..j] \) in \( T \); we call this leftmost occurrence \( T[i..j] \)’s source.

For example, if \( T \) is

```
99-bottles-of-beer-on-the-wall-99-bottles-of-beer-
take-one-down-and-pass-it-around-98-bottles-of-beer-on-the-wall-
98-bottles-of-beer-on-the-wall-98-bottles-of-beer-
take-one-down-and-pass-it-around-97-bottles-of-beer-on-the-wall-
97-bottles-of-beer-on-the-wall-97-bottles-of-beer-
take-one-down-and-pass-it-around-96-bottles-of-beer-on-the-wall-
```

1http://www.cs.helsinki.fi/u/gagie/hybrid.
then the parse of \(T\) (with parentheses around phrases) is

\[
\begin{align*}
(9) & (9) (-) (b) (a) (t) (t) (l) (e) (s) (-) (o) (f) (-b) (e) (t) (-o) (n) (-) (t) (h) (e) (-) \\
(w) & (a) (l) (l) (-) (99\text{-bottles-of-beer-}) (t) (a) (k) (e-) (on) (e-) (d) (o) (w) (n-) (a) (n) (d) \\
(-) & (p) (a) (s) (-) (l) (t) (-a) (t) (o) (u) (nd-) (9) (8) (-\text{bottles-of-beer-on-the-wall}-9) \\
(8\text{-bottles-of-beer-on-the-wall}-98\text{-bottles-of-beer-}) & (\text{take-one-down-and-pass-it-around-9}) (7) (-\text{bottles-of-beer-on-the-wall}-9) \\
(7\text{-bottles-of-beer-on-the-wall-97\text{-bottles-of-beer-}}) & (\text{take-one-down-and-pass-it-around-9}) (6) (-\text{bottles-of-beer-on-the-wall}-9) \ldots
\end{align*}
\]

and \(T\)'s encoding is

\[
\begin{align*}
9 & (2, 1) \text{-bot} (6, 1) \text{les} (3, 1) (5, 1) f (3, 2) (9, 1) (9, 1) r (11, 2) n \\
(3, 1) & (6, 1) h (9, 1) (3, 1) \text{wa} (8, 1) (8, 1) (3, 1) (1, 19) (6, 1) (28, 1) k \\
(25, 2) & (20, 2) (25, 2) d (5, 1) (27, 1) (21, 2) (28, 1) (21, 1) (60, 1) (3, 1) p \\
(28, 1) & (10, 1) (10, 2) \text{i} (6, 1) (64, 2) (18, 1) (5, 1) \text{u} (66, 3) (1, 1) 8 \\
(3, 30) & (85, 49) (51, 34) \text{t} (3, 30) (199, 49) (51, 34) \text{t} (3, 30) \ldots
\end{align*}
\]

Note some phrases—e.g. \text{8\text{-bottles-of-beer-on-the-wall}-98\text{-bottles-of-beer—} overlap their own sources. In addition, while the first verse takes the first four lines of the encoding, the next three verses together take only the last line. This is typical of \text{LZ77}'s performance on repetitive inputs. Finally, although these verses are annoyingly similar, they are much less similar than human genomes.

\[(b) \text{ Finding primary matches}\]

Let \(M\) and \(K\) be the given upper bounds on pattern lengths and edit distances. Let \(T_{M,K}\) be the text containing only those characters of \(T\) within distance \(M + K - 1\) of their nearest phrase boundary in the \text{LZ77} parse; characters not adjacent in \(T\) are separated in \(T_{M,K}\) by \(K + 1\) copies of a special character \# not in the normal alphabet. In other words, to obtain \(T_{M,K}\), in each phrase, we replace all but the \(M + K - 1\) characters at either end of that phrase by \(K + 1\) copies of \#.

For example, if \(T\) is the example given above then \(T_{4,1}\) is

\[
\begin{align*}
\text{99\text{-bottles-of-beer-on-the-wall-99\text{-b\#eer-}}} \\
\text{take-one-down-and-pass-it-around-98\text{-b\#ll-}} \\
\text{98\text{-bot\#e-e\#r-take\#nd-97\text{-bot\#ll-}}} \\
\text{97\text{-bot\#e-e\#r-take\#nd-96\text{-bot\#ll-}}} \ldots
\end{align*}
\]

or, with parentheses indicating the phrases of \(T\),

\[
\begin{align*}
(9) & (9) (-) (b) (a) (t) (t) (l) (e) (s) (-) (o) (f) (-b) (e) (t) (-o) (n) (-) (t) (h) (e) (-) \\
(w) & (a) (l) (l) (-) (99\text{-b\#e-e\#r-}) (t) (a) (k) (e-) (on) (e-) (d) (o) (w) (n-) (a) (n) (d) \\
(-) & (p) (a) (s) (-) (l) (t) (-a) (t) (o) (u) (nd-) (9) (8) (-\text{bot\#ll-9}) (8\text{-bo-\#e-e\#r-}) \\
(\text{take\#nd-9}) & (7) (-\text{bot\#ll-9}) (7\text{-bo-\#e-e\#r-}) (\text{take\#nd-9}) (6) (-\text{bot\#ll-9}) \ldots
\end{align*}
\]

Note that, for any substring of \(T\) with length at most \(M + K\) that contains the first occurrence of a distinct character in \(T\) or crosses a phrase boundary in the \text{LZ77} parse of \(T\), there is a corresponding and equal substring in \(T_{M,K}\).

We do not store \(T_{M,K}\) itself explicitly, but we store a conventional index \(I_{M,K}\) on \(T_{M,K}\). We assume \(I_{M,K}\) can handle queries with pattern lengths up to \(M\) and edit distances up to \(K\). Because \(T_{M,K}\) consists of at most \(2M + 3K - 1\) characters for each phrase, if \(T\) is highly repetitive and \(M\) and \(K\) are reasonable, then \(I_{M,K}\) should be smaller than a conventional index on all of \(T\). In addition, for any valid query and any match in \(T_{M,K}\), there is at least one match (primary or secondary) in \(T\), so querying \(I_{M,K}\) should be faster than querying a conventional index for all of \(T\).

Let \(L\) be the sorted list containing the positions of the first character of each phrase in the parse of \(T\), and let \(L_{M,K}\) be sorted lists containing the positions of the corresponding characters in \(T_{M,K}\).
We store $L$ and $L_{MK}$. If $T[i]$ is the first occurrence of a distinct character in $T$ and $T_{MK}[j]$ is the corresponding character in $T_{MK}$, then we mark $j$ in $L_{MK}$.

For our example, $L$ is

\[ [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 51, 52, 53, 54, 56, 58, 60, 61, 62, 63, 65, 66, 67, 68, 69, 70, 71, 72, 74, 75, 76, 78, 80, 81, 84, 85, 86, 88, 116, 199, 200, 230, 279, 313, 314, \ldots] \]

and $L_{4,1}$ (with asterisks indicating marked numbers) is

\[ [1\star, 2, 3\star, 4\star, 5\star, 6\star, 7, 8\star, 9\star, 10\star, 11, 12, 13\star, 14, 16, 17, 18\star, 19, 21\star, 22, 23, 24\star, 25, 26, 27\star, 28, 29, 30, 31, 32, 42, 43, 44\star, 45, 47, 49, 51\star, 52, 53, 54, 56, 57, 58, 59, 60\star, 61, 62, 63, 65\star, 66, 67, 69, 70, 71\star, 72, 75, 76\star, 77, 87, 98, 108\star, 109, 119, 130, 140\star, 141, \ldots ] \]

Given the endpoints $i$ and $j$ of a substring $T_{MK}[i..j]$ of $T_{MK}$ that does not include any occurrences of #, we can use $L_{MK}$ to determine whether the corresponding substring $T[i'..j']$ of $T$ contains the first occurrence of a distinct character in $T$ or crosses a phrase boundary in the LZ77 parse of $T$. To do this, we use binary search to find $i'$'s successor $L_{MK}[s]$. There are three cases to consider:

- if $i < L_{MK}[s] \leq j$, then $T[i'..j']$ crosses a phrase boundary;
- if $i \leq j < L_{MK}[s]$, then $T[i'..j']$ neither contains the first occurrence of a distinct character nor crosses a phrase boundary;
- if $i = L_{MK}[s] \leq j$, then $T[i'..j']$ contains the first occurrence of a distinct character or crosses a phrase boundary if and only if $L_{MK}[s]$ is marked or $L_{MK}[s + 1] \leq j$.

In addition, if $T[i'..j']$ contains the first occurrence of a distinct character or crosses a phrase boundary, then $i' = L[s] - L_{MK}[s] + i$ and $j' = i' + j - i + 1$. In other words, we can use $L$ and $L_{MK}$ as a mapping from $T_{MK}$ to $T$.

Given a query consisting of a pattern $P[1..m]$ with $m \leq M$ and an edit distance $k \leq K$, we use $I_{MK}$, $L$ and $L_{MK}$ to find all primary matches in $T$. First, we query $I_{MK}$ to find all matches in $T_{MK}$. We then discard any matches containing copies of #. We use binary search on $L_{MK}$, as described above, to determine which of the remaining matches correspond to primary matches in $T$. Finally, we use $L$ and $L_{MK}$, as described above, to find those primary matches’ positions in $T$.

(c) Finding secondary matches

By definition, any secondary match is completely contained in some phrase. In addition, a phrase contains a secondary match if and only if that phrase’s source contains an earlier match (primary or secondary). More generally, a phrase contains a substring if and only if that phrase’s source contains the same substring. It follows that each secondary match is an exact copy of some primary match; more importantly, Kärkkäinen & Ukkonen [15] showed how, once we have found all the primary matches, we can then find all the secondary matches from the structure of the LZ77 parse.

To do this, for each primary match $T[\ell..r]$, we find each phrase $T[i..j]$ whose source $T[i'..i' + j - i]$ includes $T[\ell..r]$—i.e. such that $i' \leq \ell \leq r \leq i' + j - i$. Note $T[\ell'..r'] = T[\ell..r]$, where $\ell' = i + \ell - i'$ and $r' = i + 2\ell - i' - r$. We record $T[\ell'..r']$ as a secondary recurrence and recurse on it.

To be able to find quickly all the sources that cover a match, we store a data structure for two-sided range reporting on the $n \times n$ grid containing a marker at $(i', j')$ for every phrase’s source $T[i'..j']$. With each marker, we store as a satellite datum the starting point of the actual phrase. In other words, if a phrase $T[i..j]$ is encoded as $(i', j' - i + 1)$ by LZ77, then there is a marker on the grid at $(i', i' + j - i)$ with satellite datum $i$. 


For example, for the phrases shown in §2a, there are markers at

\[(1, 1) (6, 6) (3, 3) (5, 5) (3, 4) (9, 9) (9, 9) (11, 12) (3, 3) (6, 6) (9, 9)
(3, 3) (8, 8) (8, 8) (3, 3) (1, 19) (6, 6) (28, 28) (25, 26) (20, 21) (25, 26)
(5, 5) (27, 27) (21, 22) (28, 28) (21, 21) (60, 60) (3, 3) (28, 28) (10, 10)
(10, 11) (6, 6) (64, 65) (18, 18) (5, 5) (66, 68) (1, 1) (3, 32) (85, 133)
(51, 84) (3, 32) (199, 247) (51, 84) (3, 32) ...\]

with satellite data

\[2, 7, 11, 12, 14, 16, 17, 19, 22, 23, 25, 26, 29, 30, 31, 32, 51, 52, 54, 56, 58, 61,
62, 63, 65, 66, 67, 68, 70, 71, 72, 75, 76, 78, 79, 81, 84, 86, 116, 165, 200, 230,
279, 314, ...\]

Note that the markers are simply the encodings of the phrases not consisting of the first occurrences of distinct characters in \(T\) (see §2a), with each second component \(j - i + 1\) replaced by \(i' + j - i\), where \(i'\) is the first component. In addition, the satellite data are the positions of the first characters in those phrases, which are a subset of \(L\).

(d) Implementation

Recall that, to be able to find primary matches, we store the conventional index \(I_{M,K}\) on \(T_{M,K}\), the list \(L\) and the list \(L_{M,K}\). Because we want our hybrid index to be flexible, we do not consider how \(I_{M,K}\) works. (However, we note that it may sometimes be better to use fewer than \(K + 1\) copies of \(\#\) as separators, as they serve only to limit the worst-case number of matches in \(T_{M,K}\).)

We store \(L\) and \(L_{M,K}\) using gap coding—i.e. storing the differences between consecutive values—with every \(g\)th value stored un-encoded, where \(g\) is a parameter. We write the differences as \((\lfloor \log_2 d \rfloor + 1)\)-bit integers, where \(d\) is the largest difference in the list, and we write the un-encoded values as \((\lfloor \log_2 n \rfloor + 1)\)-bit integers. To speed up binary search in \(L_{M,K}\), we also sample every \(b\)th value in it, where \(b\) is another parameter (typically a multiple of \(g\)).

Instead of marking values in \(L_{M,K}\), we store an array containing the position in \(L_{M,K}\) of the first occurrence of each distinct character, in order of appearance. We note, however, that this array is only necessary if there may be matches of length one.

To be able to find secondary matches, we store a data structure for two-sided range reporting on the grid described in §2c. To build this data structure, we sort the points by their \(x\)-coordinates. We store the sorted list \(X\) of \(x\)-coordinates using gap encoding, with every \(g\)th value stored un-encoded as before, and every \(b\)th value sampled to speed up binary search. We store a position-only range-maximum data structure (see [16]) over the list \(Y\) of \(y\)-coordinates, sorted by \(x\)-coordinate. Finally, we store each satellite datum as a \((\lfloor \log_2 z \rfloor + 1)\)-bit pointer to the cell of \(L\) holding that datum, where \(z\) is the number of phrases.

We need not store points’ \(y\)-coordinates explicitly because, if a point has \(x\)-coordinate \(i'\) and satellite datum \(i\), then that point’s \(y\)-coordinate is \(i' + j - i\), where \(j + 1\) is the value that follows \(i\) in \(L\). (We append \(n + 1\) to \(L\) to ensure that we can always compute \(j\) in this way, although this is not necessary when the last character of \(T\) is a special end-of-file symbol.) Because we can access \(i', i\) and \(j\), we can access \(Y\).

Once we have found all primary matches, we apply recursive two-sided range reporting. To do this, we place the endpoints of the primary matches in a linked list and set a pointer to the head of the list. Until we reach the end of the list, for each match \(T[\ell..r]\) in the list, we repeat the following procedure:

1. We use binary search to find \(\ell\)’s predecessor \(X[k]\) in \(X\).
2. We use recursive range-maximum queries to find the values in \(Y[1..k]\) at least \(r\).
3. For each point \((i', j')\) we find with \(i' \leq \ell \leq r \leq j'\), we compute \(\ell'\) and \(r'\) as described in §2c.
4. We append the pair \((\ell', r')\) of endpoints to the list and move the pointer forward one element in the list.

When we finish, the list contains the endpoints of all primary matches followed by the endpoints of all secondary matches.

3. Experiments

In our experiments, we compared a hybrid index based on an FM-index for the filtered text, to an FM-index for the original text. We always used the same implementation of an FM-index\(^2\) with the same parameters, chosen to achieve reasonable compression for highly repetitive datasets. We used an FM-index instead of a popular index for compressed pattern matching because the latter are usually heavily optimized to take advantage of, for example, multiple processors; we plan to compare against them after we have parallelized the hybrid index. We performed our experiments on an Intel Xeon with 96 GB RAM and eight processors at 2.4 GHz with 12 MB cache, running LINUX 2.6.32-46-server. We compiled both indexes with \texttt{g++} using full optimization.

We used benchmark datasets from the repetitive corpus of the Pizza\&Chili website.\(^3\) Specifically, we used the following files:

- \texttt{cere} — 37 \textit{Saccharomyces cerevisiae} genomes from the Saccharomyces Genome Resequencing Project;
- \texttt{einstein} — versions of the English Wikipedia page on Albert Einstein up to 10 November 2006;
- \texttt{fib41} — the 41st Fibonacci word \(F_{41}, \text{where } F_1 = 0, F_1 = 1, F_i = F_{i-1}F_{i-2};\)
- \texttt{kernel} — 36 versions of the LINUX 1.0.x and 1.1.x kernel.

We generally set \(M = 100\), as that seemed a reasonable value for many applications. Because standard FM-indexes do not support approximate pattern matching, we set \(K = 0\) throughout. Based on preliminary tests, we set the sampling parameters \(g\) and \(b\) for our hybrid index to 32 and 512, respectively. Note that these parameters have no effect on the FM-indexes.

Table 1 shows the sizes of the uncompressed files, the files compressed with \texttt{7zip}\(^4\) (which does not support pattern matching), the FM-indexes and the hybrid indexes. It also shows the construction times for the FM-indexes and hybrid indexes. The times to build the hybrid indexes include the times to compute the files’ LZ77 parses (which, in turn, include the times to build the files’ suffix arrays).

To estimate how well hybrid indexing takes advantage of repetitive structure, relative to FM-indexing, we truncated \texttt{cere} at 100, 200, 300 and 400 MB, then built FM-indexes and hybrid indexes for those prefixes. Figure 1 shows the sizes of those indexes.

For 10, 20, 40 and 80, we randomly chose 3000 non-unary substrings that length from \texttt{cere} and searched for them with its FM-index and hybrid. Figure 2 shows the average query times, using a logarithmic scale. The large difference between the query times for patterns of length 10 and those of length 20 seems to be because there are usually far more matches for patterns of length 10; the average time per match stayed roughly the same for all four lengths.

On reflection, it is not surprising that the hybrid index performs well here: while the FM-index finds all matches with its \texttt{locate} functionality, the hybrid index finds secondary matches with two-sided range reporting, which is relatively fast; because \texttt{cere} consists of 37 genomes from individuals of the same species, most matches are secondary.

\(^2\)https://github.com/simongog/sdsl-lite.
\(^3\)http://pizzachili.dcc.uchile.cl/repcorpus.html.
\(^4\)http://www.7zip.org.
Table 1. Sizes of the uncompressed files, the files compressed with 7zip, the FM-indexes and the hybrid indexes; also construction times for the FM-indexes and hybrid indexes.

|       | file (MB) | 7zip (MB) | FM      | hybrid |
|-------|-----------|-----------|---------|--------|
| cere  | 440       | 5         | 88 MB, 34 MB, 136 s | 1218 s |
| einstein | 445      | 0.3       | 26 MB, 2 MB, 139 s | 161 s |
| fib41  | 256       | 0.5       | 5 MB, 0.02 MB, 121 s | 134 s |
| kernel | 246       | 2         | 41 MB, 11 MB, 62 s | 287 s |

Figure 1. Index sizes for prefixes of cere of 100, 200, 300 and 400 MB.

Figure 2. Average query times for 3000 patterns.
4. Conclusions and future work

We have introduced a simple technique, hybrid indexing, for reducing the size of conventional indexes on highly repetitive texts. In our experiments, this technique worked well on benchmark datasets and seemed to scale well. It also significantly reduced query times. We plan to optimize our implementation to use, for example, parallelism across multiple processors, use a more powerful conventional index on the filtered texts, and then compare our hybrid index with popular conventional indexes for approximate pattern matching.

We are also working to optimize hybrid indexing in other ways. For example, readers may have noted that, in our example $T_{4,1}$, there are many copies of "eer-take". Including all these copies seems wasteful, because they could be replaced by, for example, dummy phrases. We are currently testing whether this notably further reduces the size of hybrid indexes in practice.

5. Postscript

We first circulated the basic ideas contained in this paper in a preprint [17, section 1, p. 3] in August 2012, then presented them at a scientific meeting of the Royal Society of London in February 2013. We have now learnt that Wandelt et al. [18] independently and roughly simultaneously developed many of the same ideas and submitted them in March 2013.

Acknowledgements. Many thanks to Paweł Gawrychowski, Juha Kärkkäinen, Veli Mäkinen, Gonzalo Navarro, Jouni Sirén and Jorma Tarhio, for helpful discussions.

Funding statement. This research was supported by Fondecyt grant no. 1-110066, the Helsinki Institute for Information Technology, and Academy of Finland grant nos. 134287, 250345 (CoECCR) and 258308. Parts of this research were done while the first author was visiting the University of Helsinki and while the second author was at Aalto University.

References

1. HM Government. 2012 Strategy for UK life sciences: one year on. See http://www.bis.gov.uk/assets/biscore/innovation/docs/s/12-1346-strategy-for-uk-life-sciences-one-year-on.
2. Ziv J, Lempel A. 1977 A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory 23, 337–343. (doi:10.1109/TIT.1977.1055714)
3. Langmead B, Trapnell C, Pop M, Salzberg SL. 2009 Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biol. 10, R25. (doi:10.1186/gb-2009-10-3-r25)
4. Li H, Durbin R. 2009 Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics 25, 1754–1760. (doi:10.1093/bioinformatics/btp324)
5. Liu C-M et al. 2012 SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. Bioinformatics 28, 878–879. (doi:10.1093/bioinformatics/bts061)
6. Ferragina P, Manzini G. 2005 Indexing compressed text. J. ACM 52, 552–581. (doi:10.1145/1082036.1082039)
7. Arroyuelo D, Navarro G, Sadakane K. 2012 Stronger Lempel–Ziv based compressed text indexing. Algorithmica 62, 54–101. (doi:10.1007/s00453-010-9443-8)
8. Claude F, Navarro G. 2012 Improved grammar-based compressed indexes. In String Processing and Information Retrieval, Proc. 19th Int. Symp., SPIRE 2012, Cartagena de Indias, Colombia, 21–25 October (eds L. Calderón-Benavides, C. González-Caro, E. Chávez, N. Ziviani). Lecture Notes in Computer Science, vol. 7608, pp. 180–192. Berlin, Germany: Springer.
9. Do HH, Jansson J, Sadakane K, Sung W-K. 2012 Fast relative Lempel–Ziv self-index for similar sequences. In Frontiers in Algorithms and Algorithmic Aspects in Information and Management, Proc. 2nd Joint Int. Conf., FAW-AAIM 2012, Beijing, China, 14–16 May (eds J Snoeyink, P Lu, K Su, L Wang). Lecture Notes in Computer Science, vol. 7285, pp. 291–302. Berlin, Germany: Springer.
10. Gagie T, Gawrychowski P, Kärkkäinen J, Nekrich Y, Puglisi SJ. 2012 A faster grammar-based self-index. In Language and Automata Theory and Applications, Proc. 6th Int. Conf., LATA 2012, A Coruña, Spain, 5–9 March (eds A-H Dediú, C. Martín-Vide). Lecture Notes in Computer Science, vol. 7183, pp. 240–251. Berlin, Germany: Springer.
11. Kreft S, Navarro G. 2013 On compressing and indexing repetitive sequences. Theor. Comput. Sci. 483, 115–133. (doi:10.1016/j.tcs.2012.02.006)

12. Mäkinen V, Navarro G, Sirén J, Välimäki N. 2010 Storage and retrieval of highly repetitive sequence collections. J. Comput. Biol. 17, 281–308. (doi:10.1089/cmb.2009.0169)

13. Maruyama S, Nakahara M, Kishiue N, Sakamoto H. 2013 ESP-index: a compressed index based on edit-sensitive parsing. J. Discr. Algorithms 18, 100–112. (doi:10.1016/j.jda.2012.07.009)

14. Russo LMS, Navarro G, Oliveira AL, Morales P. 2009 Approximate string matching with compressed indexes. Algorithms 2, 1105–1136. (doi:10.3390/a2031105)

15. Kärkkäinen J, Ukkonen E. 1996 Lempel–Ziv parsing and sublinear-size index structures for string matching. In Proc. 3rd South American Workshop on String Processing, Recife, Brazil, 8–9 August (eds N Ziviani, R Baeza-Yates, K Guimarães). International Informatics Series, vol. 4, pp. 141–155. Ottawa, Canada: Carleton University Press.

16. Fischer J, Heun V. 2011 Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. 40, 465–492. (doi:10.1137/090779759)

17. Gagie T, Gawrychowski P, Puglisi SJ. 2012 Faster approximate pattern matching in compressed repetitive texts. (http://arxiv.org/abs/1109.2930)

18. Wandelt S, Starlinger J, Bux M, Leser U. 2013 RCSI: scalable similarity search in thousand(s) of genomes. Proc. VLDB Endowment 6, 1534–1545.