View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Research Database In Technology

Technical University of Denmark

Flexible indexing of repetitive collections

Belazzougui, Djamal; Cunial, Fabio; Gagie, Travis; Prezza, Nicola; Raffinot, Mathieu

Published in:
Unveiling Dynamics and Complexity - 13th Conference on Computability in Europe, CiE 2017, Proceedings

Link to article, DOI:
10.1007/978-3-319-58741-7_17

Publication date:
2017

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Belazzougui, D., Cunial, F., Gagie, T., Prezza, N., & Raffinot, M. (2017). Flexible indexing of repetitive collections. In Unveiling Dynamics and Complexity - 13th Conference on Computability in Europe, CiE 2017, Proceedings (Vol. 10307 LNCS, pp. 162-174). Springer Verlag. (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 10307 LNCS). DOI: 10.1007/978-3-319-58741-7_17

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Flexible indexing of repetitive collections

Djamal Belazzougui[1], Fabio Cunial[2], Travis Gagie[3], Nicola Prezza[4], and Mathieu Raffinot[5]

[1] DTISI-CERIST, Algiers, Algeria. dbelazzougui@cerist.dz
[2] MPI-CBG, Dresden, Germany. cunial@mpi-cbg.de
[3] UDP and CeBiB, Santiago, Chile. travis.gagie@mail.udp.cl
[4] DTU, Copenhagen, Denmark. npre@dtu.dk
[5] CNRS, Bordeaux, France. mathieu.raffinot@u-bordeaux.fr

Abstract. Highly repetitive strings are increasingly being amassed by genome sequencing experiments, and by versioned archives of source code and webpages. We describe practical data structures that support counting and locating all the exact occurrences of a pattern in a repetitive text, by combining the run-length encoded Burrows-Wheeler transform (RLBWT) with the boundaries of Lempel-Ziv 77 factors. One such variant uses an amount of space comparable to LZ77 indexes, but it answers count queries between two and four orders of magnitude faster than all LZ77 and hybrid index implementations, at the cost of slower locate queries. Combining the RLBWT with the compact directed acyclic word graph answers locate queries for short patterns between four and ten times faster than a version of the run-length compressed suffix array (RLCSA) that uses comparable memory, and with very short patterns our index achieves speedups even greater than ten with respect to RLCSA.

1 Introduction

Locating and counting all the exact occurrences of a pattern in a massive, highly repetitive collection of similar texts is a fundamental primitive in the post-genome era, in which genomes from multiple related species, from multiple strains of the same species, or from multiple individuals, are being sequenced at an increasing pace. Most data structures designed for such repetitive collections take space proportional to a specific measure of repetition, for example the number z of factors in a Lempel-Ziv parsing [1, 15], or the number r of runs in a Burrows-Wheeler transform [17]. In previous work we achieved competitive theoretical tradeoffs between space and time in locate queries, by combining data structures that depend on multiple measures of repetition that all grow sublinearly in the length of a repetitive string [3]. Specifically, we described a data structure that takes approximately $O(z + r)$ words of space, and that reports all the occurrences of a pattern of length m in a text of length n in $O(m(\log \log n + \log z) + \texttt{pocc} \cdot \log^{\epsilon} z + \texttt{socc} \cdot \log \log n)$ time, where pocc and socc are the number of primary and of secondary occurrences, respectively (defined in Section 2). This compares favorably to the reporting time of Lempel-Ziv

77 (LZ77) indexes [15], and to the space of solutions based on the run-length encoded Burrows-Wheeler transform (RLBWT) and on suffix array samples [17]. We also introduced a data structure whose size depends on the number of right-extensions of maximal repeats, and that reports all the occ occurrences of a pattern in $O(m \log\log n + \mathsf{occ})$ time. The main component of our constructions is the RLBWT, which we use for counting the number of occurrences of a pattern, and which we combine with the compact directed acyclic word graph, and with data structures from LZ indexes, rather than with suffix array samples, for answering locate queries. In this paper we describe and implement a range of practical variants of such theoretical approaches, and we compare their space-time tradeoffs to a representative set of state-of-the-art indexes for repetitive collections.

2 Preliminaries

Let $\Sigma = [1..\sigma]$ be an integer alphabet, let $\# = 0 \notin \Sigma$ be a separator, and let $T \in [1..\sigma]^{n-1}$ be a string. We denote by $\overline{T}$ the reverse of T, and by $\mathcal{P}_{T\#}(W)$ the set of all starting positions of a string $W \in [0..\sigma]^+$ in the circular version of $T\#$. We set $\Sigma^r_{T\#}(W) = \{a \in [0..\sigma] : |\mathcal{P}_{T\#}(Wa)| > 0\}$ and $\Sigma^\ell_{T\#}(W) = \{a \in [0..\sigma] : |\mathcal{P}_{T\#}(aW)| > 0\}$. A *repeat* $W \in \Sigma^+$ is a string with $|\mathcal{P}_{T\#}(W)| > 1$. A repeat W is *right-maximal* (respectively, *left-maximal*) iff $|\Sigma^r_{T\#}(W)| > 1$ (respectively, iff $|\Sigma^\ell_{T\#}(W)| > 1$). A *maximal repeat* is a repeat that is both left- and right-maximal. We say that a maximal repeat W is *rightmost* (respectively, *leftmost*) if no string WV with $V \in [0..\sigma]^+$ is left-maximal (respectively, if no string VW with $V \in [0..\sigma]^+$ is right-maximal).

For reasons of space we assume the reader to be familiar with the notion of *suffix tree* $\mathsf{ST}_{T\#} = (V, E)$ of $T\#$, i.e. the compact trie of all suffixes of $T\#$ (see e.g. [12] for an introduction). We denote by $\ell(\gamma)$, or equivalently by $\ell(u, v)$, the label of edge $\gamma = (u, v) \in E$, and we denote by $\ell(v)$ the concatenation of all edge labels in the path from the root to node $v \in V$. It is well known that a string W is right-maximal (respectively, left-maximal) in $T\#$ iff $W = \ell(v)$ for some internal node v of $\mathsf{ST}_{T\#}$ (respectively, iff $W = \overline{\ell(v)}$ for some internal node v of $\mathsf{ST}_{\overline{T}\#}$). Since left-maximality is closed under prefix operation, there is a bijection between the set of all maximal repeats of $T\#$ and the set of all nodes of the suffix tree of $T\#$ that lie on paths that start from the root and that end at nodes labelled by rightmost maximal repeats (a symmetrical observation holds for the suffix tree of $\overline{T}\#$).

The *compact directed acyclic word graph* of $T\#$ (denoted by $\mathsf{CDAWG}_{T\#}$ in what follows) is the minimal compact automaton that recognizes the set of suffixes of $T\#$ [4, 7]. It can be seen as a minimization of $\mathsf{ST}_{T\#}$ in which all leaves are merged to the same node (the sink) that represents $T\#$ itself, and in which all nodes except the sink are in one-to-one correspondence with the maximal repeats of $T\#$ [20] (the source corresponds to the empty string). As in the suffix tree, transitions are labelled by substrings of $T\#$, and the subgraph of $\mathsf{ST}_{T\#}$ induced by maximal repeats is isomorphic to a spanning tree of $\mathsf{CDAWG}_{T\#}$.

For reasons of space we assume the reader to be familiar with the notion and uses of the Burrows-Wheeler transform (BWT) of T and of the FM index, including the C array, LF mapping, and backward search (see e.g. [9]). In this paper we use $\mathsf{BWT}_{T\#}$ to denote the BWT of $T\#$, and we use $\mathtt{range}(W) = [\mathtt{sp}(W)..\mathtt{ep}(W)]$ to denote the lexicographic interval of a string W in a BWT that is implicit from the context. We say that $\mathsf{BWT}_{T\#}[i..j]$ is a *run* iff $\mathsf{BWT}_{T\#}[k] = c \in [0..\sigma]$ for all $k \in [i..j]$, and moreover if every substring $\mathsf{BWT}_{T\#}[i'..j']$ such that $i' \leq i$, $j' \geq j$, and either $i' \neq i$ or $j' \neq j$, contains at least two distinct characters. We denote by $r_{T\#}$ the number of runs in $\mathsf{BWT}_{T\#}$, and we call *run-length encoded BWT* (denoted by $\mathsf{RLBWT}_{T\#}$) any representation of $\mathsf{BWT}_{T\#}$ that takes $O(r_{T\#})$ words of space, and that supports rank and select operations (see e.g. [16, 17, 21]). Since the difference between $r_{T\#}$ and $r_{\overline{T}\#}$ is negligible in practice, we denote both of them by r when T is implicit from the context.

Repetition-aware string indexes. The *run-length compressed suffix array* of $T\#$, denoted by $\mathsf{RLCSA}_{T\#}$ in what follows, consists of a run-length compressed rank data structure for $\mathsf{BWT}_{T\#}$, and of a sampled suffix array, denoted by $\mathsf{SSA}_{T\#}$ [17]. The average time for locating an occurrence is inversely proportional to the size of $\mathsf{SSA}_{T\#}$, and fast locating needs a large SSA regardless of the compressibility of the dataset. Mäkinen et al. suggested ways to reduce the size of the SSA [17], but they did not perform well enough in real repetitive datasets for the authors to include them in the software they released.

The *Lempel-Ziv 77 factorization* of T [24], abbreviated with LZ77 in what follows, is the greedy decomposition of T into *phrases* or *factors* $T_1 T_2 \cdots T_z$ defined as follows. Assume that T is virtually preceded by the set of distinct characters in its alphabet, and assume that $T_1 T_2 \cdots T_i$ has already been computed for some prefix of length k of T: then, T_{i+1} is the longest prefix of $T[k+1..n]$ such that there is a $j \leq k$ that satisfies $T[j..j+|T_{i+1}|-1] = T_{i+1}$. For reasons of space we assume the reader to be familiar with LZ77 indexes: see e.g. [10, 13]. Here we just recall that a *primary occurrence* of a pattern P in T is one that crosses or ends at a phrase boundary in the LZ77 factorization $T_1 T_2 \cdots T_z$ of T. All other occurrences are called *secondary*. Once we have computed primary occurrences, locating all $\mathtt{socc}$ secondary occurrences reduces to two-sided range reporting, and it takes $O(\mathtt{socc} \cdot \log\log n)$ time with a data structure of $O(z)$ words of space [13]. To locate primary occurrences, we use a data structure for four-sided range reporting on a $z \times z$ grid, with a marker at (x, y) if the x-th LZ factor in lexicographic order is preceded in the text by the lexicographically y-th reversed prefix ending at a phrase boundary. This data structure takes $O(z)$ words of space, and it returns all the phrase boundaries that are immediately followed by a factor in the specified range, and immediately preceded by a reversed prefix in the specified range, in $O((1+k)\log^{\epsilon} z)$ time, where k is the number of phrase boundaries reported [5]. Kärkkäinen and Ukkonen used two PATRICIA trees [18], one for the factors and the other for the reversed prefixes ending at phrase boundaries [13]. Their approach takes $O(m^2)$ total time if T is not compressed. Replacing the uncompressed text by an augmented compressed representation, we can store T in $O(z \log n)$ space such that later, given P, we can find all $\mathtt{occ}$

occurrences of P in $O(m \log m + \mathsf{occ} \cdot \log\log n)$ time [10].

Alternatively, if all queried patterns are of length at most M, we could store in a FM index the substrings of T that consist of characters within distance M from the closest phrase boundary, and use that to find primary occurrences (see e.g. [22] and references therein). This approach is known as *hybrid indexing*.

Composite repetition-aware string indexes. Combining $\mathsf{RLBWT}_{T\#}$ with the set of all starting positions $p_1, p_2, \ldots, p_z$ of the LZ factors of T, yields a data structure that takes $O(z + r)$ words of space, and that reports all the $\mathtt{pocc}$ primary occurrences of a pattern $P \in [1..\sigma]^m$ in $O(m(\log\log n + \log z) + \mathtt{pocc} \cdot \log^{\epsilon} z)$ time [3]. Since such data structure is at the core of this paper, we summarize it in what follows. The same primary occurrence of P in T can cover up to m factor boundaries. Thus, we consider every possible way of placing, inside P, the rightmost boundary between two factors, i.e. every possible split of P in two parts $P[1..k-1]$ and $P[k..m]$ for $k \in [2..m]$, such that $P[k..m]$ is either a factor or a proper prefix of a factor. For every such k, we use four-sided range reporting queries to list all the occurrences of P in T that conform to the split, as described before. We encode the sequence $p_1, p_2, \ldots, p_z$ implicitly, as follows: we use a bitvector $\mathtt{last}[1..n]$ such that $\mathtt{last}[i] = 1$ iff $\mathsf{SA}_{\overline{T}\#}[i] = n - p_j + 2$ for some $j \in [1..z]$, i.e. iff $\mathsf{SA}_{\overline{T}\#}[i]$ is the last position of a factor. We represent such bitvector as a predecessor data structure with partial ranks, using $O(z)$ words of space [23]. Let $\mathsf{ST}_{T\#} = (V, E)$ be the suffix tree of $T\#$, and let $V' = \{v_1, v_2, \ldots, v_z\} \subseteq V$ be the set of loci in $\mathsf{ST}_{T\#}$ of all the LZ factors of T. Consider the list of node labels $L = \ell(v_1), \ell(v_2), \ldots, \ell(v_z)$, sorted in lexicographic order. It is easy to build a data structure that takes $O(z)$ words of space, and that implements in $O(\log z)$ time function $\mathbb{I}(W, V')$, which returns the (possibly empty) interval of W in L (see e.g. [3]). Together with $\mathtt{last}$, $\mathsf{RLBWT}_{T\#}$ and $\mathsf{RLBWT}_{\overline{T}\#}$, this data structure is the output of our construction.

Given P, we first perform a backward search in $\mathsf{RLBWT}_{T\#}$ to determine the number of occurrences of P in $T\#$: if this number is zero, we stop. During backward search, we store in a table the interval $[i_k..j_k]$ of $P[k..m]$ in $\mathsf{BWT}_{T\#}$ for every $k \in [2..m]$. Then, we compute the interval $[i'_{k-1}..j'_{k-1}]$ of $\overline{P[1..k-1]}$ in $\mathsf{BWT}_{\overline{T}\#}$ for every $k \in [2..m]$, using backward search in $\mathsf{RLBWT}_{\overline{T}\#}$: if $\mathtt{rank}_1(\mathtt{last}, j'_{k-1}) - \mathtt{rank}_1(\mathtt{last}, i'_{k-1} - 1) = 0$, then $P[1..k-1]$ never ends at the last position of a factor, and we can discard this value of k. Otherwise, we convert $[i'_{k-1}..j'_{k-1}]$ to the interval $[\mathtt{rank}_1(\mathtt{last}, i'_{k-1} - 1) + 1..\mathtt{rank}_1(\mathtt{last}, j'_{k-1})]$ of all the reversed prefixes of T that end at the last position of a factor. Rank operations on $\mathtt{last}$ can be implemented in $O(\log\log n)$ time using predecessor queries. We get the lexicographic interval of $P[k..m]$ in the list of all distinct factors of T, in $O(\log z)$ time, using operation $\mathbb{I}(P[k..m], V')$. We use such intervals to query the four-sided range reporting data structure.

It is also possible to combine $\mathsf{RLBWT}_{T\#}$ with $\mathsf{CDAWG}_{T\#}$, building a data structure that takes $O(e_{T\#})$ words of space, and that reports all the $\mathtt{occ}$ occurrences of P in $O(m \log\log n + \mathtt{occ})$ time, where $e_{T\#}$ is the number of right-extensions of maximal repeats of $T\#$ [3]. Specifically, for every node v in the CDAWG, we store $|\ell(v)|$ in a variable $v.\mathtt{length}$. Recall that an arc (v, w) in

the CDAWG means that maximal repeat $\ell(w)$ can be obtained by extending maximal repeat $\ell(v)$ to the right *and to the left*. Thus, for every arc $\gamma = (v, w)$ of the CDAWG, we store the first character of $\ell(\gamma)$ in a variable γ.`char`, and we store the length of the right extension implied by γ in a variable γ.`right`. The length γ.`left` of the left extension implied by γ can be computed by w.`length` $-$ v.`length` $-$ γ.`right`. For every arc of the CDAWG that connects a maximal repeat W to the sink, we store just γ.`char` and the starting position γ.`pos` of string $W \cdot \gamma$.`char` in T. The total space used by the CDAWG is $O(e_{T\#})$ words, and the number of runs in $\mathsf{BWT}_{T\#}$ can be shown to be $O(e_{T\#})$ as well [3] (an alternative construction could use $\mathsf{CDAWG}_{\overline{T}\#}$ and $\mathsf{RLBWT}_{\overline{T}\#}$).

Once again, we use the RLBWT to count the number of occurrences of P in T in $O(m \log\log n)$ time: if this number is not zero, we use the CDAWG to report all the `occ` occurrences of P in $O(\texttt{occ})$ time, using a technique already sketched in [6]. Specifically, since we know that P occurs in T, we perform a blind search for P in the CDAWG, as is typically done with PATRICIA trees. We keep a variable i, initialized to zero, that stores the length of the prefix of P that we have matched so far, and we keep a variable j, initialized to one, that stores the starting position of P inside the last maximal repeat encountered during the search. For every node v in the CDAWG, we choose the arc γ such that γ.`char` $= P[i+1]$ in constant time using hashing, we increment i by γ.`right`, and we increment j by γ.`left`. If the search leads to the sink by an arc γ, we report γ.`pos`$+j$ and we stop. If the search ends at a node v that is associated with a maximal repeat W, we determine all the occurrences of W in T by performing a depth-first traversal of all nodes reachable from v in the CDAWG , updating variables i and j as described before, and reporting γ.`pos` $+ j$ for every arc γ that leads to the sink. The total number of nodes and arcs reachable from v is $O(\texttt{occ})$.

3 Combining RLBWT and LZ factors in practice

In this paper we implement[6] a range of practical variants of the combination of RLBWT and LZ factorization described in Section 2. Specifically, in addition to the version described in Section 2 (which we call *full* in what follows), we design a variant in which we drop $\mathsf{RLBWT}_{\overline{T}\#}$, simulating it with a bidirectional index, in order to save space (we call this *bidirectional* in what follows); a variant in which we drop $\mathsf{RLBWT}_{\overline{T}\#}$, the four-sided range reporting data structure, and the subset of suffix tree nodes, in order to save even more space (we call this variant *light* in what follows); and another variant in which, to reduce space even further, we use a *sparse* version of the LZ parsing, i.e. we skip a fixed number of characters after each factor (we call this index *sparse* in what follows). In addition, we design a number of optimizations to speed up locate queries in practice: we will describe them in the full version of the paper.

Our representation of the RLBWT is based on the one described in [21], which

[6] Our source code is available at `https://github.com/nicolaprezza/lz-rlbwt` and `https://github.com/nicolaprezza/slz-rlbwt` and it is based on SDSL [11].

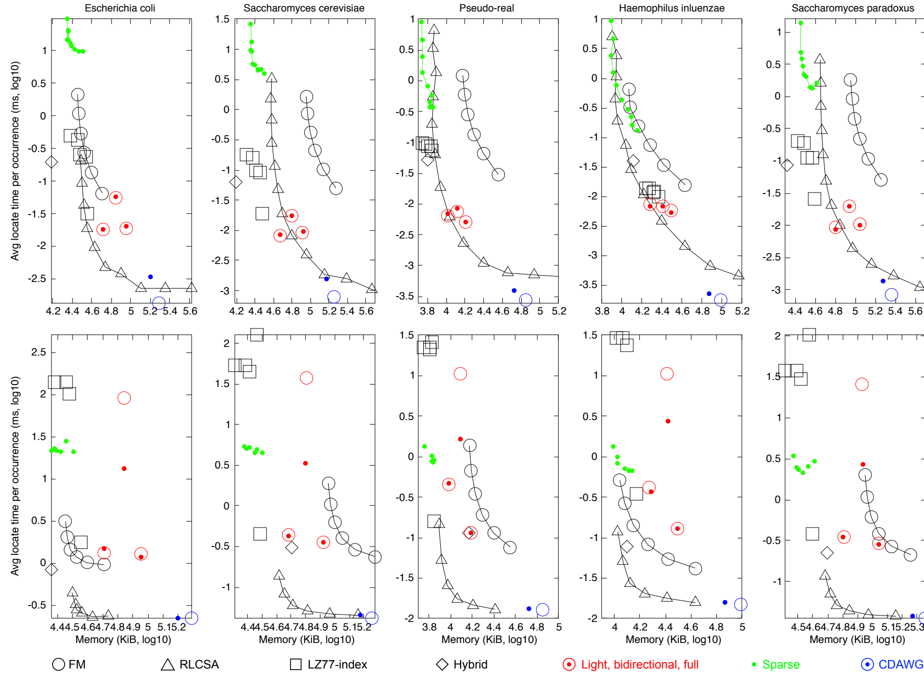

Fig. 1: Locate queries: space-time tradeoffs of our indexes (color) and of the state of the art (black). Top row: patterns of length 16. Bottom row: patterns of length 512. The full, bidirectional, and light indexes are shown with (red dots) and without (red circles) speed optimizations. The CDAWG is shown in succinct (blue dots) and non-succinct (blue circles) version.

we summarize here for completeness, but is more space-efficient. The authors of [21] store one character per run in a string $H \in \Sigma^r$, they mark with a one the beginning of each run in a bitvector $V_{all}[0..n-1]$, and for every $c \in \Sigma$ they store the lengths of all runs of character c consecutively in a bit-vector V_c: specifically, every c-run of length k is represented in V_c as $\mathtt{10}^{k-1}$. This representation allows one to map rank and access queries on $\mathsf{BWT}_{T\#}$ to rank, select and access queries on H, V_{all}, and V_c. By gap-encoding the bitvectors, this representation takes $r(2\log(n/r) + \log\sigma)(1 + o(1))$ bits of space. We reduce the multiplicative factor of the term $\log(n/r)$ by storing in V_{all} just one out of $1/\epsilon$ ones, where $0 < \epsilon \leq 1$ is a given constant (we set $\epsilon = 1/8$ in all our experiments). Note that we are still able to answer all queries on the RLBWT, by using the V_c vectors to reconstruct the positions of the missing ones in V_{all}. However, query time gets multiplied by $1/\epsilon$. We represent H as a Huffman-encoded string (`wt_huff<>` in SDSL), and we gap-encode bitvectors with Elias-Fano (`sd_vector<>` in SDSL).

Full index. Our first variant is an engineered version of the data structure described in Section 2. We store both $\mathsf{RLBWT}_{T\#}$ and $\mathsf{RLBWT}_{\overline{T\#}}$. A gap-encoded bitvector $\mathtt{end}[0..n-1]$ of $z\log(n/z)(1 + o(1))$ bits marks the rank, among all the

suffixes of $\overline{T}\#$, of every suffix $\overline{T}[i..n-1]\#$ such that $n-i-2$ is the last position of an LZ factor of T. Symmetrically, a gap-encoded bitvector $\texttt{begin}[0..n-1]$ of $z\log(n/z)(1+o(1))$ bits marks the rank, among all the suffixes of $T\#$, of every suffix $T[i..n-1]\#$ such that i is the first position of an LZ factor of T.

Geometric range data structures are implemented with wavelet trees (`wt_int` in SDSL). We manage to fit the 4-sided data structure in $2z\log z(1+o(1))$ bits, and the 2-sided data structure in $z(2\log n+1)(1+o(1))$ bits: we will detail such implementations in the full version of the paper. Finally, we need a way to compute the lexicographic range of a string among all the LZ factors of T. We implement a simpler and more space-efficient strategy than the one proposed in [3], which we will describe in the full version of the paper. In summary, the full index takes $\big(6z\log n+2(1+\epsilon)r\log(n/r)+2r\log\sigma\big)\cdot(1+o(1))$ bits of space, and it supports count queries in $O(m\cdot(\log(n/r)+\log\sigma))$ time and locate queries in $O((m+\mathsf{occ})\cdot\log n)$ time.

Bidirectional index. To save space we can drop $\mathsf{RLBWT}_{\overline{T}\#}$ and simulate it using just $\mathsf{RLBWT}_{T\#}$, by applying the synchronization step performed in bidirectional BWT indexes (see e.g. [2] and references therein). This strategy penalizes the time complexity of locate queries, which becomes quadratic in the length of the pattern. Moreover, since in our implementation we store run-lengths separately for each character, a synchronization step requires σ rank queries to find the number of characters smaller than a given character inside a BWT interval. This operation could be performed in $O(\log\sigma)$ time if the string were represented as a wavelet tree. In summary, the bidirectional variant of the index takes $\big(6z\log n+(1+\epsilon)r\log(n/r)+r\log\sigma\big)\cdot(1+o(1))$ bits of space, it supports count queries in $O(m\cdot(\log(n/r)+\log\sigma))$ time, and it supports locate queries in $O(m^2\sigma\log(n/r)+(m+\mathsf{occ})\cdot\log n)$ time.

Light index. Once we have computed the interval of the pattern in $\mathsf{BWT}_{T\#}$, we can locate all its primary occurrences by just forward-extracting at most m characters for each occurrence inside the range: this is because every primary occurrence of the pattern overlaps with the last position of an LZ factor. We implement forward extraction by using select queries on $\mathsf{RLBWT}_{T\#}$. This approach requires just $\mathsf{RLBWT}_{T\#}$, the 2-sided range data structure, a gap-encoded bitvector $\texttt{end}_T$ that marks the last position of every LZ factor in the text, a gap-encoded bitvector $\texttt{end}_{BWT}$ that marks the last position of every LZ factor in $\mathsf{BWT}_{T\#}$, and z integers of $\log z$ bits each, connecting corresponding ones in $\texttt{end}_{BWT}$ and in $\texttt{end}_T$: this array plays the role of the sparse suffix array sampling in RLCSA.

Sparse index. We can reduce the size of the index even further by *sparsifying the LZ factorization.* Intuitively, the factorization of a highly-repetitive collection of strings $T=T_1T_2\cdots T_k$, where $T_2,\ldots,T_k$ are similar to T_1, is much denser inside T_1 than it is inside $T_2\cdots T_k$. Thus, excluding long enough contiguous regions from the factorization (i.e. not outputting factors inside such regions) could reduce the number of factors in dense regions. Formally, let $d>0$, and consider the following generalization of LZ77, denoted here by LZ77-d: we factor T as $X_1Y_1X_2Y_2\cdots X_{z_d}Y_{z_d}$, where z_d is the size of the factorization, $Y_i\in\Sigma^d$

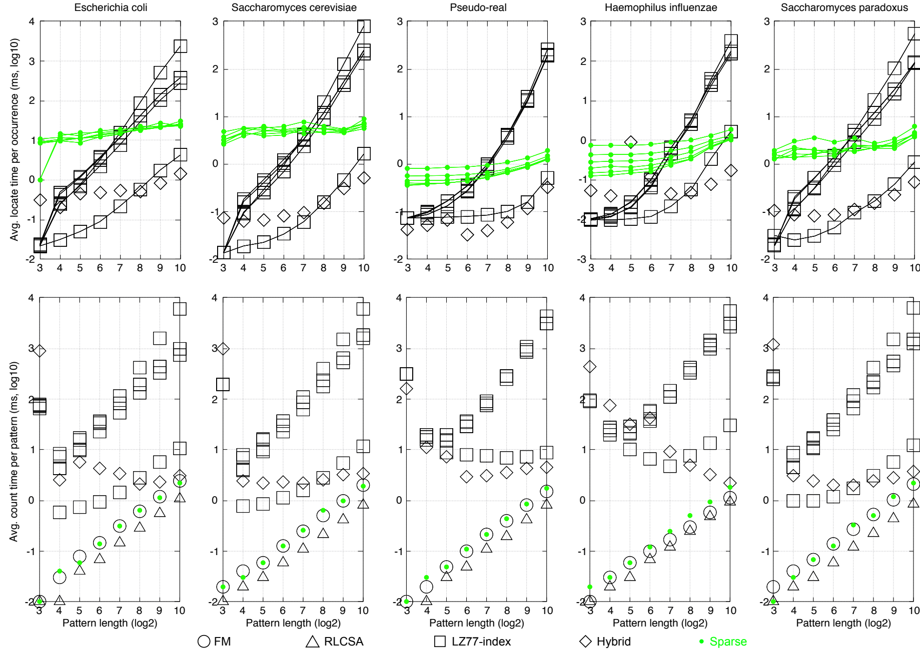

Fig. 2: Locate time per occurrence (top) and count time per pattern (bottom), as a function of pattern length, for the sparse index with skip rate 2^i, $i \in [5..10]$, the LZ77 index, and the hybrid index. Count plots show also the FM index and RLCSA.

for all $i \in [1..z_d]$, and X_i is the longest prefix of $X_i Y_i \cdots X_{z_d} Y_{z_d}$ that starts at least once inside the range of positions $[1..|X_1 Y_1 \cdots X_{i-1} Y_{i-1}|]$. To make the light index work with LZ77-d, we need to sample the suffix array of $T\#$ at the lexicographic ranks that correspond to the last position of every X_i, and we need to redefine primary occurrences as those that are not fully contained inside an X factor. To answer a locate query, we also need to extract d additional characters before each occurrence of the pattern, in order to detect primary occurrences that start inside a Y factor. Finally, the 2-sided range data structure needs to be built on the sources of the X factors. The sparse index takes $\big(z_d(3\log n + \log(n/z_d)) + (1+\epsilon)r\log(n/r)\big) \cdot (1+o(1))$ bits of space, it answers locate queries in $O((\mathtt{occ}+1)\cdot(m+d)\cdot\log n)$ time, and count queries in $O(m(\log(n/r)+\log\sigma))$ time. Setting d large enough makes z_d up to three times smaller than the number of LZ factors in realistic highly-repetitive collections.

4 Combining RLBWT and CDAWG in practice

In this paper we also engineer[7] the combination of RLBWT and CDAWG described in Section 2, and in particular we study the effects of two representations

[7] Our source code is available at `https://github.com/mathieuraffinot/locate-cdawg`

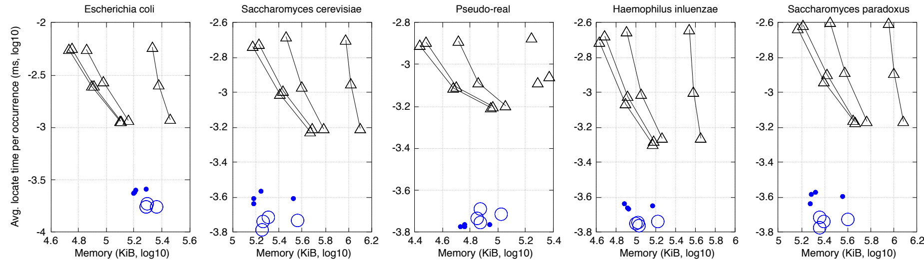

Fig. 3: Space-time traeoffs of the CDAWG (blue) compared to RLCSA (triangles) with sampling rate 2^i, $i \in [3..5]$. Patterns of length 8, 6, 4, 2 (from left to right). The CDAWG is shown in succinct (blue dots) and non-succinct (blue circles) version.

of the CDAWG. In the first one, the graph is encoded as a sequence of variable-length integers: every integer is represented as a sequence of bytes, in which the seven least significant bits of every byte are used to encode the integer, and the most significant bit flags the last byte of the integer. Nodes are stored in the sequence according to their topological order in the graph obtained from the CDAWG by inverting the direction of all arcs: to encode a pointer from a node v to its successor w in the CDAWG, we store the difference between the first byte of v and the first byte of w in the sequence. If w is the sink, such difference is replaced by a shorter code. We choose to store the length of the maximal repeat that corresponds to each node, rather than the offset of $\ell(v)$ inside $\ell(w)$ for every arc (v, w), since such lengths are short and their number is smaller than the number of arcs in practice.

In the second encoding we exploit the fact that the subgraph of the suffix tree of $T\#$ induced by maximal repeats is a spanning tree of $\mathsf{CDAWG}_{T\#}$. Specifically, we encode such spanning tree with the balanced parenthesis scheme described in [19], and we resolve the arcs of the CDAWG that belong to the tree using corresponding tree operations. Such operations work on node identifiers, thus we need to convert a node identifier to the corresponding first byte in the byte sequence of the CDAWG, and vice versa. We implement such translation by encoding the monotone sequence of the first byte of every node with the quasi-succinct representation by Elias and Fano, which uses at most $2 + \log(N/n)$ bits per starting position, where N is the number of bytes in the byte sequence and n is the number of nodes [8].

5 Experimental results

We test our implementations on five DNA datasets from the Pizza&Chili repetitive corpus[8], which include the whole genomes of approximately 36 strains of the same eukaryotic species, a collection of 23 and approximately 78 thousand substrings of the genome of the same bacterium, and an artificially repetitive

[8] http://pizzachili.dcc.uchile.cl/repcorpus.html

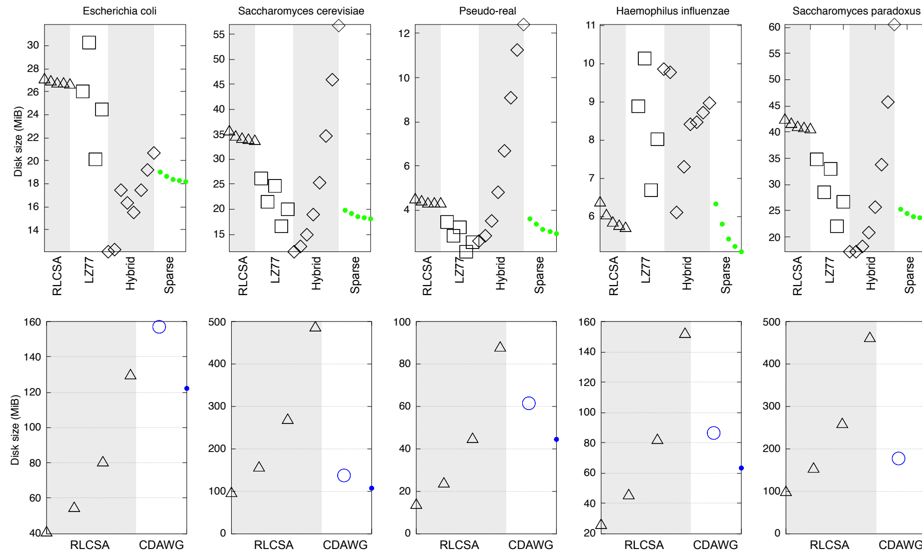

Fig. 4: (Top) Disk size of the sparse index with skip rate 2^i, $i \in [10..15]$, compared to the hybrid index with maximum pattern length 2^i, $i \in [3..10]$, the LZ77 index, and RLCSA with sampling rate 2^i, $i \in [10..15]$. (Bottom) Disk size of the CDAWG compared to RLCSA with sampling rate 2^i, $i \in [2..5]$. The CDAWG is shown in succinct (blue dots) and non-succinct (blue circles) version.

string obtained by concatenating 100 mutated copies of the same substring of the human genome. We compare our results to the FM index implementation in SDSL [11] with sampling rate 2^i for $i \in [5..10]$, to an implementation of RLCSA[9] with the same sampling rates, to the five variants in the implementation of the LZ77 index described in [14], and to a recent implementation of the compressed hybrid index [22]. The FM index uses RRR bitvectors in its wavelet tree. For brevity, we call LZ1 the implementation of the LZ77 index that uses the suffix trie and the reverse trie. For each process, and for each pattern length 2^i for $i \in [3..10]$, we measure the maximum resident set size and the number of CPU seconds that the process spends in user mode[10], both for locate and for count queries, discarding the time for loading the indexes and averaging our measurements over one thousand patterns[11]. We experiment with skipping 2^i characters

[9] We compile the sequential version of `https://github.com/adamnovak/rlcsa` with `PSI_FLAGS` and `SA_FLAGS` turned off (in other words, we use a gap-encoded bitvector rather than a succinct bitvector to mark sampled positions in the suffix array). The block size of psi vectors (`RLCSA_BLOCK_SIZE`) is 32 bytes.

[10] We perform all experiments on a single core of a 6-core, 2.50 GHz, Intel Xeon E5-2640 processor, with access to 128GiB of RAM and running CentOS 6.3. We measure resources with GNU Time 1.7, and we compile with GCC 5.3.0.

[11] We use as patterns random substrings of each dataset, containing just DNA bases, generated with the `genpatterns` tool from the Pizza&Chili repetitive corpus.

before opening a new phrase in the sparse index, where $i \in [5..10]$.

The first key result of our experiments is that, in highly-repetitive strings, the sparse index takes an amount of space that is comparable to LZ indexes, and thus typically smaller than the space taken by RLCSA and by the FM index, while supporting count operations that are approximately as fast as RLCSA and as the FM index, and thus typically faster than LZ indexes. This new tradeoff comes at the cost of slower locate queries.

Specifically, the gap between sparse index and LZ variants in the running time of count queries is large for short patterns: the sparse index is between two and four orders of magnitude faster than all variants of the LZ index, with the largest difference achieved by patterns of length 8 (Figure 2, bottom). The difference between the sparse index and variant LZ1 shrinks as pattern length increases. Locate queries are between one and three orders of magnitude slower in the sparse index than in LZ indexes, and comparable to RLCSA with sampling rates equal to or greater than 2048 (Figure 1, top). However, for patterns of length approximately 64 or larger, the sparse index becomes between one and two orders of magnitude *faster* than all variants of the LZ index, except LZ1. As a function of pattern length, the running time per occurrence of the sparse index grows more slowly than the running time of LZ1, suggesting that the sparse index might even approach LZ1 for patterns of length between 1024 and 2048 (Figure 2, top). Compared to the hybrid index, the sparse index is again orders of magnitude faster in count queries, especially for short patterns (Figure 2, bottom). As with LZ1, the difference shrinks as pattern length increases, but since the size of the hybrid index depends on maximum pattern length, the hybrid index becomes larger than the sparse index for patterns of length between 64 and 128, and possibly even shorter (Figure 4, top). As with LZ indexes, faster count queries come at the expense of locate queries, which are approximately 1.5 orders of magnitude slower in the sparse index than in the hybrid index (Figure 2, top).

The second key result of our experiments is that the CDAWG is efficient at locating very short patterns, and in this regime it achieves the smallest query time among all indexes. Specifically, the running time per occurrence of the CDAWG is between 4 and 10 times smaller than the running time per occurrence of a version of RLCSA that uses comparable memory, and with patterns of length two the CDAWG achieves speedups even greater than 10 (Figure 3). Note that short exact patterns are a frequent use case when searching large repetitive collections of versioned source code. The CDAWG does not achieve any new useful tradeoff with long patterns. Using the succinct representation of the CDAWG saves between 20% and 30% of the disk size and resident set size of the non-succinct representation, but using the non-succinct representation saves between 20% and 80% of the query time of the succinct representation, depending on dataset and pattern length. Finally, our full, bidirectional and light index implementations exhibit the same performance as the sparse index for count queries, but it turns out that they take too much space in practice to achieve any new useful tradeoff (Figure 1).

References

1. Arroyuelo, D., Navarro, G., Sadakane, K.: Stronger Lempel-Ziv based compressed text indexing. Algorithmica 62, 54–101 (2012).
2. Belazzougui, D.: Linear time construction of compressed text indices in compact space. In: Proc. STOC. pp. 148–193 (2014).
3. Belazzougui, D., et al.: Composite repetition-aware data structures. In: Proc. CPM. pp. 26–39 (2015).
4. Blumer, A., et al.: Complete inverted files for efficient text retrieval and analysis. JACM 34, 578–595 (1987).
5. Chan, T.M., Larsen, K.G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: Proc. SoCG. pp. 1–10 (2011).
6. Crochemore, M., Hancart, C.: Automata for matching patterns. In: Handbook of formal languages, pp. 399–462. Springer (1997).
7. Crochemore, M., Vérin, R.: Direct construction of compact directed acyclic word graphs. In: Proc. CPM. pp. 116–129 (1997).
8. Elias, P., Flower, R.A.: The complexity of some simple retrieval problems. JACM 22, 367–379 (1975).
9. Ferragina, P., Manzini, G.: Indexing compressed texts. JACM 52(4), 552–581 (2005).
10. Gagie, T., et al.: LZ77-based self-indexing with faster pattern matching. In: Proc. LATIN. pp. 731–742 (2014).
11. Gog, S., et al.: From theory to practice: Plug and play with succinct data structures. In: Proc. SEA. pp. 326–337 (2014).
12. Gusfield, D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press (1997).
13. Kärkkäinen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string matching. In: Proc. WSP. pp. 141–155 (1996).
14. Kreft, S.: Self-index based on LZ77. Master's thesis, Department of Computer Science, University of Chile (2010).
15. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. TCS 483, 115–133 (2013).
16. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. In: Proc. CPM. pp. 45–56 (2005).
17. Mäkinen, V., et al.: Storage and retrieval of highly repetitive sequence collections. JCB 17, 281–308 (2010).
18. Morrison, D.R.: PATRICIA — practical algorithm to retrieve information coded in alphanumeric. JACM 15, 514–534 (1968).
19. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. 31, 762–776 (2002).
20. Raffinot, M.: On maximal repeats in strings. IPL 80, 165–169 (2001).
21. Sirén, J., et al.: Run-length compressed indexes are superior for highly repetitive sequence collections. In: Proc. SPIRE. pp. 164–175 (2008).
22. Valenzuela, D.: CHICO: A compressed hybrid index for repetitive collections. In: Proc. SEA (2016).
23. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\theta(n)$. IPL 17, 81–84 (1983).
24. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE TIT 23, 337–343 (1977).