Fully-Functional Suffix Trees and Optimal Text Searching in BWT-runs Bounded Space

Travis Gagie, Dalhousie University, Canada
Gonzalo Navarro, University of Chile, Chile
Nicola Prezza, University of Pisa, Italy

Indexing highly repetitive texts — such as genomic databases, software repositories and versioned text collections — has become an important problem since the turn of the millennium. A relevant compressibility measure for repetitive texts is r, the number of runs in their Burrows-Wheeler Transforms (BWTs). One of the earliest indexes for repetitive collections, the Run-Length FM-index, used $O(r)$ space and was able to efficiently count the number of occurrences of a pattern of length m in a text of length n (in $O(m \log\log n)$ time, with current techniques). However, it was unable to locate the positions of those occurrences efficiently within a space bounded in terms of r. In this paper we close this long-standing problem, showing how to extend the Run-Length FM-index so that it can locate the occ occurrences efficiently (in $O(occ \log\log n)$ time) within $O(r)$ space. By raising the space to $O(r \log\log n)$ our index counts the occurrences in optimal time, $O(m)$, and locates them in optimal time as well, $O(m + occ)$. By further raising the space by an $O(w/\log\sigma)$ factor, where σ is the alphabet size and $w = \Omega(\log n)$ is the RAM machine size in bits, we support count and locate in $O(\lceil m\log(\sigma)/w\rceil)$ and $O(\lceil m\log(\sigma)/w\rceil + occ)$ time, which is optimal in the packed setting and had not been obtained before in compressed space. We also describe a structure using $O(r\log(n/r))$ space that replaces the text and extracts any text substring of length ℓ in the almost-optimal time $O(\log(n/r) + \ell\log(\sigma)/w)$. Within that space, we similarly provide access to arbitrary suffix array, inverse suffix array, and longest common prefix array cells in time $O(\log(n/r))$, and extend these capabilities to full suffix tree functionality, typically in $O(\log(n/r))$ time per operation. Our experiments show that our $O(r)$-space index outperforms the space-competitive alternatives by 1–2 orders of magnitude in time. Competitive implementations of the original FM-index are outperformed by 1–2 orders of magnitude in space and/or 2–3 in time.

Additional Key Words and Phrases: Repetitive string collections; Compressed text indexes; Burrows-Wheeler Transform; Compressed suffix trees

1. INTRODUCTION

The data deluge has become a pervasive problem in most organizations that aim to collect and process data. We are concerned about string (or text, or sequence) data, formed by collections of symbol sequences. This includes natural language text collections, DNA and protein sequences, source code repositories, semistructured text, and many others. The rate at which those sequence collections are growing is daunting, in some cases outpacing Moore's Law by a significant margin [Sthephens et al. 2015]. A key to handle this growth is the fact that the amount of *unique* material does not grow at the same pace of the sequences. Indeed, the fastest-growing string collections are in many cases *highly repetitive*, that is, most of the strings can be obtained from others with a few modifications. For example, most genome sequence collections store many genomes from the same species, which in the case of, say, humans differ by 0.1% [Przeworski et al. 2000] (there is some discussion about the exact percentage). The 1000-genomes project[1] uses a Lempel-Ziv-like compression mechanism that reports

[1] http://www.internationalgenome.org

Gagie partially funded by Basal Funds FB0001 and Fondecyt Grant 1-171058, Conicyt, Chile. Navarro partially funded by Basal Funds FB0001 and Fondecyt Grant 1-170048, Conicyt, Chile. Prezza partially funded by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant no. RBSI146R5L.
An early partial version of this article appeared in *Proc. SODA 2018* [Gagie et al. 2018b].

Author's addresses: Travis Gagie, University of Dalhousie, Canada, travis.gagie@gmail.com. Gonzalo Navarro, CeBiB – Center for Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile, gnavarro@dcc.uchile.cl. Nicola Prezza, Department of Computer Science, University of Pisa, Italy, nicola.prezza@di.unipi.it.

DOI: http://dx.doi.org/10.1145/0000000.0000000

compression ratios around 1% [Fritz et al. 2011] (i.e., the compressed space is two orders of magnitude less than the uncompressed space). Versioned document collections and software repositories are another natural source of repetitiveness. For example, Wikipedia reports that, by June 2015, there were over 20 revisions (i.e., versions) per article in its 10 TB content, and that `p7zip`[2] compressed it to about 1%. They also report that what grows the fastest today are the revisions rather than the new articles, which increases repetitiveness.[3] A study of GitHub (which surpassed 20 TB in 2016)[4] reports a ratio of *commit* (new versions) over *create* (brand new projects) around 20.[5]

Version management systems offer a good solution to the problem of providing efficient *access* to the documents of a versioned collection, at least when the versioning structure is known. They factor out repetitiveness by storing the first version of a document in plain form and then the edits of each version of it. It is much more challenging, however, to provide more advanced functionalities, such as *counting* or *locating* the positions where a string pattern occurs across the collection.

An application field where this need is most pressing is bioinformatics. The *FM-index* [Ferragina and Manzini 2005; Ferragina et al. 2007] was extremely successful in reducing the size of classical data structures for pattern searching, such as suffix trees [Weiner 1973] or suffix arrays [Manber and Myers 1993], to the *statistical* entropy of the sequence while emulating a significant part of their functionality. The FM-index has had a surprising impact far beyond the boundaries of theoretical computer science: if someone now sends his or her genome to be analyzed, it will almost certainly be sequenced on a machine built by Illumina[6], which will produce a huge collection of quite short substrings of that genome, called reads. Those reads' closest matches will then be sought in a reference genome, to determine where they most likely came from in the newly-sequenced target genome, and finally a list of the likely differences between the target and the reference genomes will be reported. The searches in the reference genome will be done almost certainly using software such as `Bowtie`[7], `BWA`[8], or `Soap2`[9], all of them based on the FM-index.[10]

Genomic analysis is already an important field of research, and a rapidly growing industry [Schatz and Langmead 2013]. As a result of dramatic advances in sequencing technology, we now have datasets of tens of thousands of genomes (e.g., the 100,000-human-genomes project[11] was completed in December 2018). Unfortunately, current software based on FM-indexes cannot handle such massive datasets: they use 2 bits per base at the very least [Keel and Snelling 2018]. Even though the FM-index can represent the sequences within their statistical entropy [Ferragina et al. 2007], this measure is insensitive to the repetitiveness of those datasets [Kreft and Navarro 2013, Lem. 2.6], and thus the FM-indexes would grow proportionally to the sizes of the sequences. Using current tools, indexing a set of 100,000 human genomes would require 75 TB of storage at the very least, and the index would have to reside in main memory to operate efficiently. To handle such a challenge we need, instead, compressed text indexes *whose size is proportional to the amount of unique material* in those huge datasets.

1.1. Related work

Mäkinen and Navarro [2005] pioneered the research on indexing and searching repetitive string collections [Sirén et al. 2008; Mäkinen et al. 2009; Mäkinen et al. 2010]. They regard the collection as a single concatenated text $T[1\,..\,n]$ with separator symbols, and note that the number r of *runs* (i.e., maximal substrings formed by a single symbol) in the *Burrows-Wheeler Transform (BWT)* [Burrows and Wheeler 1994] of the text is relatively very low on repetitive texts. Their index, *Run-Length FM-Index (RLFM-index)*, uses $O(r)$ words of space and can count the number of occurrences of a pattern $P[1\,..\,m]$ in time $O(m \log n)$ and even less. However, they are unable to locate where those positions are in T

[2] `http://p7zip.sourceforge.net`

[3] `https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia`

[4] `https://blog.sourced.tech/post/tab_vs_spaces`

[5] `http://blog.coderstats.net/github/2013/event-types`, see the ratios of *push/create* and *commit/push*.

[6] `https://www.illumina.com`. Over 94% of the human genomes in SRA [Kodama et al. 2012] were sequenced by Illumina.

[7] `http://bowtie-bio.sourceforge.net`

[8] `http://bio-bwa.sourceforge.net`

[9] `http://soap.genomics.org.cn`

[10] Ben Langmead, personal communication.

[11] `https://www.genomicsengland.co.uk/the-100000-genomes-project`

unless they add a set of samples that require $\Theta(n/s)$ words in order to offer $O(s \log n)$ time to locate each occurrence. On repetitive texts, either this sampled structure is orders of magnitude larger than the $O(r)$-size basic index, or the locating time is extremely high.

Many proposals since then aimed at reducing the locating time by building on other compression methods that perform well on repetitive texts: indexes based on the Lempel-Ziv parse [Lempel and Ziv 1976] of T, with size bounded in terms of the number z of phrases [Kreft and Navarro 2013; Gagie et al. 2014; Nishimoto et al. 2015; Belazzougui et al. 2015a; Navarro 2017; Bille et al. 2018; Christiansen and Ettienne 2018]; indexes based on the smallest context-free grammar (or an approximation thereof) that generates T and only T [Kieffer and Yang 2000; Charikar et al. 2005], with size bounded in terms of the size g of the grammar [Claude and Navarro 2010; 2012; Gagie et al. 2012; Navarro 2019]; and indexes based on the size e of the smallest automaton (CDAWG) [Blumer et al. 1987] recognizing the substrings of T [Belazzougui et al. 2015a; Takagi et al. 2017; Belazzougui and Cunial 2017a]. Table I summarizes the pareto-optimal achievements. We do not consider in this paper indexes based on other repetitiveness measures that only apply in restricted scenarios, such as those based on Relative Lempel-Ziv [Kuruppu et al. 2010; Do et al. 2014; Belazzougui et al. 2014; Farruggia et al. 2018] or on alignments [Na et al. 2013a; Na et al. 2013b].

There are a few known asymptotic bounds between the repetitiveness measures r, z, g, and e: $z \le g = O(z \log(n/z))$ [Rytter 2003; Charikar et al. 2005; Jeż 2016], $e = \Omega(\max(r, z, g))$ [Belazzougui et al. 2015a; Belazzougui and Cunial 2017b] and, very recently, $r = O(z \log^2 n)$ [Kempa and Kociumaka 2019]. Examples of string families are known that show that r is not comparable with z and g [Belazzougui et al. 2015a; Prezza 2016]. Experimental results [Mäkinen et al. 2010; Kreft and Navarro 2013; Belazzougui et al. 2015a; Claude et al. 2016], on the other hand, suggest that in typical repetitive texts it holds $z < r \approx g \ll e$.

For highly repetitive texts, one hopes to have a compressed index not only able to count and locate pattern occurrences, but also to *replace* the text with a compressed version that nonetheless can efficiently *extract* any substring $T[i \mathinner{..} i + \ell]$. Indexes that, implicitly or not, contain a replacement of T, are called *self-indexes*. As can be seen in Table I, self-indexes with $O(z)$ space require up to $O(z)$ time per extracted character, and none exists within $O(r)$ space. Good extraction times are instead obtained with $O(g)$, $O(z \log(n/z))$, or $O(e)$ space. A lower bound for grammar-based representations [Verbin and Yu 2013] shows that $\Omega((\log n)^{1-\epsilon}/\log g)$ time, for any constant $\epsilon > 0$, is needed to access one random position within $O(\mathrm{poly}(g))$ space. This bound shows that various current techniques using structures bounded in terms of g or z [Bille et al. 2015; Belazzougui et al. 2015c; Gagie et al. 2015; Belazzougui et al. 2015b] are nearly optimal (note that $g = \Omega(\log n)$, thus the space of all these structures is $O(\mathrm{poly}(g))$). In an extended article [Chen et al. 2012, Thm. 6], the authors give a lower bound in terms of r, for binary texts on a RAM machine of $w = \Theta(\log n)$ bits: $\Omega((\log n)^{1-\epsilon})$ for some constant ϵ when using $O(\mathrm{poly}(r \log n))$ space.

In more sophisticated applications, especially in bioinformatics, it is desirable to support a more complex set of operations, which constitute a full suffix tree functionality [Gusfield 1997; Ohlebusch 2013; Mäkinen et al. 2015]. While Mäkinen et al. [2010] offered suffix tree functionality, they had the same problem of needing $\Theta(n/s)$ space to achieve $O(s \log n)$ time for most suffix tree operations. Only recently a suffix tree of size $O(\overline{e})$ supports most operations in time $O(\log n)$ [Belazzougui et al. 2015a; Belazzougui and Cunial 2017b], where $\overline{e}$ refers to the e measure of T plus that of T reversed.

Summarizing Table I and our discussion, the situation on repetitive text indexing is as follows.

(1) The RLFM-index is the only structure able to count the occurrences of P in T in time $O(m \log n)$. However, it does not offer efficient locating within $O(r)$ space.
(2) The only structure that in practice is clearly smaller than the RLFM-index, using $O(z)$ space [Kreft and Navarro 2013], has unbounded locate time. Structures using potentially competitive space, $O(g)$, have an $\Omega(m^2)$ one-time overhead in the locate time [Claude and Navarro 2010; 2012; Gagie et al. 2012; Navarro 2019].
(3) Structures offering lower locate times require $\Omega(z \log(n/z))$ space [Gagie et al. 2014; Nishimoto et al. 2015; Bille et al. 2018; Christiansen and Ettienne 2018; Navarro 2019], $\Theta(\overline{r} + z)$ space [Belazzougui et al. 2015a] (where $\overline{r}$ is the sum of r for T and its reverse), or $\Omega(e)$ space [Belazzougui et al. 2015a; Takagi et al. 2017; Belazzougui and Cunial 2017a].

Table I. Previous and our new results on counting, locating, extracting, and supporting suffix tree functionality. We simplified some formulas with tight upper bounds. The variables are the text size n, pattern length m, number of occurrences occ of the pattern, alphabet size σ, extracted length ℓ, Lempel-Ziv parsing size z, grammar size g, BWT runs r, CDAWG size e, and machine word length in bits w. Variable $h \leq z$ is the depth of the dependency chain in the Lempel-Ziv parse, $\epsilon > 0$ is an arbitrarily small constant, and s is a parameter. Symbols $\overline{r}$ or $\overline{e}$ mean r or e of T plus r or e of its reverse. The z of Kreft and Navarro [2013] refers to the Lempel-Ziv variant that does not allow overlaps between sources and targets, but their index actually works in either variant.

Index	Space	**Count time**
Navarro [2019, Thm. 6]	$O(z\log(n/z))$	$O(m\log n + m\log^{2+\epsilon}(n/z))$
Navarro [2019, Thm. 5]	$O(g)$	$O(m^2 + m\log^{2+\epsilon} g)$
Mäkinen et al. [2010, Thm. 17]	$O(r)$	$O(m(\frac{\log\sigma}{\log\log r} + (\log\log n)^2))$
This paper (Lem. 2.1)	$O(r)$	$O(m\log\log_w(\sigma + n/r))$
This paper (Thm. 4.10)	$O(r\log\log_w(\sigma + n/r))$	$O(m)$
This paper (Thm. 4.11)	$O(rw\log_\sigma\log_w n)$	$O(\lceil m\log(\sigma)/w\rceil)$

Index	Space	**Locate time**
Kreft and Navarro [2013, Thm. 4.11]	$O(z)$	$O(m^2h + (m + occ)\log z)$
Christiansen and Ettienne [2018, Thm. 2(3)]	$O(z\log(n/z))$	$O(m + \log^\epsilon z + occ(\log^\epsilon z + \log\log n))$
Christiansen and Ettienne [2018, Thm. 2(1)]	$O(z\log(n/z) + z\log\log z)$	$O(m + occ(\log^\epsilon z + \log\log n))$
Bille et al. [2018, Cor. 1]	$O(z\log(n/z)\log\log z)$	$O(m(1 + \log^\epsilon z/\log(n/z)) + occ\log\log n)$
Bille et al. [2018, Cor. 1]	$O(z(\log(n/z) + \log\log z)\log\log z)$	$O(m + occ\log\log n)$
Claude and Navarro [2012, Thm. 1]	$O(g)$	$O(m^2\log\log_g n + (m + occ)\log g)$
Gagie et al. [2012, Thm. 4]	$O(g + z\log\log z)$	$O(m^2 + (m + occ)\log\log n)$
Mäkinen et al. [2010, Thm. 20]	$O(r + n/s)$	$O((m + s\cdot occ)(\frac{\log\sigma}{\log\log r} + (\log\log n)^2))$
Belazzougui et al. [2015a, Thm. 3]	$O(\overline{r} + z)$	$O(m(\log z + \log\log n) + occ(\log^\epsilon z + \log\log n))$
This paper (Thm. 3.6)	$O(r)$	$O((m + occ)\log\log_w(\sigma + n/r))$
This paper (Thm. 4.10)	$O(r\log\log_w(\sigma + n/r))$	$O(m + occ)$
This paper (Thm. 4.11)	$O(rw\log_\sigma\log_w n)$	$O(\lceil m\log(\sigma)/w\rceil + occ)$
Belazzougui and Cunial [2017a, Thm. 1]	$O(e)$	$O(m + occ)$

Structure	Space	**Extract time**
Kreft and Navarro [2013, Thm. 4.11]	$O(z)$	$O(\ell\, h)$
Belazzougui et al. [2015b, Thm. 2]	$O(z\log(n/z))$	$O((1 + \ell/\log_\sigma n)\log(n/z))$
Belazzougui et al. [2015c, Thm. 1]	$O(g)$	$O(\log n + \ell/\log_\sigma n)$
Belazzougui et al. [2015c, Thm. 2]	$O(g\log^\epsilon n\log(n/g))$	$O(\log n/\log\log n + \ell/\log_\sigma n)$
Mäkinen et al. [2010, Thm. 20]	$O(r + n/s)$	$O((s + \ell)(\frac{\log\sigma}{\log\log r} + (\log\log n)^2))$
This paper (Thm. 5.1)	$O(r\log(n/r))$	$O(\log(n/r) + \ell\log(\sigma)/w)$
Belazzougui and Cunial [2017a, Thm. 1]	$O(e)$	$O(\log n + \ell)$

Structure	Space	**Typical suffix tree operation time**
Mäkinen et al. [2010, Thm. 30]	$O(r + n/s)$	$O(s(\frac{\log\sigma}{\log\log r} + (\log\log n)^2))$
This paper (Thm. 6.1)	$O(r\log(n/r))$	$O(\log(n/r))$
Belazzougui and Cunial [2017b, Thm. 1]	$O(\overline{e})$	$O(\log n)$

(4) Self-indexes with efficient extraction require $\Omega(z\log(n/z))$ space [Rytter 2003; Charikar et al. 2005; Gagie et al. 2015; Belazzougui et al. 2015b; Bille et al. 2018], $\Omega(g)$ space [Bille et al. 2015; Belazzougui et al. 2015c], or $\Omega(e)$ space [Takagi et al. 2017; Belazzougui and Cunial 2017a].

(5) The only efficient compressed suffix tree requires $\Theta(\overline{e})$ space [Belazzougui and Cunial 2017b].

(6) Only a few of all these indexes have been implemented, as far as we know [Mäkinen et al. 2010; Claude and Navarro 2010; Kreft and Navarro 2013; Belazzougui et al. 2015a].

1.2. Contributions

Efficiently locating the occurrences of P in T within $O(r)$ space has been a bottleneck and an open problem for almost a decade. In this paper we finally give a solution to this problem. Our precise contributions, largely detailed in Tables I and II, are the following. Our results hold in the RAM model with a machine word of $w = \Omega(\log n)$ bits.

Table II. Our contributions. For any “Count + Locate”, we can do only “Count” in the time given by setting $occ = 0$.

Functionality	Space (words)	Time
Count + Locate (Lem. 2.1, Thm. 3.6)	$O(r)$	$O(m \log\log_w(\sigma + n/r) + occ \log\log_w(n/r))$
Count + Locate (Lem. 3.7)	$O(r \log\log_w(n/r))$	$O(m \log\log_w(\sigma + n/r) + occ)$
Count + Locate (Thm. 4.10)	$O(r \log\log_w(\sigma + n/r))$	$O(m + occ)$
Count + Locate (Thm. 4.11)	$O(rw \log_\sigma \log_w n)$	$O(\lceil m \log(\sigma)/w \rceil + occ)$
Extract (Thm. 5.1)	$O(r \log(n/r))$	$O(\log(n/r) + \ell \log(\sigma)/w)$
Access *SA*, *ISA*, *LCP* (Thm. 5.4–5.8)	$O(r \log(n/r))$	$O(\log(n/r) + \ell)$
Count + Locate (Thm. 5.9)	$O(r \log(n/r))$	$O(m + occ)$
Suffix tree (Thm. 6.1)	$O(r \log(n/r))$	$O(\log(n/r))$ for most operations

(1) We improve the counting time of the RLFM-index to $O(m \log\log_w(\sigma + n/r))$, where $\sigma \le r$ is the alphabet size of T, while retaining the $O(r)$ space, in Lemma 2.1.

(2) We show in Theorem 3.6 how to locate each occurrence in time $O(\log\log_w(n/r))$, within $O(r)$ space. We reduce that time to $O(1)$ by using slightly more space, $O(r \log\log_w(n/r))$, in Lemma 3.7.

(3) By using $O(r \log\log_w(\sigma + n/r))$ space, we obtain in Theorem 4.10 optimal locate time in the general setting, $O(m + occ)$, as well as optimal counting time, $O(m)$. This had been obtained before only with space bounds $O(e)$ [Belazzougui and Cunial 2017a] or $O(\overline{e})$ [Takagi et al. 2017].

(4) By increasing the space to $O(rw \log_\sigma \log_w n)$, we obtain in Theorem 4.11 optimal locate time, $O(\lceil m \log(\sigma)/w \rceil + occ)$, and optimal counting time, $O(\lceil m \log(\sigma)/w \rceil)$, in the packed setting (i.e., the pattern symbols come packed in blocks of $w/\log\sigma$ symbols per word). This had not been achieved so far by any compressed index, but only by uncompressed ones [Navarro and Nekrich 2017].

(5) We give the first structure built on BWT runs that replaces T while retaining direct access, in Theorem 5.1. It extracts any substring of length ℓ in time $O(\log(n/r) + \ell \log(\sigma)/w)$, using $O(r \log(n/r))$ space. As discussed, even the additive penalty is near-optimal [Chen et al. 2012, Thm. 6].

(6) Within the same $O(r \log(n/r))$ space, we also show in Theorems 5.4–5.8 how to access ℓ consecutive cells of the suffix array, inverse suffix array, and longest common prefix array of T, in time $O(\log(n/r) + \ell)$. With the recent bound $r = O(z \log^2 n)$ [Kempa and Kociumaka 2019], we also have this functionality in $O(z \log^3 n) \subseteq O(g \log^3 n)$ space, which was not achieved before.

(7) For completeness, Theorem 5.9 shows that we can also obtain optimal locating and counting time within $O(r \log(n/r))$ space, which on little-compressible texts can be less than $O(r \log\log_w(\sigma + n/r))$.

(8) We give the first compressed suffix tree whose space is bounded in terms of r, $O(r \log(n/r))$ words, in Theorem 6.1. It implements most navigation operations in time $O(\log(n/r))$. There exist only comparable suffix trees within $O(\overline{e})$ space [Belazzougui and Cunial 2017b], taking $O(\log n)$ time for most operations. Again, with the recent bound $r = O(z \log^2 n)$ [Kempa and Kociumaka 2019], we also provide suffix tree functionality in $O(z \log^3 n)$ space.

(9) We provide a proof-of-concept implementation of the most basic index (the one locating within $O(r)$ space), and show that it outperforms all the other implemented alternatives by orders of magnitude in space or in time to locate pattern occurrences.

Contribution 1 is a simple update of the RLFM-index [Mäkinen et al. 2010] with newer data structures for rank and predecessor queries [Belazzougui and Navarro 2015]. We present it in Section 2, together with a review of the basic concepts needed to follow the paper.

Contribution 2 is one of the central parts of the paper, and is obtained in Section 3 in two steps. The first uses the fact that we can carry out the classical RLFM-index counting process for P in a way that we always know the position of one occurrence in T [Prezza 2016; Policriti and Prezza 2018]; we give a simpler proof of this fact in Lemma 3.2. The second shows that, if we know the position in T of one occurrence of the BWT, then we can quickly obtain the preceding and following ones with an $O(r)$-size sampling. This is achieved by using the BWT runs to induce *phrases* in T (which are somewhat analogous to the Lempel-Ziv phrases [Lempel and Ziv 1976]) and showing that the positions of occurrences within phrases can be obtained from the positions of their preceding phrase boundary. The time $O(1)$ is obtained by using an extended sampling.

For Contributions 3 and 4, we use in Section 4 the fact that the RLFM-index on a text regarded as a sequence of metasymbols of length s, with the s possible shifts, has $O(rs)$ runs, so that we can process the pattern by chunks of s symbols. The optimal packed time is obtained by enlarging the samplings.

In Section 5, Contribution 5 uses an analogue of the Block Tree [Belazzougui et al. 2015b] built on the BWT-induced phrases, which satisfy the property that any distinct string has an occurrence overlapping a border between phrases. Contribution 6 is obtained by showing that direct access to the suffix array SA, inverse suffix array ISA, and array LCP of T, can be supported in a similar way because they inherit the same repetitiveness properties of the text.

Section 5 also includes Contribution 7, which is relevant only when $r > n/\log n$. For those values of r, the allowed space enables us to use semi-succinct representations of $O(n \log\log n)$ bits, on which we obtain optimal counting and locating.

Contribution 8 needs, in addition to accessing the arrays SA, ISA, and LCP, some sophisticated operations on the LCP array [Fischer et al. 2009] that are not well supported by Block Trees. In Section 6, we implement suffix trees by building a run-length context-free grammar [Nishimoto et al. 2016] of size $O(r\log(n/r))$ on the differential LCP array, and then implementing the required operations on it.

The results of Contribution 9 are shown in Section 7. Our experimental results show that our simple $O(r)$-space index outperforms the alternatives by 1–2 orders of magnitude in time when locating the occurrences of a pattern, while being simultaneously smaller or nearly as small. Our implementation is also 1–2 orders of magnitude smaller and/or 2–3 orders of magnitude faster than current implementations of the original FM-index on repetitive datasets. The only compact structure outperforming our index in space, the CDAWG, is 60 times larger and thus out of scale in this scenario.

We conclude in Section 8 with a discussion of the impact of the work and open problems.

In Appendix A we describe construction algorithms for all our data structures, achieving construction spaces bounded in terms of r for the simpler and most practical structures.

This article extends the conference version presented in *SODA 2018* [Gagie et al. 2018b]. The extension entails, on the one hand, a significant improvement in Contributions 3 and 4: in Section 4, optimal time locating is now obtained in a much simpler way and in less space. Further, optimal time counting is obtained as well, which is new. Contribution 6, that is, the machinery to support suffix tree functionality in Section 6, is also new. We also present an improved implementation in Section 7, with better experimental results. Finally, the construction algorithms in Appendix A are new as well.

2. BASIC CONCEPTS

A string is a sequence $S[1\,..\,\ell] = S[1]S[2]\cdots S[\ell]$, of length $\ell = |S|$, of symbols (or characters, or letters) chosen from an alphabet $[1\,..\,\sigma] = \{1, 2, \ldots, \sigma\}$, that is, $S[i] \in [1\,..\,\sigma]$ for all $1 \le i \le \ell$. We use $S[i\,..\,j] = S[i]\cdots S[j]$, with $1 \le i, j \le \ell$, to denote a substring of S, which is the empty string ε if $i > j$. A prefix of S is a substring of the form $S[1\,..\,i]$ (also written $S[..\,i]$) and a suffix is a substring of the form $S[i\,..\,\ell]$ (also written $S[i\,..]$). The juxtaposition of strings and/or symbols represents their concatenation.

We will consider indexing a *text* $T[1\,..\,n]$, which is a string over alphabet $[1\,..\,\sigma]$ terminated by the special symbol $\$ = 1$, that is, the lexicographically smallest one, which appears only at $T[n] = \$$. This makes any lexicographic comparison between suffixes well defined.

Our computation model is the transdichotomous RAM, with a word of $w = \Omega(\log n)$ bits, where all the standard arithmetic and logic operations can be carried out in constant time. In this article we generally measure space in words and assume logarithms to the base 2 by default.

2.1. Suffix trees and suffix arrays

The *suffix tree* [Weiner 1973] of $T[1\,..\,n]$ is a compacted trie where all the n suffixes of T have been inserted. By compacted we mean that chains of degree-1 nodes are collapsed into a single edge that is labeled with the concatenation of the individual symbols that labeled the collapsed edges. The suffix tree has n leaves and less than n internal nodes. By representing edge labels with pointers to T, the suffix tree uses $O(n)$ space, and can be built in $O(n)$ time [Weiner 1973; McCreight 1976; Ukkonen 1995; Farach-Colton et al. 2000].

The *suffix array* [Manber and Myers 1993] of $T[1\,..\,n]$ is an array $SA[1\,..\,n]$ storing a permutation of $[1\,..\,n]$ so that, for all $1 \le p < n$, the suffix $T[SA[p]\,..]$ is lexicographically smaller than the suffix $T[SA[p+1]\,..]$. Thus $SA[p]$ is the starting position in T of the pth lexicographically smallest suffix of T.

The suffix array can be regarded as an array collecting the suffix tree leaves. It uses n words of space and can be built in $O(n)$ time without building the suffix tree [Kim et al. 2005; Ko and Aluru 2005; Kärkkäinen et al. 2006].

All the occurrences of a pattern string $P[1..m]$ in T can be easily spotted in the suffix tree or array. In the suffix tree, we descend from the root matching the successive symbols of P with the strings labeling the edges. If P is in T, the symbols of P will be exhausted at a node v or inside an edge leading to a node v; this node is called the *locus* of P, and all the occ leaves descending from v are the suffixes starting with P, that is, the starting positions of the occurrences of P in T. By using perfect hashing to store the first characters of the edge labels descending from each node v, we reach the locus in optimal time $O(m)$ and the space is still $O(n)$. If P comes packed using $w/\log\sigma$ symbols per computer word, we can descend in time $O(\lceil m\log(\sigma)/w\rceil)$ [Navarro and Nekrich 2017], which is optimal in the packed model. In the suffix array, all the suffixes starting with P form a range $SA[sp..ep]$, which is binary searched for in time $O(m\log n)$, or $O(m+\log n)$ with additional structures [Manber and Myers 1993].

The inverse permutation of SA, $ISA[1..n]$, is called the *inverse suffix array*, so that $ISA[i]$ is the lexicographical position of the suffix $T[i..]$ among all the suffixes of T.

Another important concept related to suffix arrays and trees is the longest common prefix array. Let $lcp(S,S')$ be the length of the longest common prefix between two strings $S \neq S'$, that is, $S[1..lcp(S,S')] = S'[1..lcp(S,S')]$ but $S[lcp(S,S')+1] \neq S'[lcp(S,S')+1]$. Then we define the *longest common prefix array* $LCP[1..n]$ as $LCP[1]=0$ and $LCP[p] = lcp(T[SA[p-1]..], T[SA[p]..])$. The LCP array uses n words and can be built in $O(n)$ time [Kasai et al. 2001].

2.2. Self-indexes

A *self-index* is a data structure built on $T[1..n]$ that provides at least the following functionality:

Count. Given a pattern $P[1..m]$, compute the number occ of occurrences of P in T.
Locate. Given a pattern $P[1..m]$, return the occ positions where P occurs in T.
Extract. Given a range $[i..i+\ell-1]$, return $T[i..i+\ell-1]$.

The last operation allows a self-index to replace T, that is, it is not necessary to store T since any desired substring can be extracted from the self-index. This can be trivially obtained by including a copy of T as a part of the self-index, but it is challenging when the self-index must use little space.

In principle, suffix trees and arrays can be regarded as self-indexes that can count in time $O(m)$ or $O(\lceil m\log(\sigma)/w\rceil)$ (suffix tree, by storing occ in each node v) and $O(m\log n)$ or $O(m+\log n)$ (suffix array, with $occ = ep-sp+1$), locate each occurrence in $O(1)$ time, and extract in time $O(\lceil \ell\log(\sigma)/w\rceil)$ (because they maintain a plain copy of T). However, they use $O(n\log n)$ bits, much more than the $n\log\sigma$ bits needed to represent T in plain form. We are interested in *compressed self-indexes* [Navarro and Mäkinen 2007; Navarro 2016], which use the space required by a compressed representation of T (under some compressibility measure) plus some redundancy (at worst $o(n\log\sigma)$ bits). We describe later the FM-index, a particular self-index of interest to us.

2.3. The Burrows-Wheeler Transform

The *Burrows-Wheeler Transform (BWT)* of $T[1..n]$ [Burrows and Wheeler 1994] is a string $BWT[1..n]$ defined as $BWT[p] = T[SA[p]-1]$ if $SA[p]>1$, and $BWT[p] = T[n] = \$$ if $SA[p]=1$. That is, BWT has the same symbols of T in a different order, and is a reversible transform.

The array BWT is obtained from T by first building SA, although it can be built directly, in $O(n)$ time and within $O(n\log\sigma)$ bits of space [Munro et al. 2017]. To obtain T from BWT [Burrows and Wheeler 1994], one considers two arrays, $L[1..n] = BWT$ and $F[1..n]$, which contains all the symbols of L (or T) in ascending order. Alternatively, $F[p] = T[SA[p]]$, so $F[p]$ follows $L[p]$ in T. We need a function that maps any $L[p]$ to the position q of that same character in F. The formula is $LF(p) = C[c] + \mathrm{rank}[p]$, where $c = L[p]$, $C[c]$ is the number of occurrences of symbols less than c in L, and $\mathrm{rank}[p]$ is the number of occurrences of symbol $L[p]$ in $L[1..p]$. A simple $O(n)$-time pass on L suffices to compute arrays C and rank using $O(n\log\sigma)$ bits of space. Once they are computed, we reconstruct $T[n] = \$$ and $T[n-k] \leftarrow L[LF^{k-1}(1)]$ for $k = 1,\ldots,n-1$, in $O(n)$ time as well. Note that LF is a permutation with a single cycle.

2.4. Compressed suffix arrays and FM-indexes

Compressed suffix arrays [Navarro and Mäkinen 2007] are a particular case of self-indexes that simulate SA in compressed form. Therefore, they aim to obtain the suffix array range $[sp\,..\,ep]$ of P, which is sufficient to count since P then appears $occ = ep - sp + 1$ times in T. For locating, they need to access the content of cells $SA[sp], \ldots, SA[ep]$, without having SA stored.

The FM-index [Ferragina and Manzini 2005; Ferragina et al. 2007] is a compressed suffix array that exploits the relation between the string $L = BWT$ and the suffix array SA. It stores L in compressed form (as it can be easily compressed to the high-order empirical entropy of T [Manzini 2001]) and adds sublinear-size data structures to compute (i) any desired position $L[p]$, (ii) the generalized *rank function* $\mathrm{rank}_c(L,p)$, which is the number of times symbol c appears in $L[1\,..\,p]$. Note that these two operations permit, in particular, computing $\mathrm{rank}[p] = \mathrm{rank}_{L[p]}(L,p)$, which is called *partial rank*. Therefore, they compute

$$LF(p) \;=\; C[L[p]] + \mathrm{rank}_{L[p]}(L,p).$$

For counting, the FM-index resorts to *backward search*. This procedure reads P backwards and at any step knows the range $[sp_j, ep_j]$ of $P[j\,..\,m]$ in T. Initially, we have the range $[sp_{m+1}\,..\,ep_{m+1}] = [1\,..\,n]$ for $P[m+1\,..\,m] = \varepsilon$. Given the range $[sp_{j+1}\,..\,ep_{j+1}]$, one obtains the range $[sp_j\,..\,ep_j]$ from $c = P[j]$ with the operations

$$\begin{aligned} sp_j &= C[c] + \mathrm{rank}_c(L, sp_{j+1} - 1) + 1, \\ ep_j &= C[c] + \mathrm{rank}_c(L, ep_{j+1}). \end{aligned}$$

Thus the range $[sp\,..\,ep] = [sp_1\,..\,ep_1]$ is obtained with $O(m)$ computations of rank, which dominates the counting complexity.

For locating, the FM-index (and most compressed suffix arrays) stores sampled values of SA at regularly spaced text positions, say multiples of s. Thus, to retrieve $SA[p]$, we find the smallest k for which $SA[LF^k(p)]$ is sampled, and then the answer is $SA[p] = SA[LF^k(p)] + k$. This is because function LF virtually traverses the text backwards, that is, it drives us from $L[p]$, which precedes suffix $SA[p]$, to its position $F[q]$, where the suffix $SA[q]$ starts with $L[p]$, that is, $SA[q] = SA[p] - 1$:

$$SA[LF(p)] \;=\; SA[p] - 1.$$

Since it is guaranteed that $k < s$, each occurrence is located with s accesses to L and computations of LF, and the extra space for the sampling is $O((n \log n)/s)$ bits, or $O(n/s)$ words.

For extracting, a similar sampling is used on ISA, that is, we sample the positions of ISA that are multiples of s. To extract $T[i\,..\,i+\ell-1]$ we find the smallest multiple of s in $[i+\ell\,..\,n]$, $j = s \cdot \lceil (i+\ell)/s \rceil$, and extract $T[i\,..\,j]$. Since $ISA[j] = p$ is sampled, we know that $T[j-1] = L[p]$, $T[j-2] = L[LF(p)]$, and so on. In total we require at most $\ell + s$ accesses to L and computations of LF to extract $T[i\,..\,i+\ell-1]$. The extra space of this second sampling is also $O(n/s)$ words.

For example, using a representation [Belazzougui and Navarro 2015] that accesses L and computes partial ranks in constant time (so LF is computed in $O(1)$ time), and computes rank in the optimal time $O(\log\log_w \sigma)$, an FM-index can count in time $O(m \log\log_w \sigma)$, locate each occurrence in $O(s)$ time, and extract ℓ symbols of T in time $O(s+\ell)$, by using $O(n/s)$ space on top of the empirical entropy of T [Belazzougui and Navarro 2015]. There exist even faster variants [Belazzougui and Navarro 2014], but they do not rely on backward search.

2.5. The Run-Length FM-index

One of the sources of the compressibility of BWT is that symbols are clustered into $r \le n$ *runs*, which are maximal substrings formed by the same symbol. Mäkinen and Navarro [2005] proved a (relatively weak) bound on r in terms of the high-order empirical entropy of T and, more importantly, designed an FM-index variant that uses $O(r)$ words of space, called *Run-Length FM-index* or *RLFM-index*. They later experimented with several variants of the RLFM-index, where the one called RLFM+ [Mäkinen et al. 2010, Thm. 17] corresponds to the original RLFM-index [Mäkinen and Navarro 2005].

The structure stores the *run heads*, that is, the first positions of the runs in BWT, in a data structure $E = \{1\} \cup \{1 < p \le n, BWT[p] \neq BWT[p-1]\}$ that supports predecessor searches. Each element $e \in E$ has associated the value $e.v = |\{e' \in E, e' \le e\}|$, which is its position in a string $L'[1\,..\,r]$ that stores the run symbols. Another array, $D[0\,..\,r]$, stores the cumulative lengths of the runs after stably sorting them lexicographically by their symbols (with $D[0] = 0$). Let array $C'[1\,..\,\sigma]$ count the number of *runs* of symbols smaller than c in L. One can then simulate

$$\mathrm{rank}_c(L, p) \;=\; D[C'[c] + \mathrm{rank}_c(L', q.v - 1)] + [\text{if } L'[q.v] = c \text{ then } p - q + 1 \text{ else } 0],$$

where $q = \mathit{pred}(E, p)$, at the cost of a predecessor search (pred) in E and a rank on L'. By using up-to-date data structures, the counting performance of the RLFM-index can be stated as follows.

LEMMA 2.1. *The Run-Length FM-index of a text $T[1\,..\,n]$ whose BWT has r runs can occupy $O(r)$ words and count the number of occurrences of any pattern $P[1\,..\,m]$ in $O(m \log\log_w(\sigma + n/r))$ time. It also computes any $LF(p)$ and access to any symbol $BWT[p]$ in time $O(\log\log_w(n/r))$.*

PROOF. We use the RLFM+ [Mäkinen et al. 2010, Thm. 17], using the structure of Belazzougui and Navarro [2015, Thm. 10] for the sequence L' (with constant access time) and the predecessor data structure described by Belazzougui and Navarro [2015, Thm. 14] to implement E (instead of the bitvector used in the original RLFM+). The RLFM+ also implements D with a bitvector, but we use a plain array. The sum of both operation times is $O(\log\log_w \sigma + \log\log_w(n/r))$, which can be written as $O(\log\log_w(\sigma + n/r))$. To access $BWT[p] = L[p] = L'[\mathit{pred}(E, p).v]$ we only need a predecessor search on E, which takes time $O(\log\log_w(n/r))$, and a constant-time access to L'. Finally, we compute LF faster than a general rank query, since we only need the partial rank query

$$\mathrm{rank}_{L[p]}(L, p) \;=\; D[C'[L'[q.v]] + \mathrm{rank}_{L'[q.v]}(L', q.v) - 1] + (p - q + 1),$$

which is correct since $L[p] = L'[q.v]$. The operation $\mathrm{rank}_{L'[q.v]}(L', q.v)$ can be supported in constant time using $O(r)$ space, by just recording all the answers, and therefore the time for LF on L is also dominated by the predecessor search on E (to compute q), which takes $O(\log\log_w(n/r))$ time. □

To provide locating and extracting functionality, Mäkinen et al. [2010] use the sampling mechanism we described for the FM-index. Therefore, although they can efficiently count within $O(r)$ space, they need a much larger space, $O(n/s)$, to support these operations in time proportional to s. Despite various efforts [Mäkinen et al. 2010], this has been a bottleneck in theory and in practice since then.

We will generally assume that σ is the *effective* alphabet of T, that is, the σ symbols appear in T. This implies that $\sigma \le r \le n$. If this is not the case, we can map T to an effective alphabet $[1\,..\,\sigma']$ before indexing it. A mapping of $\sigma' \le r$ words then stores the actual symbols when extracting a substring of T is necessary. For searches, we have to map the m positions of P to the effective alphabet. By storing a perfect hash or a deterministic dictionary [Ružić 2008] of $O(\sigma') = O(r)$ words, we map each symbol of P in constant time. On the other hand, to handle packed symbols we must use tables of size $O(2^{\epsilon w})$, for any constant $\epsilon > 0$, to translate $\Theta(w/\log\sigma)$ symbols in constant time (in either direction). Note that the packed setting is asymptotically relevant only when σ is small, $\log\sigma = o(w)$, and thus it is unlikely that we use it with large $\sigma = \omega(r)$. Only under this combination the assumption $\sigma \le r$ requires us to spend $O(2^{\epsilon w})$ extra space and construction time.

2.6. Compressed suffix trees

Suffix trees provide a much more complete functionality than self-indexes, and are used to solve complex problems especially in bioinformatic applications [Gusfield 1997; Ohlebusch 2013; Mäkinen et al. 2015]. A *compressed suffix tree* is regarded as an enhancement of a compressed suffix array (which, in a sense, represents only the leaves of the suffix tree). Such a compressed representation must be able to simulate the operations on the classical suffix tree (see Table IV later in the article), while using little space on top of the compressed suffix array. The first such compressed suffix tree [Sadakane 2007] used $O(n)$ extra bits, and there are several variants using $o(n)$ extra bits [Fischer et al. 2009; Fischer 2010; Russo et al. 2011; Gog and Ohlebusch 2013; Abeliuk et al. 2013].

Instead, there are no compressed suffix trees using $O(r\,\mathrm{polylog}(n))$ space. An extension of the RLFM-index [Mäkinen et al. 2010] still needs $O(n/s)$ space to carry out most of the suffix tree operations in time $O(s \log n)$. Most variants designed for repetitive text collections [Abeliuk et al. 2013; Navarro and Ordóñez 2016; Farruggia et al. 2018; Cáceres and Navarro 2019] are heuristic and do not offer worst-case guarantees. The only exception is the compressed suffix tree of Belazzougui and Cunial [2017b], which uses space $O(\bar{e})$ and supports most operations in time $O(\log n)$.

3. LOCATING IN BWT-RUNS BOUNDED SPACE

In this section we show that, if the BWT of a text $T[1\,..\,n]$ has r runs, then we can have an index using $O(r)$ space that not only efficiently finds the interval $SA[sp\,..\,ep]$ of the occurrences of a pattern $P[1\,..\,m]$ (as was already known in the literature, see Section 2.5) but that can locate each such occurrence in time $O(\log\log_w(n/r))$ on a RAM machine of w bits. Further, the time per occurrence becomes constant if the space is raised to $O(r\log\log_w(n/r))$.

We start with Lemma 3.2, which shows that the typical backward search process can be enhanced so that we always know the position of one of the values in $SA[sp\,..\,ep]$. We give a simplification of the previous proof [Prezza 2016; Policriti and Prezza 2018]. Lemma 3.5 then shows how to efficiently obtain the two cells of SA that surround the value of one cell we know. This allows us to extend the first known cell in both directions, until obtaining the whole interval $SA[sp\,..\,ep]$. Theorem 3.6 summarizes the main result of this section.

Later, Lemma 3.7 shows how this process can be accelerated by using more space. We extend the idea in Lemma 3.8, obtaining LCP values in the same way we obtain SA values. While not of immediate use for locating, this result is useful later in the article and also has independent interest.

Definition 3.1. We say that a text character $T[i]$ is *sampled* if and only if $i = 1$ or $T[i]$ is the first or last character in its BWT run. That is, $T[1]$, $T[SA[n]-1]$, and $T[SA[1]-1] = T[n-1]$ are sampled and, if $p > 1$ and $BWT[p] \neq BWT[p-1]$, then $T[SA[p-1]-1]$ and $T[SA[p]-1]$ are sampled. In general, $T[i]$ is *s-sampled* if $i = 1$ or $i = SA[p]-1$ and p is at distance at most s from a BWT run border, where sampled characters are assumed to be at distance 1.

LEMMA 3.2. *We can store $O(r)$ words such that, given $P[1\,..\,m]$, in time $O(m\log\log_w(\sigma+n/r))$ we can compute the interval $SA[sp,ep]$ of the occurrences of P in T and also return the position p and content $SA[p]$ of at least one cell in the interval $[sp,ep]$.*

PROOF. We store a RLFM-index and predecessor structures R_c storing the position in BWT of all the sampled characters equal to c, for each $c \in [1\,..\,\sigma]$. Each BWT position $p \in R_c$ is associated with its corresponding text position, that is, we store pairs $\langle p, SA[p]-1\rangle$ in the structures R_c. These structures take a total of $O(r)$ words.

The interval of characters immediately preceding occurrences of the empty string is the entire $BWT[1\,..\,n]$, which clearly includes $P[m]$ as the last character in some run (unless P does not occur in T). It follows that we find an occurrence of $P[m]$ in predecessor time by querying $pred(R_{P[m]}, n)$.

Assume we have found the interval $BWT[sp,ep]$ containing the characters immediately preceding all the occurrences of some (possibly empty) suffix $P[j+1\,..\,m]$ of P, and we know the position and content of some cell $SA[p]$ in the corresponding interval, $sp \le p \le ep$. Since $SA[LF(p)] = SA[p]-1$, if $BWT[p] = P[j]$ then, after the next application of LF-mapping, we still know the position and value of some cell $SA[p']$ corresponding to the interval $BWT[sp',ep']$ for $P[j\,..\,m]$, namely $p' = LF(p)$ and $SA[p'] = SA[p]-1$.

On the other hand, if $BWT[p] \neq P[j]$ but P still occurs somewhere in T (i.e., $sp' \le ep'$), then there is at least one $P[j]$ and one non-$P[j]$ in $BWT[sp,ep]$, and therefore the interval intersects an extreme of a run of copies of $P[j]$, thus holding a sampled character. Then, a predecessor query $pred(R_{P[j]}, ep)$ gives us the desired pair $\langle p', SA[p']-1\rangle$ with $sp \le p' \le ep$ and $BWT[p'] = P[j]$.

Therefore, by induction, when we have computed the BWT interval for P, we know the position and content of at least one cell in the corresponding interval in SA.

To obtain the desired time bounds, we concatenate all the universes of the R_c structures into a single one of size σn, and use a single structure R on that universe: each $\langle p, SA[p-1]\rangle \in R_c$ becomes $\langle (c-1)n+p, SA[p]-1\rangle$ in R, and a search $pred(R_c,q)$ becomes $pred(R,(c-1)n+q)-(c-1)n$. Since R contains $2r$ elements on a universe of size σn, we can have predecessor searches in time $O(\log\log_w(n\sigma/r))$ and space $O(r)$ [Belazzougui and Navarro 2015, Thm. 14]. This is the same $O(\log\log_w(\sigma+n/r))$ time we obtained in Lemma 2.1 to carry out the normal backward search operations on the RLFM-index. □

Lemma 3.2 gives us a toehold in the suffix array, and we now show that this is all we need. We first show that, given the position and contents of one cell of the suffix array SA of a text T, we can compute the contents of the neighbouring cells in $O(\log\log_w(n/r))$ time. It follows that, once we have counted the occurrences of a pattern in T, we can locate them all in $O(\log\log_w(n/r))$ time each.

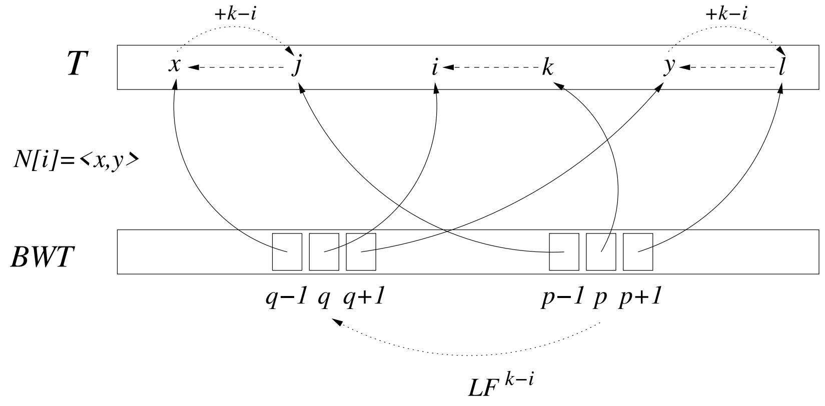

Fig. 1. Illustration of Lemma 3.5. Since $BWT[p] = T[k]$ and i is the predecessor of k, the cells $p-1$, p, and $p+1$ would travel together through consecutive applications of LF, reaching the positions $N[i] = \langle x, y\rangle$ after $k-i$ steps. Thus it must be that $BWT[p-1] = T[x+k-i]$ and $BWT[p+1] = T[y+k-i]$.

Definition 3.3. ([Kärkkäinen et al. 2009]) Let permutation ϕ be defined as $\phi(i) = SA[ISA[i]-1]$ if $ISA[i] > 1$ and $\phi(i) = SA[n]$ otherwise.

That is, given a text position $i = SA[p]$ pointed from suffix array position p, $\phi(i) = SA[ISA[SA[p]]-1] = SA[p-1]$ gives the value of the preceding suffix array cell. Similarly, $\phi^{-1}(i) = SA[p+1]$.

Definition 3.4. We parse T into *phrases* such that $T[i]$ is the first character in a phrase if and only if $T[i]$ is sampled.

LEMMA 3.5. *We can store $O(r)$ words such that functions ϕ and ϕ^{-1} are evaluated in $O(\log\log_w(n/r))$ time.*

PROOF. We store an $O(r)$-space predecessor data structure $P^{\pm}$ with $O(\log\log_w(n/r))$ query time [Belazzougui and Navarro 2015, Thm. 14] for the starting phrase positions i of T (i.e., the sampled text positions). We also store, associated with such values $i \in P^{\pm}$, the positions in T next to the characters immediately preceding and following the corresponding position $BWT[q]$, that is, $N[i] = \langle SA[q-1], SA[q+1]\rangle$ for $i = SA[q]-1$ (for $q = 1$ and $q = n$ we store $\langle null, SA[q+1]\rangle$ and $\langle SA[q-1], null\rangle$, respectively).

Suppose we know $SA[p] = k+1$ and want to know $SA[p-1]$ and $SA[p+1]$. This is equivalent to knowing the position $BWT[p] = T[k]$ and wanting to know the positions in T of $BWT[p-1]$ and $BWT[p+1]$. To compute these positions, we find in $P^{\pm}$ the position i in T of the first character of the phrase containing $T[k]$, take the associated positions $N[i] = \langle x, y\rangle$, and return $SA[p-1] = x+k-i$ and $SA[p+1] = y+k-i$.

To see why this works, let $SA[p-1] = j+1$ and $SA[p+1] = l+1$, that is, j and l are the positions in T of $BWT[p-1] = T[j]$ and $BWT[p+1] = T[l]$. Note that, for all $0 \le t < k-i$, $T[k-t]$ is not the first nor the last character of a run in BWT. Thus, by definition of LF, $LF^t(p-1)$, $LF^t(p)$, and $LF^t(p+1)$, that is, the BWT positions of $T[j-t]$, $T[k-t]$, and $T[l-t]$, are contiguous and within a single run, thus $T[j-t] = T[k-t] = T[l-t]$. Therefore, for $t = k-i-1$, $T[j-(k-i-1)] = T[i+1] = T[l-(k-i+1)]$ are contiguous in BWT, and thus a further LF step yields that $BWT[q] = T[i]$ is immediately preceded and followed by $BWT[q-1] = T[j-(k-i)]$ and $BWT[q+1] = T[l-(k-i)]$. That is, $N[i] = \langle SA[q-1], SA[q+1]\rangle = \langle j-(k-i)+1, l-(k-i)+1\rangle$ and our answer is correct. Figure 1 illustrates the proof. □

We then obtain the main result of this section.

THEOREM 3.6. *We can store a text $T[1..n]$, over alphabet $[1..\sigma]$, in $O(r)$ words, where r is the number of runs in the BWT of T, such that later, given a pattern $P[1..m]$, we can count the occurrences of P in T in $O(m\log\log_w(\sigma+n/r))$ time and (after counting) report their occ locations in overall time $O(occ\cdot\log\log_w(n/r))$.*

3.1. Larger and faster

The following lemma shows that the above technique can be generalized. The result is a space-time tradeoff allowing us to list each occurrence in constant time at the expense of a slight increase in space usage. This will be useful later in the article, in particular to obtain optimal-time locating. To obtain it, we enhance the sampling $P^{\pm}$ of Lemma 3.5 to ensure that we find s consecutive samples. For this reason, we must store separately the sampled positions preceding and following run borders.

LEMMA 3.7. *Let $s > 0$. We can store a data structure of $O(rs)$ words such that, given $SA[p]$, we can compute $SA[p-j]$ and $SA[p+j]$ for $j = 1, \ldots, s'$ and any $s' \leq s$, in $O(\log\log_w(n/r) + s')$ time.*

PROOF. Consider all BWT positions $q_1 < \cdots < q_t$ of s-sampled characters, and let $W[1\,..\,t]$ be an array such that $W[k]$ is the text position corresponding to q_k, for $k = 1, \ldots, t$. Now let $q_1^+ < \cdots < q_{t^+}^+$ be the BWT positions having a run border at most s positions after them, and $q_1^- < \cdots < q_{t^-}^-$ be the BWT positions having a run border at most s positions before them; note $t^+, t^- \leq t$. We store the text positions corresponding to $q_1^+ < \cdots < q_{t^+}^+$ and $q_1^- < \cdots < q_{t^-}^-$ in two predecessor structures P^+ and P^-, respectively, of size $O(rs)$. We store, for each $i \in P^+ \cup P^-$, its position $f(i)$ in W, that is, $W[f(i)] = i$.

To answer queries given $SA[p]$, we first compute its P^+-predecessor $i < SA[p]$ in $O(\log\log_w(n/r))$ time, and retrieve $f(i)$. Then, it holds that $SA[p+j] = W[f(i)+j] + (SA[p]-i)$, for $j = 0, \ldots, s$. Computing $SA[p-j]$ is symmetric; we just use P^- instead of P^+.

To see why this procedure is correct, consider the range $SA[p\,..\,p+s]$. We distinguish two cases.

(i) $BWT[p\,..\,p+s]$ contains at least two distinct characters. Then, $SA[p]-1 \in P^+$ (because p is followed by a run break at most s positions away), and is therefore the immediate predecessor of $SA[p]$. Moreover, all BWT positions $[p\,..\,p+s]$ are in $q_1, \ldots, q_t$ (since they are at distance at most s from a run break), and their corresponding text positions are therefore contained in a contiguous range of W (i.e., $W[f(SA[p]-1)\,..\,f(SA[p]-1)+s]$). The claim follows.

(ii) $BWT[p\,..\,p+s]$ contains a single character; we say it is unary. Then $SA[p]-1 \notin P^+$, since there are no run breaks in $BWT[p\,..\,p+s]$. Moreover, by the LF formula, the LF mapping applied on the unary range $BWT[p\,..\,p+s]$ gives a contiguous range $BWT[LF(p)\,..\,LF(p+s)] = BWT[LF(p)\,..\,LF(p)+s]$. Note that this corresponds to a parallel backward step on text positions $SA[p] \to SA[p]-1, \ldots, SA[p+s] \to SA[p+s]-1$. We iterate the application of LF until we end up in a range $BWT[LF^{\delta}(p)\,..\,LF^{\delta}(p+s)]$ that is not unary. Then, $SA[LF^{\delta}(p)]-1$ is the immediate predecessor of $SA[p]$ in P^+, and $\delta+1$ is their distance. This means that with a single predecessor query on P^+ we "skip" all the unary BWT ranges $BWT[LF^k(p)\,..\,LF^k(p+s)]$ for $k = 1, \ldots, \delta-1$ and, as in case (i), we retrieve the contiguous range in W containing the values $SA[p]-\delta, \ldots, SA[p+s]-\delta$; we then add δ to obtain the desired SA values. □

3.2. Accessing LCP

Lemma 3.7 can be further extended to entries of the LCP array, which we will use later in the article. Given $SA[p]$, we compute $LCP[p]$ and its s adjacent entries (note that we do not need to know p, but $SA[p]$). For $s = 0$ this is known as the *permuted LCP (PLCP)* array [Sadakane 2007]. Our result can indeed be seen as an extension of a *PLCP* representation by Fischer et al. [2009]. In Section 6.2 we use different structures that enable the classical access, that is, compute $LCP[p]$ from p, not from $SA[p]$.

LEMMA 3.8. *Let $s > 0$. We can store a data structure of $O(rs)$ words such that, given $SA[p]$, we can compute $LCP[p-j+1]$ and $LCP[p+j]$, for $j = 1, \ldots, s'$ and any $s' \leq s$, in $O(\log\log_w(n/r) + s')$ time.*

PROOF. The proof follows closely that of Lemma 3.7, except that now we sample LCP entries corresponding to suffixes *following* s-sampled BWT positions. Let us define $q_1 < \cdots < q_t$, $q_1^+ < \cdots < q_{t^+}^+$, and $q_1^- < \cdots < q_{t^-}^-$, as well as the predecessor structures P^+ and P^-, exactly as in the proof of Lemma 3.7. We store $LCP'[1\,..\,t] = LCP[q_1], \ldots, LCP[q_t]$. We also store, for each $i \in P^+ \cup P^-$, its corresponding position $f(i)$ in LCP', that is, $LCP'[f(i)] = LCP[ISA[i+1]]$.

To answer queries given $SA[p]$, we first compute its P^+-predecessor $i < SA[p]$ in $O(\log\log_w(n/r))$ time, and retrieve $f(i)$. It then holds that $LCP[p+j] = LCP'[f(i)+j] - (SA[p]-i-1)$, for $j = 1, \ldots, s$. Computing $LCP[p-j]$ for $j = 0, \ldots, s-1$ is symmetric (using P^- instead of P^+).

To see why this procedure is correct, consider the range $SA[p\,..\,p+s]$. We distinguish again two cases.

(i) $BWT[p\,..\,p+s]$ contains at least two distinct characters. Then, as in case (i) of Lemma 3.7, $SA[p]-1 \in P^+$ and is therefore the immediate predecessor $i = SA[p]-1$ of $SA[p]$. Moreover, all BWT positions $[p\,..\,p+s]$ are in $q_1, \ldots, q_t$, and therefore values $LCP[p\,..\,p+s]$ are explicitly stored in a contiguous range in LCP' (i.e., $LCP'[f(i)\,..\,f(i)+s]$). Note that $SA[p]-i = 1$, so $LCP'[f(i)+j]-(SA[p]-i-1) = LCP'[f(i)+j]$ for $j = 0, \ldots, s$. The claim follows.

(ii) $BWT[p\,..\,p+s]$ contains a single character, so it is unary. Then we reason exactly as in case (ii) of Lemma 3.7 to define δ so that $i' = SA[LF^{\delta}(p)]-1$ is the immediate predecessor of $SA[p]$ in P^+ and, as in case (i) of this proof, retrieve the contiguous range $LCP'[f(i')\,..\,f(i')+s]$ containing the values $LCP[LF^{\delta}(p)\,..\,LF^{\delta}(p+s)]$. Since the skipped BWT ranges are unary, it is not hard to see that $LCP[LF^{\delta}(p+j)] = LCP[p+j]+\delta$ for $j = 1, \ldots, s$ (note that we do not include $j = 0$ because we cannot exclude that, for some $k < \delta$, $LF^{k}(p)$ is the first position in its run). From the equality $\delta = SA[p]-i'-1 = SA[p]-SA[LF^{\delta}(p)]$ (that is, δ is the distance between $SA[p]$ and its predecessor minus one or, equivalently, the number of LF steps virtually performed), we then compute $LCP[p+j] = LCP'[f(i')+j]-\delta$ for $j = 1, \ldots, s$. □

As a simplification that does not change our asymptotic bounds (but that we consider in the implementation), note that it is sufficient to sample only the last (or the first) characters of BWT runs. In this case, our toehold in Lemma 3.2 will be the last cell $SA[ep]$ of our current range $SA[sp\,..\,ep]$: if $BWT[ep] = P[j]$, then the next toehold is ep' and its position is $SA[ep]-1$. Otherwise, there must be a run end (i.e., a sampled position) in $SA[sp\,..\,ep]$, which we find with $pred(R_{P[j]}, ep)$, and this stores $SA[ep']$. Therefore, we only need to store $N[i] = SA[q-1]$ in Lemma 3.5 and just P^- in Lemmas 3.7 and 3.8, thus reducing the space for sampling. This was noted simultaneously by several authors after our conference paper [Gagie et al. 2018b] and published independently [Bannai et al. 2018]. For this paper, our definition is better suited as the sampling holds crucial properties — see the next section.

4. COUNTING AND LOCATING IN OPTIMAL TIME

In this section we show how to obtain optimal counting and locating time in the unpacked — $O(m)$ and $O(m+occ)$ — and packed — $O(\lceil m\log(\sigma)/w\rceil)$ and $O(\lceil m\log(\sigma)/w\rceil + occ)$ — scenarios, by using $O(r\log\log_w(\sigma+n/r))$ and $O(rw\log_\sigma\log_w n)$ space, respectively. To improve upon the times of Theorem 3.6 we process P by chunks of s symbols on a text T^* formed by chunks as well, and resort to the faster locating of Lemma 3.7.

4.1. A RLFM-index on chunks

Given an integer $s \geq 1$, let us define texts $T^k[1\,..\,\lceil n/s\rceil]$ for $k = 0, \ldots, s-1$, so that $T^k[i] = T[k+(i-1)s+1\,..\,k+is]$, where we assume T is padded with $s-1+\lceil n/s\rceil\cdot s-n < 2s-1$ copies of \$. That is, T^k is T devoid of its first k symbols and then seen as a sequence of *metasymbols* formed by s original symbols. We then define a new text $T^* = T^0\,T^1\cdots T^{s-1}$. The text T^* has length $n^* = s\cdot\lceil n/s\rceil < n+s$ and its alphabet is of size at most σ^s. The order between the metasymbols of T^* is defined according to the lexicographic order of their corresponding length-s strings.

Note that each suffix in T^* has a corresponding suffix in T from where it is extracted.

Definition 4.1. Suffix $T^*[i^*\,..\,n^*]$ *corresponds* to suffix $T[i\,..\,n]$ iff the concatenation of the symbols forming the metasymbols in $T^*[i^*\,..\,n^*]$ is equal to the suffix $T[i\,..\,n]$, if we compare them up to the first occurrence of \$.

The next observation specifies the algebraic transformation between the positions in T^* and T.

OBSERVATION 1. *Suffix $T^*[i^*\,..\,n^*]$ corresponds to suffix $T[i\,..\,n]$ iff $i = ((i^*-1) \bmod \lceil n/s\rceil)\cdot s + \lceil i^*/\lceil n/s\rceil\rceil$.*

We exploit the property that corresponding suffixes of T and T^* have the same lexicographic rank.

LEMMA 4.2. *For any suffixes $T^*[i^*\,..\,n^*]$ and $T^*[j^*\,..\,n^*]$ corresponding to $T[i\,..\,n]$ and $T[j\,..\,n]$, respectively, it holds that $T^*[i^*\,..\,n^*] \leq T^*[j^*\,..\,n^*]$ iff $T[i\,..\,n] \leq T[j\,..\,n]$.*

PROOF. Consider any $i^* \neq j^*$, otherwise the result is trivial because $i = j$. We proceed by induction on n^*-i^*. If this is zero, then $T[i^*\,..\,n^*] = T[n^*] = T^{s-1}[\lceil n/s\rceil] = T[s-1+(\lceil n/s\rceil-1)s+1\,..\,s-1+\lceil n/s\rceil s] =$

$\$^s$ is always $\leq T[j^* \mathinner{.\,.} n^*]$ for any j^*. Further, by Observation 1, $i = \lceil n/s \rceil \cdot s$, which is the rightmost suffix of T (extended with \$s) and it is formed by all \$s, and thus it is $\leq T[j \mathinner{.\,.} n]$ for any j.

Now, given a general pair $T^*[i^* \mathinner{.\,.} n^*]$ and $T^*[j^* \mathinner{.\,.} n^*]$, consider the first metasymbols $T^*[i^*]$ and $T^*[j^*]$. If they are different, then the comparison depends on which of them is lexicographically smaller. Similarly, since $T^*[i^*] = T[i \mathinner{.\,.} i+s-1]$ and $T^*[j^*] = T[j \mathinner{.\,.} j+s-1]$, the comparison of the suffixes $T[i \mathinner{.\,.} n]$ and $T[j \mathinner{.\,.} n]$ depends on which is smaller between the substrings $T[i \mathinner{.\,.} i+s-1] \neq T[j \mathinner{.\,.} j+s-1]$. Since the metasymbols $T^*[i^*]$ and $T^*[j^*]$ are ordered lexicographically, the outcome of the comparison is the same. If, instead, $T^*[i^*] = T^*[j^*]$, then also $T[i \mathinner{.\,.} i+s-1] = T[j \mathinner{.\,.} j+s-1]$. The comparison in T^* is then decided by the suffixes $T^*[i^*+1 \mathinner{.\,.} n^*]$ and $T^*[j^*+1 \mathinner{.\,.} n^*]$, and in T by the suffixes $T[i+s \mathinner{.\,.} n]$ and $T[j+s \mathinner{.\,.} n]$. By Observation 1, the suffixes $T^*[i^*+1 \mathinner{.\,.} n^*]$ and $T^*[j^*+1 \mathinner{.\,.} n^*]$ almost always correspond to $T[i+s \mathinner{.\,.} n]$ and $T[j+s \mathinner{.\,.} n]$, and then by the inductive hypothesis the result of the comparisons is the same. The case where $T^*[i^*+1 \mathinner{.\,.} n^*]$ or $T^*[j^*+1 \mathinner{.\,.} n^*]$ do not correspond to $T[i+s \mathinner{.\,.} n]$ or $T[j+s \mathinner{.\,.} n]$ arises when i^* or j^* are a multiple of $\lceil n/s \rceil$, but in this case they correspond to some $T^k[\lceil n/s \rceil]$, which contains at least one \$. Since $i^* \neq j^*$, the number of \$s must be distinct, and then the metasymbols cannot be equal. □

An important consequence of Lemma 4.2 is that the suffix arrays SA^* and SA of T^* and T, respectively, list the corresponding suffixes in the same order (the positions of the corresponding suffixes in T^* and T differ, though). Thus we can find suffix array ranges in SA via searches on SA^*. More precisely, we can use the RLFM-index of T^* instead of that of T. The following result is the key to bound the space usage of our structure.

LEMMA 4.3. *If the BWT of T has r runs, then the BWT of T^* has $r^* = O(rs)$ runs.*

PROOF. Kempa [2019, see before Thm. 3.3] shows that the number of *s-runs* in the BWT of T, that is, the number of maximal runs of equal substrings of length s preceding the suffixes in lexicographic order, is at most $s \cdot r$. Since SA and SA^* list the corresponding suffixes in the same order, the number of s-runs in T essentially corresponds to the number of runs in T^*, formed by the length-s metasymbols preceding the same suffixes. The only exceptions are the s metasymbols that precede some metasymbol $T^k[1]$ in T^*. Other $O(s)$ runs can appear because we have padded T with $O(s)$ copies of \$, and thus T has $O(s)$ further suffixes. Still, the total is $r^* = rs + O(s) = O(rs)$. □

4.2. Mapping the alphabet

The alphabet size of T^* is σ^s, which can be large. Depending on σ and s, we could even be unable to handle the metasymbols in constant time. Note, however, that the *effective* alphabet of T^* must be $\sigma^* \leq r^* = O(rs)$, which will always be in $O(n^2)$ for any $s \leq n$. Thus we can always manage metasymbols in $[1 \mathinner{.\,.} \sigma^*]$ in constant time. We use a compact trie of height s to convert the existing substrings of length s of T into numbers in $[1 \mathinner{.\,.} \sigma^*]$, respecting the lexicographic order. The trie uses perfect hashing to find the desired child in constant time, and the strings labeling the edges are represented as pointers to an area storing all the distinct substrings of length s in T. We now show that this area is of length $O(rs)$.

Definition 4.4. We say that a text substring $T[i \mathinner{.\,.} j]$ is *primary* iff it contains at least one sampled character (see Definition 3.1).

LEMMA 4.5. *Every text substring $T[i \mathinner{.\,.} j]$ has a primary occurrence $T[i' \mathinner{.\,.} j'] = T[i \mathinner{.\,.} j]$.*

PROOF. We prove the lemma by induction on $j-i$. If $j-i=0$, then $T[i \mathinner{.\,.} j]$ is a single character, and every character has a sampled occurrence i' in the text. Now let $j-i>0$. By the inductive hypothesis, $T[i+1 \mathinner{.\,.} j]$ has a primary occurrence $T[i'+1 \mathinner{.\,.} j']$. If $T[i] = T[i']$, then $T[i' \mathinner{.\,.} j']$ is a primary occurrence of $T[i \mathinner{.\,.} j]$. Assume then that $T[i] \neq T[i']$. Let $[sp, ep]$ be the BWT range of $T[i+1 \mathinner{.\,.} j]$. Then there are two distinct symbols in $BWT[sp, ep]$, and thus there must be a run of $T[i]$'s ending or beginning in $BWT[sp, ep]$, say at position $sp \leq q \leq ep$. Thus it holds that $BWT[q] = T[i]$ and the text position $i'' = SA[q] - 1$ is sampled. We then have a primary occurrence $T[i'' \mathinner{.\,.} j''] = T[i \mathinner{.\,.} j]$. □

LEMMA 4.6. *There are at most $2rs$ distinct s-mers in the text, and they are all contained in a string of length $4rs$.*

PROOF. From Lemma 4.5, every distinct s-mer appearing in the text has a primary occurrence. It follows that, in order to count the number of distinct s-mers, we can restrict our attention to the regions of size $2s-1$ overlapping the at most $2r$ sampled positions. Each sampled position overlaps with s s-mers, so the claim easily follows. □

The compact trie then has size $O(rs)$, since it has $\sigma^* \le r^* = O(rs)$ leaves and no unary paths, and the area containing the distinct strings is also of size $O(rs)$. The structure maps any metasymbol to the new alphabet $[1\,..\,\sigma^*]$, by storing the corresponding symbol in every leaf. Every internal trie node v also stores the first and last symbols of $[1\,..\,\sigma^*]$ stored at leaves descending from it, $v_{\min}$ and $v_{\max}$.

We then build the RLFM-index of T^* on the mapped alphabet $[1\,..\,\sigma^*]$, and the structures using space proportional to the alphabet size become bounded by space $O(\sigma^*) = O(r^*)$ rather than $O(\sigma^s)$.

4.3. Counting in optimal time

Let us start with the base FM-index. Recalling Section 2.4, the FM-index of T^* consists of an array $C^*[1\,..\,\sigma^*]$ and a string $L^*[1\,..\,n^*]$, where $C^*[c]$ tells the number of times metasymbols less than c occur in T^*, and where L^* is the BWT of T^*, with the (meta)symbols mapped to $[1\,..\,\sigma^*]$.

To use this FM-index, we process P by metasymbols too. We define two patterns, $P^* \cdot L_P$ and $P^* \cdot R_P$, with $P^*[1\,..\,m^*] = P[1\,..\,s]P[s+1\,..\,2s] \cdots P[\lfloor m/s - 1 \rfloor \cdot s + 1\,..\,\lfloor m/s \rfloor \cdot s]$, $L_P = P[\lfloor m/s \rfloor \cdot s + 1\,..\,m] \cdot \$^{s-(m \bmod s)}$, and $R_P = P[\lfloor m/s \rfloor \cdot s + 1\,..\,m] \cdot @^{s-(m \bmod s)}$, @ being the largest symbol in the alphabet. That is, $P^* \cdot P_L$ and $P^* \cdot P_R$ are P padded with the smallest and largest alphabet symbols, respectively, and then regarded as a sequence of $m^* + 1 = \lfloor m/s \rfloor + 1$ metasymbols. This definition and Lemma 4.2 ensure that the suffixes of T starting with P correspond to the suffixes of T^* starting with strings lexicographically between $P^* \cdot P_L$ and $P^* \cdot P_R$.

We use the trie to map the symbols of P^* to the alphabet $[1\,..\,\sigma^*]$. If a metasymbol of P^* is not found, it means that P does not occur in T. To map the symbols L_P and R_P, we descend by the symbols $P[\lfloor m/s \rfloor \cdot s + 1\,..\,m]$ and, upon reaching trie node v, we use the precomputed limits $v_{\min}$ and $v_{\max}$ as the mappings of L_P and R_P, respectively. Overall, we map P^*, L_P and R_P in $O(m)$ time.

We can then apply backward search almost as in Section 2.4, but with a twist for the last symbols of $P^* \cdot P_L$ and $P^* \cdot P_R$: We start with the range $[sp_{m^*}, ep_{m^*}] = [C^*[v_{\min}] + 1, C^*[v_{\max} + 1]]$, and then carry out $m^* - 1$ steps, for $j = m^* - 1, \ldots, 1$, as follows, c being the mapping of $P^*[j]$:

$$\begin{aligned} sp_j &= C^*[c] + \mathrm{rank}_c(L^*, sp_{j+1} - 1) + 1, \\ ep_j &= C^*[c] + \mathrm{rank}_c(L^*, ep_{j+1}). \end{aligned}$$

The resulting range, $[sp, ep] = [sp_1, ep_1]$, corresponds to the range of P in T, and is obtained with $2(m^* - 1) \le 2m/s$ operations $\mathrm{rank}_c(L, i)$.

A RLFM-index (Section 2.5) on T^* stores, instead of C^* and L^*, structures E, L', D, and C', of total size $O(\sigma^* + r^*) = O(r^*)$. These simulate the operation $\mathrm{rank}_c(L^*, i)$ in the time of a predecessor search on E and rank and access operations on L'. These add up to $O(\log\log_w(\sigma^* + n^*/r^*))$ time. We can still retain C^* to carry out the first step of our twisted backward search on L_P and R_P in constant time, and then switch to the RLFM-index.

LEMMA 4.7. *Let $T[1\,..\,n]$, on alphabet $[1\,..\,\sigma]$, have a BWT with r runs, and let $s \le n$ be a positive integer. Then there exists a data structure using $O(rs)$ space that counts the number of occurrences of any pattern $P[1\,..\,m]$ in T in $O(m + (m/s)\log\log_w(\sigma + n/r))$. In particular, a structure using space $O(r \log\log_w(\sigma + n/r))$ counts in time $O(m)$.*

PROOF. We build the mapping trie, the RLFM-index on T^* using the mapped alphabet, and the array C^* of the FM-index of T^*. All these require $O(\sigma^* + r^*) = O(r^*)$ space, which is $O(rs)$ by Lemma 4.3. To count the number of occurrences of P, we first compute P^*, L_P, and R_P on the mapped alphabet with the trie, in time $O(m)$. We then carry out the backward search, which requires one constant-time step to find $[sp_{m^*}, ep_{m^*}]$ and then $2(m^* - 1) \le 2m/s$ steps requiring $\mathrm{rank}_c(L, i)$, which is simulated by the RLFM-index in time $O(\log\log_w(\sigma^* + n^*/r^*))$. Since $\sigma^* \le \sigma^s$, $n^* \le n + s$, and $r^* \ge r$, we can write that time as $O(\log\log_w(\sigma^s + n/r)) \subseteq O(\log s + \log\log_w(\sigma + n/r))$. The term $O(\log s)$ vanishes when multiplied by $2m/s$ because there is an $O(m)$ additive term. □

4.4. Locating in optimal time

To locate in optimal time, we will use the toehold technique of Lemma 3.2 on T^* and P^*. The only twist is that, when we look for L_P and R_P in our trie, we must store in the internal trie node v we reach by $P[\lfloor m/s \rfloor \cdot s+1\,..\,m]$ the position p in SA^*, and the value $SA^*[p]$, of some metasymbol starting with that string. From then on, we do exactly as in Lemma 3.2, so we can recover the interval $SA^*[sp, ep]$ of P^* in T^* with the values p and $SA^*[p]$ of some $sp \leq p \leq sp$. Since, by Observation 1, we can easily convert any position $SA^*[p]$ to the corresponding position $SA[p]$ in T, we have the following result.

LEMMA 4.8. *We can store $O(rs)$ words such that, given $P[1\,..\,m]$, in time $O(m + (m/s)\log\log_w(\sigma + n/r))$ we can compute the interval $SA[sp, ep]$ of the occurrences of P in T, and also return the position p and content $SA[p]$ of at least one cell in the interval $[sp, ep]$.*

We now use the structures of Lemma 3.7 on the original text T and with the same value of s. Thus, once we obtain some value $SA[p]$ within the interval, we return the occurrences in $SA[sp\,..\,ep]$ by chunks of $s' \leq s$ elements, in time $O(s' + \log\log_w(n/r))$. This allows us to retrieve the occ occurrences in time $O(occ + (1 + occ/s)\log\log_w(n/r))$.

LEMMA 4.9. *We can store $O(rs)$ words such that, given $P[1\,..\,m]$, we can count its occurrences in $O(m + (m/s)\log\log_w(\sigma + n/r))$ time and (after counting) locate them in overall time $O(occ + (1 + occ/s)\log\log_w(n/r))$.*

In particular, by choosing $s = \log\log_w(\sigma + n/r)$, the times obtained are $O(m)$ for counting and $O(m + \log\log_w(n/r) + occ)$ for locating. The additive term $O(\log\log_w(n/r))$ is relevant when $m, occ < \log\log_w(n/r)$, and it corresponds to the first predecessor search in P^+ and P^- performed when starting to locate the occurrences in Lemma 3.7. Since the term matters only when $m < s$, that is, patterns that fit in a single metasymbol, we can handle this case using the internal trie used to map metasymbols. Note that the SA range for those short patterns, corresponding to some trie node v, is $[C^*[v_{\min}] + 1, C^*[v_{\max} + 1]]$, where $v_{\min}$ and $v_{\max}$ are stored at v. We are also storing at v a position p in that range and $SA^*[p]$, which can be converted into $SA[p]$ by Observation 1. In addition, we will store the result of the predecessor search for $SA[p]$ in P^+ and P^-. As a result, the first s occurrences of short patterns are obtained without the predecessor search overhead. We then have the following result.

THEOREM 4.10. *We can store a text $T[1\,..\,n]$, over alphabet $[1\,..\,\sigma]$, in $O(r\log\log_w(\sigma + n/r))$ words, where r is the number of runs in the BWT of T, such that later, given a pattern $P[1\,..\,m]$, we can count the occurrences of P in T in $O(m)$ time and (after counting) locate their occ positions in overall time $O(occ)$.*

4.5. RAM-optimal counting and locating

We now describe how to obtain RAM-optimal time, that is, we replace m by $\lceil m\log(\sigma)/w \rceil$ in the counting and locating times. First, observe that the counting time in the previous section was $O(m + (m/s)\log s + (m/s)\log\log_w(\sigma + n/r))$, which simplified to $O(m + (m/s)\log\log(\sigma + n/r))$ because $(m/s)\log s = O(m)$. In the RAM-optimal setting, this simplification can no longer be applied since we cannot bound $\log(s)/s$ by $O(\log(\sigma)/w)$ for any value of s if σ is small. Our solution is to use Lemma 4.9 with $s = (w/\log\sigma) \cdot \log\log_w n = w\log_\sigma \log_w n$ and to use a different bound for the predecessor search time: we upper-bound $O(\log\log_w(\sigma^* + n^*/r^*))$ by $O(\log\log_w n)$, because $\sigma^* \leq r^* \leq n^* \leq n + s = O(n)$. The term $\log(s)/s$ therefore disappears and the search time $O((m/s)\log\log_w n)$ becomes the optimal $O(1 + m\log(\sigma)/w)$.

There is, however, a remaining $O(m)$ time coming from traversing the trie in order to obtain the mapped alphabet symbols of P^*, P_L, and P_R. To reduce it, we replace our trie by a more sophisticated structure, which is described by Navarro and Nekrich [2017, Sec. 2], built on the $O(rs)$ distinct strings of length s. Let $d = \lfloor w/\log\sigma \rfloor$. The structure is like our compact trie but it also stores, at selected nodes, perfect hash tables that allow descending by d symbols in $O(1)$ time. This is sufficient to find the locus of a string of length s in $O(\lceil s/d \rceil) = O(\lceil s\log(\sigma)/w \rceil)$ time, except for the last $s \bmod d$ symbols. For those, the trie also stores weak prefix search (wps) structures [Belazzougui et al. 2018] on the selected nodes, which allow descending by up to $d - 1$ symbols in constant time.

The wps structures, however, may fail if the string has no locus, so we must include a verification step. Such verification is done in RAM-optimal time by storing the strings of length $2s - 1$ extracted around sampled text positions in packed form, in our memory area associated with the edges. The space of the whole data structure is $O(1)$ words per compact trie node, so in our case it is $O(rs)$. We then map P^*, P_L, and P_R, in time $O(\lceil m\log(\sigma)/w\rceil)$.

We therefore obtain $O(m\log(\sigma)/w + \log\log_w(n/r) + occ)$ time for locating, where the middle term comes from the first predecessor search in the P^+ or P^- structures of T (not T^*). Analogously to the previous subsection, the additive term $O(\log\log_w(n/r))$ may only matter if $m\log(\sigma)/w < \log\log_w(n/r)$, which implies $m < s$, and thus it can be solved in the same way, by storing in the trie nodes the precomputed results of predecessor searches in P^+ and P^-.

THEOREM 4.11. *We can store a text $T[1\,..\,n]$, over alphabet $[1\,..\,\sigma]$, in $O(rw\log_\sigma\log_w n)$ words, where r is the number of runs in the BWT of T, such that later, given a pattern $P[1\,..\,m]$, we can count the occurrences of P in T in $O(\lceil m\log(\sigma)/w\rceil)$ time and (after counting) report their occ locations in overall time $O(occ)$.*

5. ACCESSING THE TEXT, THE SUFFIX ARRAY, AND RELATED STRUCTURES

In this section we show how we can provide direct access to the text T, the suffix array SA, its inverse ISA, and the longest common prefix array LCP. The latter operations enable functionalities that go beyond the basic counting, locating, and extracting that are required for self-indexes, and will be used to enable a full-fledged compressed suffix tree in Section 6.

We introduce a representation of T that uses $O(r\log(n/r))$ space and can retrieve any substring of length ℓ in time $O(\log(n/r) + \ell\log(\sigma)/w)$. The second term is optimal in the packed setting and, as explained in the Introduction, the $O(\log(n/r))$ additive penalty is also near-optimal in general. The structure exploits Lemma 4.5, that is, all the distinct substrings appear around phrase boundaries.

For the other arrays, we exploit the fact that the runs that appear in the BWT of T induce equal substrings in the *differential* suffix array, its inverse, and longest common prefix arrays, DSA, $DISA$, and $DLCP$, where we store the difference between each cell and the previous one. That is, an analogous of Lemma 4.5 holds on those arrays as well. Therefore, all the solutions will be variants of the one that extracts substrings of T. The extraction time in these arrays will be $O(\log(n/r) + \ell)$.

The $O(r\log(n/r))$ space we use for these structures (and for the suffix tree in Section 6) is generally higher than the $O(r\log\log_w(\sigma + n/r))$ space we used for optimal searching in Section 4. Since it may be interesting to have all the functionality in $O(r\log(n/r))$ space, we close this section showing how optimal search times can also be obtained in space $O(r\log(n/r))$, in the particular cases where this space is less than $O(r\log\log_w(\sigma + n/r))$.

5.1. Accessing T

Our structure to extract substrings of T is a variant of Block Trees [Belazzougui et al. 2015b] built around Lemma 4.5.

THEOREM 5.1. *Let $T[1\,..\,n]$ be a text over alphabet $[1\,..\,\sigma]$. We can build a data structure of $O(r\log(n/r))$ words that extracts any length-ℓ substring of T in $O(\log(n/r) + \ell\log(\sigma)/w)$ time.*

PROOF. We describe a data structure supporting the extraction of $\alpha = (w/\log\sigma)\log(n/r)$ packed characters in $O(\log(n/r))$ time. To extract a text substring of length ℓ we divide it into $\lceil \ell/\alpha\rceil$ blocks and extract each block with the proposed data structure. Overall, this will take $O((1 + \ell/\alpha)\log(n/r)) = O(\log(n/r) + \ell\log(\sigma)/w)$ time.

Our data structure is stored in $O(\log(n/r))$ levels. For simplicity, we assume that r divides n and that n/r is a power of two. The top level (level 0) is special: we divide the text into r blocks $T[1\,..\,n/r], T[n/r + 1\,..\,2n/r], \ldots, T[n - n/r + 1\,..\,n]$ of size n/r. For levels $l > 0$, we let $s_l = n/(r\cdot 2^{l-1})$ and, for every sampled position i, we consider the two non-overlapping blocks of length s_l: $X^1_{l,i} = T[i - s_l\,..\,i - 1]$ and $X^2_{l,i} = T[i\,..\,i + s_l - 1]$. Each such block $X^k_{l,i}$, for $k = 1, 2$, is composed of two half-blocks, $X^k_{l,i} = X^k_{l,i}[1\,..\,s_l/2]\, X^k_{l,i}[s_l/2{+}1\,..\,s_l]$. We moreover consider three additional consecutive and non-overlapping half-blocks, starting in the middle of the first, $X^1_{l,i}[1\,..\,s_l/2]$, and ending in the middle of the last, $X^2_{l,i}[s_l/2{+}1\,..\,s_l]$, of the 4 half-blocks just described: $T[i{-}s_l{+}s_l/4\,..\,i{-}s_l/4{-}1]$, $T[i{-}s_l/4\,..\,i{+}s_l/4{-}1]$, and $T[i{+}s_l/4\,..\,i{+}s_l{-}s_l/4{-}1]$. Figure 2 illustrates how half-blocks distribute around sampled positions.

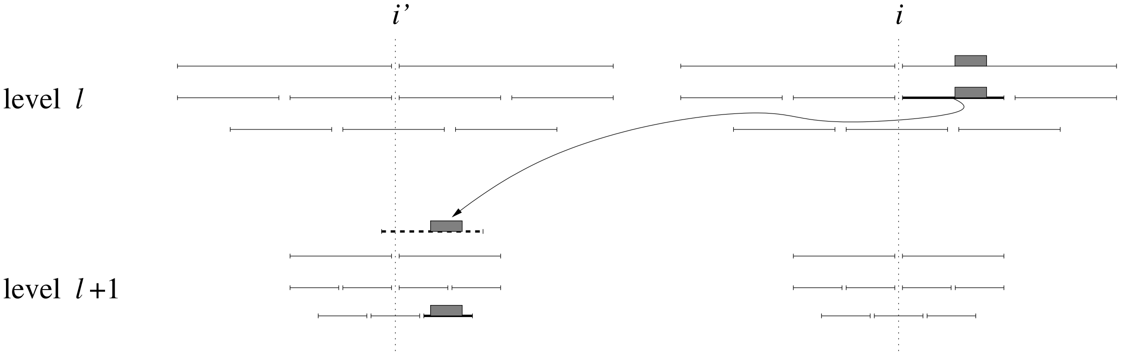

Fig. 2. Illustration of the proof of Theorem 5.1. Extracting the grayed square, we have arrived at a block around sampled position i in level l. Due to its size, the square must be contained in a half-block. This half-block (in thick line) has a copy crossing a sampled position i' (we show this copy with a dashed line). Thus the extraction task is translated to level $l+1$, inside another block of half the length. Since the square is still small enough, it must fall inside some half-block of level $l+1$ (also in thick line). This continues until the last level, where the symbols are stored directly.

From Lemma 4.5, blocks at level $l = 0$ and each half-block at level $l > 0$ have a primary occurrence covered by blocks at level $l+1$. Such an occurrence can be fully identified by the coordinate $\langle i', \mathit{off} \rangle$, where i' is a sampled position (actually we store a pointer ptr to the data associated with the sampled position i'), and $0 < \mathit{off} \le s_{l+1}$ indicates that the occurrence starts at position $i' - s_{l+1} + \mathit{off}$ of T.

Let l^* be the smallest number such that $s_{l^*} < 4\alpha = 4(w/\log\sigma)\log(n/r)$. Then l^* is the last level of our structure. We now list all the information that is explicitly stored in our data structure:

— At level l^*, we explicitly store a packed string with the characters contained in the blocks. This uses in total $O(r \cdot s_{l^*} \log(\sigma)/w) = O(r\log(n/r))$ words of space.
— The r blocks at level 0 and the seven half-blocks surrounding every sampled position at each other level $0 < l < l^* - 1$ store instead the pair of coordinates $\langle i', \mathit{off} \rangle$ of their primary occurrence in the next level. These pointers use up to $O(r \cdot l^*) = O(r\log(n/r))$ words of space.
— Finally, each half-block at level $l^* - 1$ stores a pointer to the string of explicitly stored characters (i.e. the position in that string where the half-block appears). These pointers use $O(r)$ words of space.

Let $S = T[j\,..\,j+\alpha-1]$ be the text substring to be extracted. Note that we can assume $n/r \ge \alpha$; otherwise all the text can be stored in plain packed form using $n\log(\sigma)/w < \alpha r\log(\sigma)/w = O(r\log(n/r))$ words and we do not need any data structure. It follows that S either spans two blocks at level 0, or it is contained in a single block. The former case can be solved with two queries of the latter, so we assume, without losing generality, that S is fully contained inside a block at level 0. To retrieve S, we map it down to the next levels (using the stored coordinates of primary occurrences of half-blocks) as a contiguous text substring as long as this is possible, that is, as long as it fits inside a single half-block. Note that, because of the way half-blocks overlap, this is always possible as long as $\alpha \le s_l/4$ (see Figure 2). By definition, then, we arrive in this way precisely at level l^*, where characters are stored explicitly and we can return the packed text substring. □

5.2. Accessing SA

Let us define the differential suffix array $DSA[p] = SA[p] - SA[p-1]$ for all $p > 1$, and $DSA[1] = SA[1]$. The next lemmas show that the runs of BWT induce analogous repeated substrings in DSA.

LEMMA 5.2. *Let $[p-1, p]$ be within a BWT run. Then $LF(p-1) = LF(p) - 1$ and $DSA[LF(p)] = DSA[p]$.*

PROOF. Since p is not the first position in a BWT run, it holds that $BWT[p-1] = BWT[p]$, and thus $LF(p-1) = LF(p) - 1$ follows from the formula of LF. Therefore, if $q = LF(p)$, we have $SA[q] = SA[p] - 1$ and $SA[q-1] = SA[LF(p-1)] = SA[p-1] - 1$; therefore $DSA[q] = DSA[p]$. □

LEMMA 5.3. *Let $[p-1\,..\,p+s]$ be within a BWT run, for some $1 < p \le n$ and $0 \le s \le n-p$. Then there exists $q \ne p$ such that $DSA[q\,..\,q+s] = DSA[p\,..\,p+s]$ and $[q-1\,..\,q+s]$ contains the first position of a BWT run.*

PROOF. By Lemma 5.2, it holds that $DSA[p'\,..\,p'+s] = DSA[p\,..\,p+s]$, where $p' = LF(p)$. If $DSA[p'-1\,..\,p'+s]$ contains the first position of a BWT run, we are done. Otherwise, we apply Lemma 5.2 again on $[p'\,..\,p'+s]$, and repeat until we find a range that contains the first position of a run. This search eventually terminates because there are $r > 0$ run beginnings, there are only $n-s+1$ distinct ranges, and the sequence of visited ranges, $[LF^k(p)\,..\,LF^k(p)+s]$, forms a single cycle; recall Section 2.3. Therefore our search will visit all the existing ranges before returning to $[p\,..\,p+s]$. □

This means that there exist $2r$ positions in DSA, namely those $[q, q+1]$ where $BWT[q]$ is the first position of a run, such that any substring $DSA[p\,..\,p+s]$ has a copy covering some of those $2r$ positions. This is analogous to the property proved in Lemma 4.5, which enabled efficient access on T. We now exploit it to access cells in SA by building a similar structure on DSA.

THEOREM 5.4. *Let the BWT of a text $T[1\,..\,n]$ contain r runs. Then there exists a data structure using $O(r\log(n/r))$ words that retrieves any ℓ consecutive values of its suffix array SA in time $O(\log(n/r)+\ell)$.*

PROOF. We describe a data structure supporting the extraction of $\alpha = \log(n/r)$ consecutive cells in $O(\log(n/r))$ time. To extract ℓ consecutive cells of SA, we divide it into $\lceil \ell/\alpha \rceil$ blocks and extract each block independently. This yields the promised time complexity.

Our structure is stored in $O(\log(n/r))$ levels. As before, let us assume that r divides n and that n/r is a power of two. At the top level ($l = 0$), we divide DSA into r blocks $DSA[1\,..\,n/r], DSA[n/r+1\,..\,2n/r], \ldots, DSA[n-n/r+1\,..\,n]$ of size n/r. For levels $l > 0$, we let $s_l = n/(r\cdot 2^{l-1})$ and, for every position q that starts a run in BWT, we consider the two non-overlapping blocks of length s_l: $X^1_{l,q} = DSA[q-s_l+1\,..\,q]$ and $X^2_{l,q} = DSA[q+1\,..\,q+s_l]$.[12] Each such block $X^k_{l,q}$, for $k = 1,2$, is composed of two half-blocks, $X^k_{l,q} = X^k_{l,q}[1\,..\,s_l/2]\,X^k_{l,q}[s_l/2+1\,..\,s_l]$. We moreover consider three additional consecutive and non-overlapping half-blocks, starting in the middle of the first, $X^1_{l,q}[1\,..\,s_l/2]$, and ending in the middle of the last, $X^2_{l,q}[s_l/2+1\,..\,s_l]$, of the 4 half-blocks just described: $DSA[q-s_l+s_l/4+1\,..\,q-s_l/4]$, $DSA[q-s_l/4+1\,..\,q+s_l/4]$, and $DSA[q+s_l/4+1\,..\,q+s_l-s_l/4]$.

From Lemma 5.3, blocks at level $l = 0$ and each half-block at level $l > 0$ have an occurrence covered by blocks at level $l+1$. Let the half-block X of level l (blocks at level 0 are analogous) have an occurrence containing position $q^* \in \{q, q+1\}$, where q starts a run in BWT. Then we store the pointer $\langle q^*, \mathit{off}, \delta\rangle$ associated with X, where $0 < \mathit{off} \le s_{l+1}$ indicates that the occurrence of X starts at position $q^* - s_{l+1} + \mathit{off}$ of DSA, and $\delta = SA[q-s_{l+1}] - SA[q^*-s_{l+1}+\mathit{off}-1]$ (we also store the pointer to the data structure of the half-block of level $l+1$ containing the position q^*).

Additionally, every level-0 block $X' = DSA[q'+1\,..\,q'+s_l]$ stores the value $S(X') = SA[q']$ (assume $SA[0] = 0$ throughout), and every half-block $X' = DSA[q'+1\,..\,q'+s_{l+1}/2]$ corresponding to the area $X^1_{l+1,q}X^2_{l+1,q} = DSA[q-s_{l+1}+1\,..\,q+s_{l+1}]$ stores the value $\Delta(X') = SA[q'] - SA[q-s_{l+1}]$.

Let l^* be the smallest number such that $s_{l^*} < 4\alpha = 4\log(n/r)$. Then l^* is the last level of our structure. At this level, we explicitly store the sequence of DSA cells of the areas $X^1_{l^*,q}X^2_{l^*,q}$, for each q starting a run in BWT. This uses in total $O(r\cdot s_{l^*}) = O(r\log(n/r))$ words of space. The pointers stored for the $O(r)$ blocks at previous levels also add up to $O(r\log(n/r))$ words.

Let $S = SA[p\,..\,p+\alpha-1]$ be the sequence of cells to be extracted. This range either spans two blocks at level 0, or it is contained in a single block. In the former case, we decompose it into two queries that are fully contained inside a block at level 0. To retrieve a range contained in a single block or half-block, we map it down to the next levels using the pointers from blocks and half-blocks, as a contiguous sequence as long as it fits inside a single half-block. This is always possible as long as $\alpha \le s_l/4$. By definition, then, we arrive in this way precisely to level l^*, where the symbols of DSA are stored explicitly and we can return the sequence.

[12] Note that this symmetrically covers both positions q and $q+1$; in Theorem 5.1, one extra unnecessary position is covered with $X^1_{l,q}$, for simplicity.

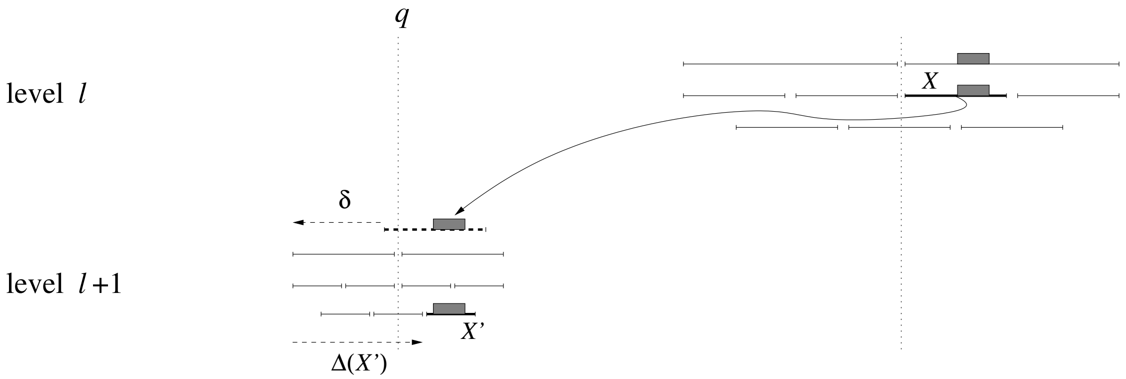

Fig. 3. Illustration of Theorem 5.4. The area to extract (a gray square) is inside the thick half-block (X), which points inside another area around position q in the next level. The sum of DSA over the offset from the beginning of the area to the mapped block (in thick dashed line) is stored in field δ of X, in negative (hence the direction of the arrow). The squared area is mapped to a smaller half-block, X', which records in $\Delta(X')$ the sum of DSA between the beginning of the area and X' (see the other dashed arrow). By adding $\delta + \Delta(X')$, we map from the first thick block to the second.

We need, however, the contents of $SA[p\,..\,p+\alpha-1]$, not of $DSA[p\,..\,p+\alpha-1]$. To obtain the former from the latter, we need only the value of $SA[p]$. During the traversal, we will maintain a value f with the invariant that, whenever the original position $DSA[p]$ has been mapped to a position $X[p']$ in the current block X, it holds that $SA[p] = f + X[1] + \ldots + X[p']$. This invariant must be maintained when we use pointers, where the original DSA values in a block X are obtained from a copy that appears elsewhere in DSA.

The invariant is initially valid by setting f to the $S(X)$ value associated with the level-0 block X that contains $SA[p]$. When we follow a pointer $\langle q^*, \mathit{off}, \delta\rangle$ from a block X and choose X', starting at $q'+1$, from the 7 half-blocks that cover the target, we update $f \leftarrow f + \delta + \Delta(X') = f + (SA[q-s_{l+1}] - SA[q^* - s_{l+1} + \mathit{off} - 1]) + (SA[q'] - SA[q - s_{l+1}]) = f + SA[q'] - SA[q^* - s_{l+1} + \mathit{off} - 1]$. This correctly adds to f the differences between the start of (the copy of) X and the start of X'. When we arrive at a block X at level l^*, we scan $O(\alpha)$ symbols until reaching the first value of the desired position $X[p']$. The values $X[1], \ldots, X[p']$ scanned are also summed to f. At the end, we have that $SA[p] = f$. See Figure 3. □

5.3. Accessing ISA and LCP

A similar method can be used to access inverse suffix array cells, $ISA[i]$. Let us define $DISA[i] = ISA[i] - ISA[i-1]$ for all $i > 1$, and $DISA[1] = ISA[1]$. The role of the runs in BWT will now be played by the phrases in ISA, which will be defined analogously as in the proof of Lemma 3.5: Phrases in ISA start at the positions $SA[p]$ such that a new run starts in $BWT[p]$ (here, last positions of runs do not start phrases). Instead of LF, we use the cycle $\phi(i)$ of Definition 3.3. We make use of the following lemmas.

LEMMA 5.5. *Let $[i-1\,..\,i]$ be within a phrase of ISA. Then it holds that $\phi(i-1) = \phi(i) - 1$ and $DISA[i] = DISA[\phi(i)]$.*

PROOF. Consider the pair of positions $T[i-1\,..\,i]$ within a phrase. Let them be pointed from $SA[p] = i$ and $SA[q] = i-1$, therefore $ISA[i] = p$, $ISA[i-1] = q$, and $LF(p) = q$. Now, since i is not a phrase beginning, p is not the first position in a BWT run. Therefore, $BWT[p-1] = BWT[p]$, from which it follows that $LF(p-1) = LF(p) - 1 = q-1$. Now let $SA[p-1] = j$, that is, $j = \phi(i)$. Then $\phi(i-1) = SA[ISA[i-1]-1] = SA[q-1] = SA[LF(p-1)] = SA[p-1] - 1 = j - 1 = \phi(i) - 1$. It also follows that $DISA[i] = p - q = DISA[j] = DISA[\phi(i)]$. □

LEMMA 5.6. *Let $[i-1\,..\,i+s]$ be within a phrase of DISA, for some $1 < i \leq n$ and $0 \leq s \leq n-i$. Then there exists $j \neq i$ such that $DISA[j\,..\,j+s] = DISA[i\,..\,i+s]$ and $[j-1\,..\,j+s]$ contains the first position of a phrase.*

PROOF. By Lemma 5.5, it holds that $DISA[i' \mathinner{.\,.} i'+s] = DISA[i \mathinner{.\,.} i+s]$, where $i' = \phi(i)$. If $DISA[i'-1 \mathinner{.\,.} i'+s]$ contains the first position of a phrase, we are done. Otherwise, we apply Lemma 5.5 again on $[i' \mathinner{.\,.} i'+s]$, and repeat until we find a range that contains the first position of a phrase. Just as in Lemma 5.2, this search eventually terminates because ϕ is a permutation with a single cycle. □

We can then use on $DISA$ exactly the same data structure we defined to access SA in Theorem 5.4, and obtain a similar result for ISA.

THEOREM 5.7. *Let the BWT of a text $T[1 \mathinner{.\,.} n]$ contain r runs. Then there exists a data structure using $O(r\log(n/r))$ words that retrieves any ℓ consecutive values of its inverse suffix array ISA in time $O(\log(n/r)+\ell)$.*

Finally, by combining Theorem 5.4 and Lemma 3.8, we also obtain access to array LCP without knowing the corresponding text positions. Note that we do not build on $DLCP$; this array and its repetitiveness properties will be used in Section 6.

THEOREM 5.8. *Let the BWT of a text $T[1 \mathinner{.\,.} n]$ contain r runs. Then there exists a data structure using $O(r\log(n/r))$ words that retrieves any ℓ consecutive values of its longest common prefix array LCP in time $O(\log(n/r)+\ell)$.*

PROOF. Build the structure of Theorem 5.4, as well as the one of Lemma 3.8 with $s = \log(n/r)$. Then, to retrieve $LCP[p \mathinner{.\,.} p+s'-1]$ for any $0 \le s' \le s$, we first compute $SA[p]$ in time $O(\log(n/r))$ using Theorem 5.4 and then, given $SA[p]$, we compute $LCP[p \mathinner{.\,.} p+s'-1]$ using Lemma 3.8 in time $O(\log\log_w(n/r)+s')$. Adding both times gives $O(\log(n/r))$.

To retrieve an arbitrary sequence of cells $LCP[p \mathinner{.\,.} p+\ell-1]$, we use the method above by chunks of s cells, plus a possibly smaller final chunk. As we use $\lceil \ell/s \rceil$ chunks, the total time is $O(\log(n/r)+\ell)$. □

5.4. Optimal counting and locating in $O(r\log(n/r))$ space

The $O(r\log(n/r))$ space we need for accessing T is not comparable with the $O(r\log\log_w(\sigma+n/r))$ space we need for optimal counting and locating. The latter is in general more attractive, because the former is better only when $r = \omega(n/\log_w^{\epsilon}\sigma)$ for any constant $\epsilon > 0$, which means that the text is not very compressible. Anyway, we show how to obtain optimal counting and locating within space $O(r\log(n/r))$.

By the discussion above, we only have to care about the case $r \ge n/\log n$. In such a case, it holds that $r\log(n/r) \ge (n\log\log n)/\log n$,[13] and thus we are allowed to use $\Theta(n\log\log n)$ bits of space. We can then make use of a result of Belazzougui and Navarro [2014, Lem. 6]. They show how we can enrich the $O(n)$-bit compressed suffix tree of Sadakane [2007] so that, using $O(n(\log t_{SA}+\log\log\sigma))$ bits, one can find the interval $SA[sp \mathinner{.\,.} ep]$ of P in time $O(m+t_{SA})$ plus the time to extract a substring of length m from T.[14] Since we provide $t_{SA} = O(\log(n/r))$ in Theorem 5.4 and extraction time $O(\log(n/r)+m\log(\sigma)/w)$ in Theorem 5.1, this arrangement uses $O(n(\log\log(n/r)+\log\log\sigma)) \subseteq O(n\log\log n)$ bits, and it supports counting in time $O(m+\log(n/r))$.

Once we know the interval, apart from counting, we can use Theorem 5.4 to obtain $SA[p]$ for any $sp \le p \le ep$ in time $O(\log(n/r))$, and then use the structure of Lemma 3.7 with $s = \log(n/r)$ to extract packs of $s' \le s$ consecutive SA entries in time $O(\log\log_w(n/r)+s') \subseteq O(\log(n/r)+s)$. Overall, we can locate the occ occurrences of P in time $O(m+\log(n/r)+occ)$.

Finally, to remove the $O(\log(n/r))$ term in the time complexities, we must speed up the searches for patterns shorter than $\log(n/r)$. We index them using a compact trie as that of Section 4.2. We store in each explicit trie node (i) the number of occurrences of the corresponding string, to support counting, and (ii) a position p where it occurs in SA, the value $SA[p]$, and the result of the predecessor queries on P^+ and P^-, as required for locating in Lemma 3.7, so that we can retrieve any number $s' \le s$ of consecutive entries of SA in time $O(s')$. By Lemma 4.6, the size of the trie and of the text substrings explicitly stored to support path compression is $O(r\log(n/r))$.

[13] Since $r\log(n/r)$ grows with r up to $r = n/e$ (with $e = 2.718...$), at which point the space is $\Theta(n)$, we obtain the lower bound by evaluating it at the smallest allowed value, $r = n/\log n$.

[14] The $O(n\log\log\sigma)$ bits of the space are not explicit in their lemma, but are required in their Section 5, which is used to prove their Lemma 6.

Table III. Suffix tree operations.

Operation	Description
Root()	Suffix tree root.
Locate(v)	Text position i of leaf v.
Ancestor(v, w)	Whether v is an ancestor of w.
SDepth(v)	String depth for internal nodes, i.e., length of string represented by v.
TDepth(v)	Tree depth, i.e., depth of tree node v.
Count(v)	Number of leaves in the subtree of v.
Parent(v)	Parent of v.
FChild(v)	First child of v.
NSibling(v)	Next sibling of v.
SLink(v)	Suffix-link, i.e., if v represents $a \cdot \alpha$ then the node that represents α, for $a \in [1 \mathinner{..} \sigma]$.
WLink(v, a)	Weiner-link, i.e., if v represents α then the node that represents $a \cdot \alpha$.
$SLink^i(v)$	Iterated suffix-link.
LCA(v, w)	Lowest common ancestor of v and w.
Child(v, a)	Child of v by letter a.
Letter(v, i)	The ith letter of the string represented by v.
$LAQ_S(v, d)$	String level ancestor, i.e., the highest ancestor of v with string-depth $\geq d$.
$LAQ_T(v, d)$	Tree level ancestor, i.e., the ancestor of v with tree-depth d.

THEOREM 5.9. *We can store a text $T[1 \mathinner{..} n]$, over alphabet $[1 \mathinner{..} \sigma]$, in $O(r \log(n/r))$ words, where r is the number of runs in the BWT of T, such that later, given a pattern $P[1 \mathinner{..} m]$, we can count the occurrences of P in T in $O(m)$ time and (after counting) report their occ locations in overall time $O(occ)$.*

6. A RUN-LENGTH COMPRESSED SUFFIX TREE

In this section we show how to implement a compressed suffix tree within $O(r \log(n/r))$ words, which supports a large set of navigation operations in time $O(\log(n/r))$. The only exceptions are going to a child by some letter and performing level ancestor queries, which cost $O(\log(n/r) \log \sigma)$ and up to $O(\log(n/r) \log n)$, respectively. The first compressed suffix tree for repetitive collections was built on runs [Mäkinen et al. 2010], but just like the self-index, it needed $\Theta(n/s)$ space to obtain $O(s \log n)$ time in key operations like accessing SA. Other compressed suffix trees for repetitive collections appeared later [Abeliuk et al. 2013; Navarro and Ordóñez 2016; Farruggia et al. 2018; Cáceres and Navarro 2019], but they do not offer formal space guarantees (see later). The only one offering time guarantees uses $O(\bar{e})$ words and supports a number of operations in time typically $O(\log n)$ [Belazzougui and Cunial 2017b]. Their space measure is not comparable with $O(r \log(n/r))$.

6.1. Compressed suffix trees without storing the tree

Fischer et al. [2009] showed that a rather complete suffix tree functionality, consisting of all the operations in Table III, can be efficiently supported by a representation where suffix tree nodes v are identified with the suffix array intervals $SA[v_l \mathinner{..} v_r]$ they cover. Their representation builds on the following primitives:

(1) Access to arrays SA and ISA, in time we call t_{SA}.
(2) Access to array LCP, in time we call t_{LCP}.
(3) Three special queries on LCP:
  (a) Range Minimum Query,

$$\mathrm{RMQ}(p, q) \;=\; \arg\min_{p \leq k \leq q} LCP[k],$$

  choosing the smallest position upon ties, in time we call t_{RMQ}.
  (b) Previous/Next Smaller Value queries,

$$\begin{aligned} \mathrm{PSV}(p) &= \max(\{q < p, LCP[q] < LCP[p]\} \cup \{0\}), \\ \mathrm{NSV}(p) &= \min(\{q > p, LCP[q] < LCP[p]\} \cup \{n+1\}), \end{aligned}$$

  in time we call t_{SV}.

An interesting finding of Fischer et al. [2009] related to our results is that the array $PLCP$, which stores the LCP values in text order, can be stored in $O(r)$ words and accessed efficiently; therefore we can compute any LCP value in time t_{SA} (see also Fischer [2010]). We obtained a generalization of this property in Section 3.2. Fischer et al. [2009] also show how to represent the array $TDE[1..n]$, where $TDE[i]$ is the tree-depth of the lowest common ancestor of the $(i-1)$th and ith suffix tree leaves (and $TDE[1] = 0$). They represent its values in text order in an array $PTDE$, which just like $PLCP$ can be stored in $O(r)$ words and accessed efficiently, thereby giving access to TDE in time t_{SA}. They use TDE to compute operations ***TDepth*** and ***LAQ***$_T$ efficiently.

Abeliuk et al. [2013] show that primitives RMQ, PSV, and NSV can be implemented using a simplified variant of *range min-Max trees (rmM-trees)* [Navarro and Sadakane 2014], consisting of a perfect binary tree on top of LCP where each node stores the minimum LCP value in its subtree. The three primitives are then computed in logarithmic time. They define the extended primitives

$$\begin{aligned}\mathrm{PSV}'(p,d) &= \max(\{q < p, LCP[q] < d\} \cup \{0\}),\\ \mathrm{NSV}'(p,d) &= \min(\{q > p, LCP[q] < d\} \cup \{n+1\}),\end{aligned}$$

and compute them in time $t_{\mathrm{SV}'}$, which in their setting is the same t_{SV} of the basic PSV and NSV primitives. The extended primitives are used to simplify some of the operations of Fischer et al. [2009].

The resulting time complexities are given in the second column of Table IV, where t_{LF} is the time to compute function LF or its inverse, or to access a position in BWT. Operation ***WLink***, not present in Fischer et al. [2009], is trivially obtained with two LF-steps. We note that most times appear multiplied by t_{LCP} in Fischer et al. [2009] because their RMQ, PSV, and NSV structures do not store LCP values inside, so they need to access the array all the time; this is not the case when we use rmM-trees. The time of ***LAQ***$_S$ is due to improvements obtained with the extended primitives PSV' and NSV' [Abeliuk et al. 2013].[15] The time for ***Child***(v,a) is obtained by binary searching among the σ minima of $LCP[v_l, v_r]$, and extracting the desired letter (at position ***SDepth***$(v)+1$) to compare with a. Each binary search operation can be done with an extended primitive $\mathrm{RMQ}'(p,q,m)$ that finds the mth left-to-right occurrence of the minimum in a range. This is easily done in $t_{\mathrm{RMQ}'} = t_{\mathrm{RMQ}}$ time on a rmM-tree by storing, in addition, the number of times the minimum of each node occurs below it [Navarro and Sadakane 2014], but it may be not so easy to do on other structures. Finally, the complexities of ***TDepth*** and ***LAQ***$_T$ make use of array TDE. While Fischer et al. [2009] use an RMQ operation to compute ***TDepth***, we note that ***TDepth***$(v) = 1 + \max(TDE[v_l], TDE[v_r+1])$, because the suffix tree has no unary nodes (they used this simpler formula only for leaves).[16]

An important idea of Abeliuk et al. [2013] is that they represent LCP differentially, that is, the array $DLCP[1..n]$, where $DLCP[i] = LCP[i] - LCP[i-1]$ if $i > 1$ and $DLCP[1] = LCP[1]$, using a *context-free grammar (CFG)*. Further, they store the rmM-tree information in the nonterminals, that is, a nonterminal X expanding to a substring $D = DLCP[p..q]$ stores the (relative) minimum

$$m(X) \;=\; \min_{1\le k\le |D|} \sum_{i=1}^{k} D[i] \;=\; \min_{p\le k\le q} \sum_{i=p}^{k} DLCP[i] \;=\; \left(\min_{p\le k\le q} LCP[k]\right) - LCP[p-1]$$

of any LCP segment having those differential values, and its position inside the segment,

$$p(X) \;=\; \arg\min_{1\le k\le |D|} \sum_{i=1}^{k} D[i] \;=\; \left(\arg\min_{p\le k\le q} LCP[k]\right) - (p-1).$$

Thus, instead of a perfect rmM-tree, they conceptually use the parse tree as an rmM-tree. They show how to adapt the algorithms on the perfect rmM-tree to run on the grammar, and thus solve primitives RMQ, PSV', and NSV', in time proportional to the grammar height.

[15] They also use these primitives for ***NSibling***, mentioning that the original formula has a bug. Since we obtain better t_{RMQ} than $t_{\mathrm{SV}'}$ time, we rather prefer to fix the original bug [Fischer et al. 2009]. The formula fails for the penultimate child of its parent. To compute the next sibling of $[v_l, v_r]$ with parent $[w_l, w_r]$, the original formula $[v_r+1, u]$ with $u = \mathrm{RMQ}(v_r+2, w_r) - 1$ (used only if $v_r < w_r - 1$) must now be checked as follows: if $u < w_r$ and $LCP[v_r+1] \neq LCP[u+1]$, then correct it to $u = w_r$.

[16] We observe that ***LAQ***$_T$ can be solved exactly as ***LAQ***$_S$, with the extended $\mathrm{PSV}'/\mathrm{NSV}'$ operations, now defined on the array TDE instead of on LCP. However, an equivalent to Lemma 6.2 for the differential TDE array does not hold, and therefore we cannot use that solution within the desired space bounds.

Table IV. Complexities of suffix tree operations. *Letter*(v, i) can also be solved in time $O(i \cdot t_{LF}) = O(i \log\log_w(n/r))$.

Operation	Generic Complexity	Our Complexity
Root()	1	1
Locate(v)	t_{SA}	$\log(n/r)$
Ancestor(v, w)	1	1
SDepth(v)	$t_{\mathrm{RMQ}} + t_{LCP}$	$\log(n/r)$
TDepth(v)	t_{SA}	$\log(n/r)$
Count(v)	1	1
Parent(v)	$t_{LCP} + t_{\mathrm{SV}}$	$\log(n/r)$
FChild(v)	t_{RMQ}	$\log(n/r)$
NSibling(v)	$t_{LCP} + t_{\mathrm{RMQ}}$	$\log(n/r)$
SLink(v)	$t_{LF} + t_{\mathrm{RMQ}} + t_{\mathrm{SV}}$	$\log(n/r)$
WLink(v)	t_{LF}	$\log\log_w(n/r)$
SLink$^i(v)$	$t_{SA} + t_{\mathrm{RMQ}} + t_{\mathrm{SV}}$	$\log(n/r)$
LCA(v, w)	$t_{\mathrm{RMQ}} + t_{\mathrm{SV}}$	$\log(n/r)$
Child(v, a)	$t_{LCP} + (t_{\mathrm{RMQ}'} + t_{SA} + t_{LF}) \log\sigma$	$\log(n/r)\log\sigma$
Letter(v, i)	$t_{SA} + t_{LF}$	$\log(n/r)$
LAQ$_S(v, d)$	$t_{\mathrm{SV}'}$	$\log(n/r) + \log\log r$
LAQ$_T(v, d)$	$(t_{\mathrm{RMQ}} + t_{LCP}) \log n$	$\log(n/r)\log n$

Abeliuk et al. [2013], and also Fischer et al. [2009], claim that the grammar produced by RePair [Larsson and Moffat 2000] is of size $O(r\log(n/r))$. This is an incorrect result borrowed from González and Navarro [2007] (also in González et al. [2014]), where it was claimed for DSA. The proof fails for a reason we describe in our technical report [Gagie et al. 2017, App. A].

We now start by showing how to build a grammar of size $O(r\log(n/r))$ and height $O(\log(n/r))$ for $DLCP$. This grammar is of an extended type called *run-length context-free grammar (RLCFG)* [Nishimoto et al. 2015], which allows rules of the form $X \to Y^t$ that count as size 1. We then show how to implement the operations RMQ and NSV/PSV in time $O(\log(n/r))$ on the resulting RLCFG, and $\mathrm{NSV}'/\mathrm{PSV}'$ in time $O(\log(n/r) + \log\log_w r)$. Finally, although we cannot implement RMQ' in time below $\Theta(\log n)$, we show how the specific *Child* operation can be implemented in time $O(\log(n/r)\log\sigma)$.

Note that, although we could represent $DLCP$ using a Block-Tree-like structure as we did in Section 5 for DSA and $DISA$, we have not devised a way to implement the more complex operations we need on $DLCP$ using such a Block-Tree-like data structure within polylogarithmic time.

Using the results we obtain in this and previous sections, that is, $t_{SA} = O(\log(n/r))$, $t_{LF} = O(\log\log_w(n/r))$, $t_{LCP} = t_{SA} + O(\log\log_w(n/r)) = O(\log(n/r))$, $t_{\mathrm{RMQ}} = t_{\mathrm{SV}} = O(\log(n/r))$, $t_{SV'} = O(\log(n/r) + \log\log_w r)$, and our specialized algorithm for *Child*, we obtain our result.

THEOREM 6.1. *Let the BWT of a text $T[1\,..\,n]$, over alphabet $[1\,..\,\sigma]$, contain r runs. Then a compressed suffix tree on T can be represented using $O(r\log(n/r))$ words, and it supports the operations with the complexities given in the third column of Table IV.*

6.2. Representing $DLCP$ with a run-length grammar

In this section we show that the differential array $DLCP$ can be represented by a RLCFG of size $O(r\log(n/r))$. We first prove a lemma analogous to those of Section 5.

LEMMA 6.2. *Let $[p-2, p]$ be within a BWT run. Then $LF(p-1) = LF(p) - 1$ and $DLCP[LF(p)] = DLCP[p]$.*

PROOF. Let $i = SA[p]$, $j = SA[p-1]$, and $k = SA[p-2]$. Then $LCP[p] = lcp(T[i\,..], T[j\,..])$ and $LCP[p-1] = lcp(T[j\,..], T[k\,..])$. We know from Lemma 5.2 that, if $q = LF(p)$, then $LF(p-1) = q-1$ and $LF(p-2) = q-2$. Also, $SA[q] = i-1$, $SA[q-1] = j-1$, and $SA[q-2] = k-1$. Therefore, $LCP[LF(p)] = LCP[q] = lcp(T[SA[q]\,..], T[SA[q-1]\,..) = lcp(T[i-1\,..], T[j-1\,..])$. Since p is not the first position in a BWT run, it holds that $T[j-1] = BWT[p-1] = BWT[p] = T[i-1]$, and thus $lcp(T[i-1\,..], T[j-1\,..]) = 1 + lcp(T[i\,..], T[j\,..]) = 1 + LCP[p]$. Similarly, $LCP[LF(p)-1] = LCP[q-1] = lcp(T[SA[q-1]\,..], T[SA[q-2]\,..) = lcp(T[j-1\,..], T[k-1\,..])$. Since $p-1$ is not the first position in a BWT run, it holds that $T[k-1] = BWT[p-2] = BWT[p-1] = T[j-1]$, and thus $lcp(T[j-1\,..], T[k-1\,..]) =$

$1 + lcp(T[j\,..], T[k\,..]) = 1 + LCP[p-1]$. Therefore $DLCP[q] = LCP[q] - LCP[q-1] = (1 + LCP[p]) - (1 + LCP[p-1]) = DLCP[p]$. $\square$

It follows that, if there are r runs in BWT, then we can define a *bidirectional macro scheme* [Storer and Szymanski 1982] of size $O(r)$ on $DLCP$ (in fact, the same holds for T, DSA, and $DISA$).

Definition 6.3. A *bidirectional macro scheme (BMS)* of size b on a sequence $S[1\,..\,n]$ is a partition $S = S_1 \cdots S_b$ such that each S_t is of length 1 (and is represented as an explicit symbol) or it appears somewhere else in S (and is represented by a pointer to that other occurrence). Let $f(i)$, for $1 \le i \le n$, be defined arbitrarily if $S[i]$ is an explicit symbol, and $f(i) = j + i' - 1$ if $S[i] = S_t[i']$ is inside some S_t that is represented as a pointer to $S[j\,..\,j']$. A correct BMS must satisfy that, for any i, there is a $k \ge 0$ such that $f^k(i)$ is an explicit symbol.

Note that $f(i)$ maps the position $S[i]$ to the source from which it is to be obtained. The last condition then ensures that we can recover any symbol $S[i]$ by following the chain of copies until finding an explicitly stored symbol. Finally, note that all the f values inside a block are consecutive: if $S_t = S[i\,..\,i']$ has a pointer to $S[j\,..\,j']$, then $f([i\,..\,i']) = [j\,..\,j']$.

LEMMA 6.4. *Let $p_1 < \cdots < p_r$ be the positions that start runs in BWT, and assume $p_0 = -2$ and $p_{r+1} = n + 1$. Then, the partition formed by (1) all the explicit symbols $DLCP[p_i + k]$ for $1 \le i \le r$ and $k \in \{0, 1, 2\}$, and (2) all the nonempty regions $DLCP[p_i + 3\,..\,p_{i+1} - 1]$ for all $0 \le i \le r$, pointing to $DLCP[LF(p_i + 3)\,..\,LF(p_{i+1} - 1)]$, is a BMS of size at most $4r + 1$.*

PROOF. By Lemma 6.2, it holds that $LF(p_i + 3 + k) = LF(p_i + 3) + k$ and $DLCP[p_i + 3 + k] = DLCP[LF(p_i + 3) + k]$ for all $0 \le k \le p_{i+1} - p_i - 4$, so the partition is well defined and the copies are correct. To see that it is a BMS, it is sufficient to notice that LF is a permutation with one cycle on $[1\,..\,n]$, and therefore $LF^k(p)$ will eventually reach an explicit symbol, for some $0 \le k < n$. $\square$

We now make use of the following result.

LEMMA 6.5 ([GAGIE ET AL. 2018A, THM. 1]). *Let $S[1\,..\,n]$ have a BMS of size b. Then there exists a RLCFG of size $O(b \log(n/b))$ that generates S.*

Since $DLCP$ has a BMS of size at most $4r + 1$, the following corollary is immediate.

LEMMA 6.6. *Let the BWT of $T[1\,..\,n]$ have r runs. Then there exists a RLCFG of size $O(r \log(n/r))$ that generates its differential LCP array, $DLCP$.*

6.3. Supporting the primitives on the run-length grammar

We describe how to compute the primitives RMQ and PSV/NSV on the RLCFG of $DLCP$, in time $t_{\mathrm{RMQ}} = t_{\mathrm{SV}} = O(\log(n/r))$. The extended primitives PSV′/NSV′ are solved in time $t_{SV'} = O(\log(n/r) + \log\log_w r)$. While analogous procedures have been described before on CFGs and trees [Abeliuk et al. 2013; Navarro and Sadakane 2014], the extension to RLCFGs and the particular structure of our grammar requires a complete description.

The RLCFG built in Lemma 6.5 [Gagie et al. 2018a] is of height $O(\log(n/r))$ and has one initial rule $S \to X_1 \cdots X_{O(r)}$. The other rules are of the form $X \to Y_1 Y_2$ or $X \to Y^t$ for $t > 2$. All the right-hand symbols can be terminals or nonterminals.

The data structure we use is formed by a sequence $DLCP' = X_1 \cdots X_{O(r)}$ capturing the initial rule of the RLCFG, and an array of the other $O(r \log(n/r))$ rules. For each nonterminal X expanding to a substring $D = DLCP[p\,..\,q]$, we store its length $l(X)$ and its total difference $d(X)$:

$$\begin{aligned} l(X) &= |D| = q - p + 1 \\ d(X) &= D[1] + \cdots + D[l(X)] = LCP[q] - LCP[p-1]. \end{aligned}$$

For terminals X, we assume $l(X) = 1$ and $d(X) = X$. We also store a cumulative length array $L[0] = 0$ and $L[x] = L[x-1] + l(X_x)$ that can be binary searched to find the symbol of $DLCP'$ that contains any desired position $DLCP[p]$. To ensure that this binary search takes time $O(\log(n/r))$ when $r = \omega(n/r)$, we can store a sampled array of positions $S[1\,..\,r]$, where $S[t] = x$ if $L[x-1] < t \cdot (n/r) \le L[x]$ to narrow

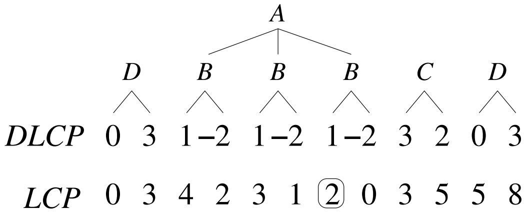

Fig. 4. Example LCP and $DLCP$ arrays, with a grammar built on $DLCP$ as follows: $D \to 03$, $B \to 1(-2)$, $C \to 32$, $A \to B^3$, and $S \to DACD$, so $DLCP'[1\,..\,4] = DACD$. Our arrays are $L[0\,..\,4] = \langle 0,2,8,10,12\rangle$ and $A[0\,..\,4] = \langle 0,3,0,5,8\rangle$. Further, $l(D) = l(B) = l(C) = 2$, $l(A) = 6$, $l(S) = 12$, $d(D) = 3$, $d(B) = -1$, $d(C) = 5$, $d(A) = -3$, $d(S) = 8$. To compute $LCP[p = 7]$, we first binary search L to find that $x = 2$ satisfies $2 = L[x-1] < p \le L[x] = 8$; therefore we must look inside $DLCP'[x] = A$ with local offset $p \leftarrow p - L[x-1] = 5$ and with initial value $f \leftarrow A[x-1] = 3$. Since $A \to B^3$ and $l(B) = 2$, the position $p = 5$ must be inside the $\lceil l(B)/p \rceil = 3$rd B. We then skip $t' = \lceil l(B)/p \rceil - 1 = 2$ copies of B with $p \leftarrow p - t' \cdot l(B) = 1$ and $f \leftarrow f + t' \cdot d(B) = 1$. We now enter into B with offset $p = 1$. Since $B \to 1(-2)$ and $p \le l(1) = 1$, we enter into the 1. Since this is a terminal symbol, we just answer $f + d(1) = 1 + 1 = 2 = LCP[7]$. For RMQs, we additionally store the values $m(D) = 0$, $m(B) = -1$, $m(C) = 3$, $m(A) = -3$, $m(S) = 0$, $p(D) = 1$, $p(B) = 2$, $p(C) = 1$, $p(A) = 6$, $p(S) = 1$, and the array $M[1\,..\,4] = \langle 0,0,3,5\rangle$.

down the binary search to a range of $O(n/r)$ entries of L. Finally, we store a cumulative differences array $A[0] = 0$ and $A[x] = A[x-1] + d(X_x)$. Note $A[x] = LCP[L[x]]$ for all $x > 0$.

Accessing LCP. Although we have already provided access to any $LCP[p]$ in Section 5.3, it is also possible to do it with these structures, and it is illustrative for some more complex operations that follow: We first find x by binary searching L for p, possibly with the help of S, so that $L[x-1] < p \le L[x]$. The position p is then inside the symbol $X_x = DLCP'[x]$, which expands to the substring $D_x = DLCP[L[x-1]+1\,..\,L[x]]$. The local offset of p inside X_x is $p - L[x-1]$. It then holds that

$$LCP[p] \ = \ A[x-1] + LCP[p] - LCP[L[x-1]] \ = \ A[x-1] + D_x[1] + \cdots + D_x[p - L[x-1]].$$

Thus we set $p \leftarrow p - L[x-1]$ as the local offset sought inside D_x and set $f \leftarrow A[x-1]$ to initialize our cumulative computation. We then enter recursively into nonterminal $X = X_x$. If its rule is $X \to Y_1 Y_2$, we continue by Y_1 if $p \le l(Y_1)$; otherwise position p is inside Y_2. Before continuing by Y_2, however, we must first skip Y_1: we set $f \leftarrow f + d(Y_1)$ and $p \leftarrow p - l(Y_1)$.

If, instead, the rule is $X \to Y^t$, we must simulate entering into the $\lceil p/l(Y) \rceil$th copy of Y: we compute $t' = \lceil p/l(Y) \rceil$, set $f \leftarrow f + (t'-1) \cdot d(Y)$, $p \leftarrow p - (t'-1) \cdot l(Y)$, and continue by Y. When we finally arrive at a terminal X, the answer is $f + d(X)$.

All this process takes time $O(\log(n/r))$, the height of the RLCFG. Figure 4 illustrates it.

Answering RMQ. To answer this query, we store a few additional structures. We define an array

$$M[x] \ = \min_{L[x-1] < k \le L[x]} LCP[k] \ = \ LCP[L[x-1]] + m(X_x) \ = \ A[x-1] + m(X_x),$$

of size $O(r)$ with the minimum value in the area of LCP expanded by $X_x = DLCP'[x]$. Note that the leftmost position in that area where the minimum $M[x]$ is reached is $L[x-1] + p(X_x)$. We do not need to store M but just a succinct data structure RMQ_M, which requires just $O(r)$ bits and finds the leftmost position of a minimum in any range $M[x\,..\,y]$ in constant time, without need to access M [Fischer and Heun 2011]. We also store, for each nonterminal X, the values $m(X)$ and $p(X)$ (for terminals X, we can store $m(X)$ and $p(X)$ or compute them on the fly). If we wanted to find $\mathrm{RMQ}(p,q)$ on LCP where $LCP[p\,..\,q]$ is exactly aligned with $DLCP'[x\,..\,y]$ (i.e., $DLCP'[x\,..\,y]$ expands to $DLCP[p\,..\,q]$), then the answer would simply be $L[z-1] + p(X_z)$, with $z = \mathrm{RMQ}_M(x,y)$.

To compute general $\mathrm{RMQ}(p,q)$ queries on LCP, we first use L and S to find x and y such that $DLCP[p\,..\,q]$ contains the expansion of $DLCP'[x+1\,..\,y-1]$, whereas $DLCP'[x\,..\,y]$ expands to $DLCP[p'\,..\,q']$ with $p' < p \le q < q'$. Therefore, $DLCP[p\,..\,q]$ is formed by three parts, each of which can be empty: (i) a leftmost part that partially overlaps the expansion of $DLCP'[x]$, (ii) a central part that corresponds to the whole expansion of $DLCP[x+1\,..\,y-1]$, and (iii) a rightmost part that partially overlaps the expansion of $DLCP'[y]$. We first obtain in constant time the minimum position of the central part, $z = \mathrm{RMQ}_M(x+1, y-1)$, and then the minimum value in area (ii) is $A[z-1] + m(X_z)$, as explained. To complete the query, we must compare this value with the minima in $X_x\langle p - p' + 1, l(X_x)\rangle$ (area (i))

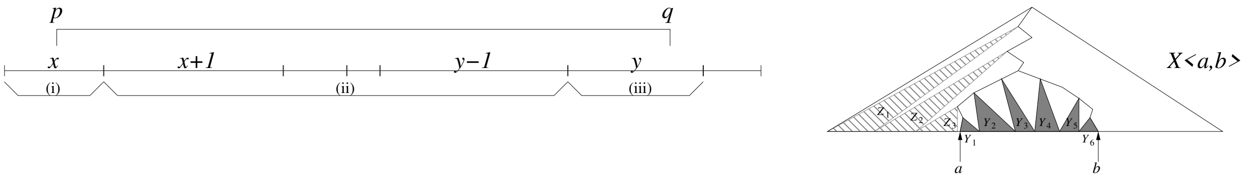

Fig. 5. A scheme of the way $\mathrm{RMQ}(p, q)$ queries are handled.

and $X_y\langle 1, l(X_y) + q - q'\rangle$ (area (iii)), where $X\langle a, b\rangle$ refers to the substring $D[a\,..\,b]$ in the expansion D of X. A relevant special case in this scheme is that the whole $DLCP[p\,..\,q]$ can be inside a single symbol $DLCP'[x]$ expanding to $DLCP[p'\,..\,q']$, in which case the query boils down to finding the minimum value in $X_x\langle p - p' + 1, l(X_x) + q - q'\rangle$.

We now describe how to find the minimum in $X_w\langle a, b\rangle$. Let us disregard the rules $X \to Y^t$ for a moment. Similarly as done for accessing LCP, we descend by the rules that generate the expansion of X_w towards the positions a and b of the expansion of X_w. As we descend, we identify the $O(\log(n/r))$ maximal nodes $Z_1, \ldots, Z_{k'}$ of the grammar tree that cover the range $[1\,..\,a-1]$ (i.e., the node of Z_1 every time we descend from $Z \to Z_1 Z_2$ towards Z_2 in the path to a). We also identify the $O(\log(n/r))$ maximal nodes $Y_1, \ldots, Y_k$ that cover the range $[a\,..\,b]$ (i.e., after we reach the lowest node Y^* shared by the paths towards a and b, we collect the nodes Y_2 whenever we descend from $Y \to Y_1 Y_2$ towards Y_1 in the path to a, and the nodes Y_1 whenever we descend from $Y \to Y_1 Y_2$ towards Y_2 in the path to b).

We then find the minimum among $m(Y_1)$, $d(Y_1) + m(Y_2)$, $d(Y_1) + d(Y_2) + m(Y_3)$, ..., in $O(k)$ time. Once the minimum value $d(Y_1) + \cdots + d(Y_{s-1}) + m(Y_s)$ is identified, we obtain its absolute value by adding $A[w-1] + d(Z_1) + \cdots + d(Z_{k'})$. The absolute position of that minimum is $L[w-1] + l(Z_1) + \cdots + l(Z_{k'}) + l(Y_1) + \cdots + l(Y_{s-1}) + p(Y_s)$.

Our grammar also has rules of the form $X \to Y^t$, and thus the maximal coverage $Y_1, \ldots, Y_k$ may include a part of these rules, say $Y^{t'}$ for some $1 \le t' < t$. We can then compute $m(Y^{t'})$ in constant time, as follows. If $d(Y) > 0$, then the minimum of $m(Y), d(Y) + m(Y), d(Y) + d(Y) + m(Y), \ldots$ is clearly $m(Y)$, that is, the minimum occurs in the first copy of Y. If $d(Y) = 0$, then the minimum occurs in every copy of Y, but the leftmost is still in the first copy. In both cases, then, it holds that $m(Y^{t'}) = m(Y)$ and $p(Y^{t'}) = p(Y)$. Instead, if $d(Y) < 0$, the minimum is $(t'-1) \cdot d(Y) + m(Y)$, which occurs in the last copy of Y, and therefore we have $m(Y^{t'}) = (t'-1) \cdot d(Y) + m(Y)$ and $p(Y^{t'}) = (t'-1) \cdot l(Y) + p(Y)$. We may also need to compute $l(Y^{t'}) = t' \cdot l(Y)$ and $d(Y^{t'}) = t' \cdot d(Y)$; these computations may also be needed if run-length rules appear in the sequence $Z_1, \ldots, Z_{k'}$.

Once we have the (up to) three minima from the cases (i), (ii), and (iii), the absolute position of the smallest of the absolute values is $\mathrm{RMQ}(p, q)$. The total time is $t_{\mathrm{RMQ}} = O(\log(n/r))$. Figure 5 shows a scheme of the process and Figure 4 gives some example values $m(\cdot)$, $p(\cdot)$, and array M.

Answering $\mathrm{PSV}/\mathrm{NSV}$ *and* $\mathrm{PSV}'/\mathrm{NSV}'$. These queries are solved analogously as RMQs. We describe $\mathrm{NSV}'(p, d)$, since $\mathrm{PSV}'(p, d)$ is similar. Let $DLCP[p\,..]$ be included in the expansion of $DLCP'[x\,..]$, which expands to $DLCP[p'\,..]$ (for the largest possible $p' \le p$). We subtract $LCP[p-1] = A[x-1] + d(Z_1) + \cdots + d(Z_{k'})$ from d to put it in relative form, where as before the nonterminals Z_i cover $X\langle 1, a-1\rangle$. We now consider $X_x\langle p - p' + 1, l(X_x)\rangle = X\langle a, b\rangle$, obtaining as before the $O(\log(n/r))$ maximal nonterminals $Y_1, Y_2, \ldots, Y_k$ that cover $X\langle a, b\rangle$, and find the first Y_s where $d(Y_1) + \cdots + d(Y_{s-1}) + m(Y_s) < d$. We then subtract $d(Y_1) + \cdots + d(Y_{s-1})$ from d, add $l(Y_1) + \cdots + l(Y_{s-1})$ to p, and continue recursively inside Y_s to find the precise point where the cumulative differences fall below d.

The recursive traversal from Y_s works as follows. If $Y_s \to Y_1 Y_2$, we first see if $m(Y_1) < d$. If so, we continue recursively on Y_1; otherwise, we subtract $d(Y_1)$ from d, add $l(Y_1)$ to p, and continue recursively on Y_2. If, instead, the rule is $Y_s \to Y^t$, we proceed as follows. If $d(Y) \ge 0$, then the answer, if any, must lie inside the first copy of Y, because as seen before the first copy of Y contains an occurrence of the smallest value of Y_s. Thus, we recursively continue on Y. If $d(Y) < 0$, instead, we must find the copy t' inside which the cumulative differences fall below d. This is the smallest t' such that $(t'-1) \cdot d(Y) + m(Y) < d$, that is, $t' = \max(1, 2 + \lfloor (d - m(Y))/d(Y) \rfloor)$. Thus we subtract $(t'-1) \cdot d(Y)$ from d, add $(t'-1) \cdot l(Y)$ to p, and continue with Y. Finally, when we arrive at a terminal X, it holds that $m(X) < d$

and the answer to the query is the current value of p. All of this process takes time $O(\log(n/r))$, the height of the grammar.

It might be, however, that we traverse $Y_1, Y_2, \ldots, Y_k$, that is, the whole $X_x\langle p-p'+1, l(X_x)\rangle$, and still do not find a value below d. We then must find where we fall below (the current value of) d inside $DLCP'[x+1\,..]$. Once this search identifies the leftmost position $DLCP'[z]$ where the answer lies, we complete the search on $X_z\langle 1, l(X_z)\rangle$ as before, for $d \leftarrow d - A[z-1] + A[x]$.

The search problem can be regarded as follows: Given the array $B[z] = A[z]+m(X_z)$, find the leftmost position $z > x$ such that $B[z] < A[x] + d$. Navarro and Sadakane [2014, Sec. 5.1] show that this query can be converted into a weighted ancestor query on a tree: given nodes with weights that decrease toward the root, the query gives a node v and a weight h and seeks for its nearest ancestor with weight $< h$. In our case, the tree has $O(r)$ nodes and the weights are LCP values, in the range $[0\,..\,n-1]$.

Kopelowitz and Lewenstein [2007, Sec. 3.2] show how this query can be solved in $O(r)$ space and the time of a predecessor query plus $O(\log^* r)$. Those predecessor queries are done on universes of size n where there can be arbitrarily few elements. However, we can resort to binary search if there are $O(n/r)$ elements, within the allowed time $O(\log(n/r))$. Therefore, the predecessor queries have to be implemented only on sets of $\Omega(n/r)$ elements. By using the structure of Belazzougui and Navarro [2015, Thm. 14], the predecessor time is $O(\log\log_w r)$. Therefore, we obtain time $t_{\mathrm{SV}'} = O(\log(n/r) + \log\log r)$.

This time can be reduced to $t_{\mathrm{SV}} = O(\log(n/r))$ for the simpler primitives PSV/NSV as follows: When r is so large that $\log(n/r) < \log\log r$, which is covered by $r > n/\log n$, the allowed $\Theta(r\log(n/r)w)$ bits of space are actually $\Omega(n\log\log n)$. We are then entitled to use $O(n)$ bits of space, within which we can solve queries PSV and NSV in $O(1)$ time [Fischer et al. 2009, Thm. 3].

6.4. Supporting operation *Child*

To solve $\mathit{Child}(v,a)$ we binary search the $O(\sigma)$ positions where the minimum occurs in $LCP[v_l+1\,..\,v_r]$, and choose the one that descends by letter a. Each check for a takes $O(\log(n/r))$ time, as explained, so we aim at obtaining time $O(\log(n/r)\log\sigma)$.

To implement this operation efficiently, we store for each nonterminal X the number $n(X)$ of times $m(X)$ occurs inside the expansion of X. To do the binary search on $LCP[p\,..\,q]$ (with $p = v_l + 1$ and $q = v_r$), we first compute $\mathrm{RMQ}(p,q)$ as in the previous section, cutting the interval into areas (i), (ii), and (iii), and finding the absolute position and value of the (leftmost occurrence of the) minimum in each area. The global minimum $\mu = LCP[\mathrm{RMQ}(p,q)]$ may have occurrences inside each of the three areas, $X_x\langle p-p'+1, l(X_x)\rangle$, $DLCP'[x+1\,..\,y-1]$, and $X_y\langle 1, l(X_y)+q-q'\rangle$ (the values p', q', x, and y are those we computed to obtain $\mathrm{RMQ}(p,q)$). By computing the letter corresponding to the leftmost occurrence of μ inside $DLCP'[x+1\,..\,y-1]$, and $X_y\langle 1, l(X_y)+q-q'\rangle$, we determine in which of the three areas we must binary search for a.

Searching inside a nonterminal. To process $X_w\langle a,b\rangle$ (cases (i) and (iii)) we first determine how many occurrences of μ it contains. We initialize a counter $c = 0$ and scan again $Z_1, \ldots, Z_{k'}$, converting $\mu \leftarrow \mu - A[w-1] - d(Z_1) - \cdots - d(Z_{k'})$ into relative form. We then scan $Y_1, \ldots, Y_k$. For each Y_s, if $d(Y_1) + \cdots + d(Y_{s-1}) + m(Y_s) = \mu$, then the local minimum inside Y_s is indeed μ, so we add up all of its occurrences inside Y_s, $c \leftarrow c + n(Y_s)$. To process $Y^{t'}$ in constant time, we have already seen how to compute $m(Y^{t'})$. Further, remind that, if μ occurs in $Y^{t'}$, then it occurs only in the first copy of Y if $d(Y) > 0$, only in the last if $d(Y) < 0$, and in every copy if $d(Y) = 0$. Thus, $n(Y^{t'}) = n(Y)$ if $d(Y) \neq 0$ and $t' \cdot n(Y)$ if $d(Y) = 0$.

After we compute c in $O(\log(n/r))$ time, we binary search the c occurrences of μ in $X_w\langle a,b\rangle$. For each of the $O(\log c) = O(\log\sigma)$ steps of this binary search, we must find a specific occurrence of μ, and then compute the corresponding letter to compare with a and decide the direction of the search. As said, we can compute the corresponding letter in time $O(\log(n/r))$. We now show how a specific occurrence of μ is found within the same time complexity.

Finding a specific occurrence inside a nonterminal. Assume we want to find the c'th occurrence of μ in $X_w\langle a,b\rangle$. We initialize $p \leftarrow L[w-1] + l(Z_1) + \cdots + l(Z_{k'})$ and scan $Y_1, \ldots, Y_k$ again. For each Y_s, if $d(Y_1) + \cdots + d(Y_{s-1}) + m(Y_s) = \mu$, we subtract $c' \leftarrow c' - n(Y_s)$. When the result is less than 1, the occurrence sought is inside Y_s. We then add $l(Y_1) + \cdots + l(Y_{s-1})$ to p, subtract $d(Y_1) + \cdots + d(Y_{s-1})$ from μ, restore $c' \leftarrow c' + n(Y_s)$, and recursively search for μ inside Y_s.

Let $Y_s \to Y_1 Y_2$. If $m(Y_1) \neq \mu$, we continue on Y_2 with $p \leftarrow p + l(Y_1)$ and $\mu \leftarrow \mu - d(Y_1)$. If $m(Y_1) = \mu$ and $n(Y_1) \geq c'$, we continue on Y_1 because it contains the occurrence sought. Otherwise, Y_1 contains occurrences of μ but not the one sought. We then continue on Y_2 with $p \leftarrow p + l(Y_1)$, $\mu \leftarrow \mu - d(Y_1)$, and $c' \leftarrow c' - n(Y_1)$. On the other hand, if $Y_s \to Y^t$, we do as follows.

— If $d(Y) > 0$, then μ can only occur in the first copy of Y. Thus, if $m(Y) \neq \mu$, we just skip Y^t with $p \leftarrow p + t \cdot l(Y)$ and $\mu \leftarrow \mu - t \cdot d(Y)$. If $m(Y) = \mu$, instead, we see if $n(Y) \geq c'$. If so, then we enter into Y; otherwise we skip Y^t with $p \leftarrow p + t \cdot l(Y)$, $\mu \leftarrow \mu - t \cdot d(Y)$, and $c' \leftarrow c' - n(Y)$.
— The case where $d(Y) < 0$ is similar, except that when we enter into Y, it is the last one of Y^t, and thus we set $p \leftarrow p + (t-1) \cdot l(Y)$ and $\mu \leftarrow \mu - (t-1) \cdot d(Y)$.
— Finally, if $d(Y) = 0$, the minimum of Y appears many times. If $m(Y) \neq \mu$, we skip Y^t with $p \leftarrow p + t \cdot l(Y)$ and $\mu \leftarrow \mu - t \cdot d(Y)$. Otherwise, if $t \cdot n(Y) < c'$, we must also skip Y^t, updating p and μ, and also $c' \leftarrow c' - t \cdot n(Y)$. Otherwise, we must enter into the t'th occurrence of Y, where $t' = \lceil c'/n(Y) \rceil$, by continuing on Y with $p \leftarrow p + (t'-1) \cdot l(Y)$, $\mu \leftarrow \mu - (t'-1) \cdot d(Y)$ and $c' \leftarrow c' - (t'-1) \cdot n(Y)$.

Searching the central area. If the desired letter is inside area (ii), then any minimum in $DLCP'[x+1\,..\,y-1]$ is an occurrence of μ. The classical RMQ_M data structure gives us the leftmost one, thus we would be forced to sequentially search for the letter sought, in worst-case time $O(\sigma)$. To simulate a binary search, we would like that, if there are many occurrences of μ in $M[x+1\,..\,y-1]$, the structure RMQ_M returns us the median of the positions, not the leftmost. Such a data structure is not known, however, but fortunately there is one offering an approximation that still leads to logarithmic search times. Fischer and Heun [2010] presented a data structure that uses $O(r)$ bits on top of M (i.e., we need to represent M explicitly, using $O(r)$ space) and finds a position of μ in $M[x+1\,..\,y-1]$ whose rank among all the positions of μ in $M[x+1\,..\,y-1]$ is a fraction between 1/16 and 15/16 of the total.

For each position z where $M[z] = \mu$, returned by the data structure, we know that X_z contains some occurrence(s) of μ. We obtain the leftmost position $L[z-1] + p(X_z)$ of μ, find the associated letter, compare it with a, and determine if the binary search on $DLCP'[x+1\,..\,y-1]$ goes left or right. Since there are $O(\sigma)$ minima in $DLCP'[x+1\,..\,y-1]$, the search takes $O(\log(n/r)\log\sigma)$ time. Once we have finally determined that a, if it occurs, must occur inside some X_z, we process it as done on $X_w\langle a, b\rangle$ to determine the exact occurrence, if it exists.

7. EXPERIMENTAL RESULTS

We implemented our simplest scheme, that is, Theorem 3.6, and compared it with the state of the art.

7.1. Implementation

We implemented the simplified version described by Bannai et al. [2018] of the structure of Theorem 3.6 using the `sdsl` library [Gog et al. 2014].[17] For the RLFM-index, we used the implementation of L', E, D, and R (Lemmas 2.1 and 3.2) described by Prezza [2016, Thm. 28] (suffix array sampling excluded), taking $(1+\epsilon)r(\log(n/r)+2) + r\log\sigma$ bits of space (lower-order terms omitted for readability) for any constant $\epsilon > 0$ fixed at construction time and supporting $O(\log(n/r) + \log\sigma)$-time LF mapping. In our implementation, we chose $\epsilon = 0.5$. This structure employs Huffman-compressed wavelet trees (`sdsl`'s `wt_huff`) to represent the array L', as in our experiments they turned out to be comparable in size and faster than the structure of Golynski et al. [2006], which is implemented in `sdsl`'s `wt_gmr`.

Our `locate` machinery is implemented as follows. We store one gap-encoded bitvector `First`$[1\,..\,n]$ marking with a bit set the text positions 1 and those that are the first in their BWT run (note that `First`$[i]$ refers to *text* position i, not BWT position). `First` is implemented using `sdsl`'s `sd_vector`, takes $r(\log(n/r) + 2)$ bits of space (lower-order terms omitted), and answers queries in $O(\log(n/r))$ time. We also store a vector `FirstToRun`$[1\,..\,r]$ such that text position `First.select`$_1$`(i)` belongs to the `FirstToRun`$[i]$-th BWT run. `FirstToRun` is a packed integer vector stored in $r\log r$ bits. Finally, we explicitly store r suffix array samples in a vector `Samples`$[1\,..\,r]$: `Samples`$[p]$ is the text position corresponding to the last letter in the p-th BWT run. `Samples` is also a packed vector, using $r\log n$ bits.

Let $SA[sp\,..\,ep]$ be the range of our query pattern. The RLFM-index and vector `Samples` are sufficient to find the range $[sp\,..\,ep]$ and locate $SA[ep]$ using the simplified toe-hold lemma [Ban-

[17] `https://github.com/simongog/sdsl-lite`

nai et al. 2018], since Samples supplies the second component of the pairs in R, $SA[p]$. Moreover, with First and FirstToRun we obtain the functionality of $P^{\pm}$ (Lemma 3.5): it holds that $\phi(i) = \texttt{Samples}[\texttt{FirstToRun}[\texttt{First.rank}_1(i)] - 1] + \Delta$, where $\Delta = i - \texttt{First.predecessor}(i)$. Note that ϕ is evaluated in just $O(\log(n/r))$ time. Notably, this time drops to $O(1)$ in the average case, if the bits set in First are uniformly distributed. This is because sdsl's sd_vector breaks the bitvector into r equal-sized buckets and solves queries inside each bucket (which in the average case contains just $O(1)$ bits set). Occurrences $SA[ep-1], SA[ep-2], \ldots, SA[sp]$ are then retrieved as $\phi^k(SA[ep])$, for $k = 1, \ldots, ep-sp$.

Overall, our index takes at most $((1+\epsilon)\log(n/r) + 2\log n + \log\sigma + 4 + 2\epsilon)\,r$ bits of space for any constant $\epsilon > 0$ (lower-order terms omitted for readability) and, after counting, locates each pattern occurrence in $O(\log(n/r))$ time. The space of our index essentially coincides with the information-theoretic minimum needed for storing the run-length encoded BWT and $2r$ text positions in plain format (which is $r\log(n/r) + r\log\sigma + 2r\log n$ bits); therefore it is close to the optimum, since our locate strategy requires storing $2r$ text positions. From now on, we refer to our index as r-index; the code is publicly available[18].

7.2. Experimental Setup

We compared r-index with the state-of-the-art index for each compressibility measure: lzi[19] [Kreft and Navarro 2013; Claude et al. 2016] (z), slp[19] [Claude and Navarro 2010; Claude et al. 2016] (g), rlcsa[20] [Mäkinen et al. 2009; Mäkinen et al. 2010] (r), and cdawg[21] [Belazzougui et al. 2015a] (e). We also included hyb[22] [Ferrada et al. 2013; Ferrada et al. 2018], which combines a Lempel-Ziv index with an FM-index, with parameter $M = 8$, which is optimal for our experiment, and two implementations [23] of the FM-index from sdsl: fmi-rrr, which combines a Huffman-shaped topology with RRR compressed bitvectors [Raman et al. 2007], and fmi-suc, which combines a Huffman-shaped topology with succinct bitvectors. We tested rlcsa, fmi-rrr, and fmi-suc using different suffix array sample rates in order to highlight the space-time trade-off introduced by this component.

We measured memory usage and locate times per occurrence of all indexes on 1000 patterns of length 8 extracted from four repetitive datasets, which are also published with our implementation:

DNA. A synthetic dataset of 629,145 copies of a human DNA sequence of length 1000 where each position was mutated with probability 10^{-3} (typical rate in human DNA [Przeworski et al. 2000]);
boost. A dataset consisting of concatenated versions of the GitHub's boost library;
einstein. A dataset consisting of concatenated versions of Wikipedia's English Einstein page;
world_leaders. A collection of all pdf files of CIA World Leaders from 2003 to 2009 downloaded from the Pizza&Chili corpus.

Table V shows the main characteristics of the datasets: the length n, the alphabet size σ, the number of runs r in their BWT, the number z of Lempel-Ziv phrases[24], the size of g the grammar generated by Repair[25], and the sum e between the number of CDAWG nodes and edges (only for DNA, where cdawg can be built). Note the varying degrees of repetitiveness: boost is the most repetitive dataset, followed by DNA and einstein, which are similar, and then followed by the least repetitive one, world_leaders. It can be seen that $g \geq z$ by a factor of 1.3–2.8 and $r \geq g$ by a factor of 1.0–1.8. Therefore, we could expect in general that the indexes based on grammars or on Lempel-Ziv parsing are smaller than r-index, but as we see soon, the differences are not that large.

Memory usage (Resident Set Size, RSS) was measured using /usr/bin/time between index loading time and query time. This choice was motivated by the fact that, due to the datasets' high repetitiveness, the number occ of pattern occurrences was very large. This impacts sharply on the working space of indexes such as lzi and slp, which report the occurrences in a recursive fashion. When consider-

[18] https://github.com/nicolaprezza/r-index
[19] https://github.com/migumar2/uiHRDC
[20] https://github.com/adamnovak/rlcsa
[21] https://github.com/mathieuraffinot/locate-cdawg
[22] https://github.com/hferrada/HydridSelfIndex
[23] https://github.com/nicolaprezza/FMI
[24] Using code requested to the authors of an efficient Lempel-Ziv parser [Kärkkäinen et al. 2013].
[25] Using the "balanced" version offered at http://www.dcc.uchile.cl/gnavarro/repair.tgz

Table V. The main characteristics of our dataset. The numbers in parentheses are rough approximations to the bits/symbol achievable by the associated compressors by using one 4-byte integer per run, phrase, right-hand-side grammar symbol, or CDAWG edge/node.

Dataset	n	σ	r	z	g	e
DNA	629,140,006	10	1,287,508 (0.065)	551,237 (0.028)	727,671 (0.037)	248,489,728 (12.638)
boost	629,145,600	96	62,025 (0.003)	22,747 (0.001)	63,480 (0.003)	-
einstein	629,145,600	194	958,671 (0.049)	292,117 (0.015)	631,239 (0.032)	-
world_leaders	46,968,181	89	573,487 (0.391)	175,740 (0.120)	507,525 (0.346)	-

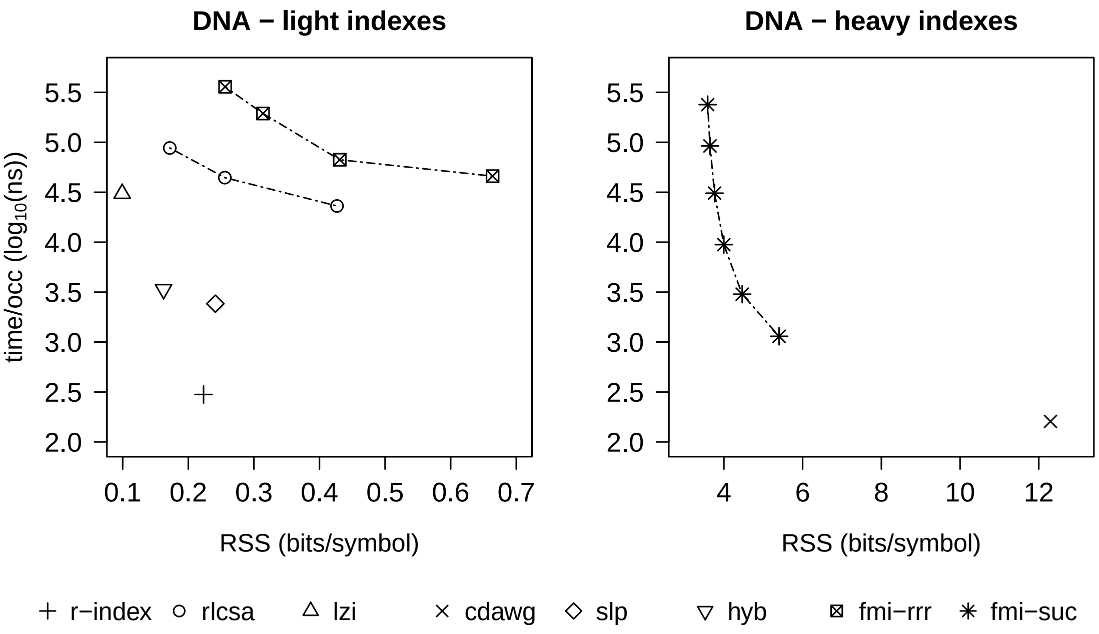

Fig. 6. Locate time per occurrence and working space (in bits per symbol) of the indexes. The y-scale measures nanoseconds per occurrence reported and is logarithmic. To improve readability, we separate light and heavy indexes in two plots, corresponding to two compression regimes.

ing this extra space, these indexes always use more space than r-index, but we prefer to emphasize the relation between the index sizes and their associated compressibility measure. The only existing implementation of cdawg works only on DNA files, so we tested it only on the DNA dataset.

7.3. Results

Figures 6 to 9 summarize the results of our experiments. Due to the very diverse compression regimes of the tested indexes, we separate them in two plots per dataset to improve readability.

On all the datasets, the time per occurrence of r-index is 100–300 nanoseconds per occurrence, outperforming all the indexes based on Lempel-Ziv or grammars by a factor of 10 to 100. These indexes are generally smaller, using 45%–95% (lzi), 80%–105% (slp), and 45%–100% (hyb) of the space of r-index, at the expense of being orders of magnitude slower, as said: 20–100 (lzi), 8–50 (slp), and 7–11 (hyb) times. Comparing with the bits per symbol of Table V, we note that the space of r-index is 2–4 words per run, whereas lzi and hyb use 3–6 words per Lempel-Ziv phrase and slp uses 4–6 words per symbol on the right-hand-side of a rule. The low space per run of r-index compared to the indexes based on z or g shrink the space gap one could expect from comparing the measures r, z, and g.

Further, r-index dominates all practical space-time tradeoffs of rlcsa, fmi-rrr, and fmi-suc. Using the same space, rlcsa is 20–500 times slower than rindex; letting it use 1.7–4.4 times the space of r-index, it is still 5–100 times slower. The lightest FM-index, fmi-rrr, is dominated in space by rlcsa

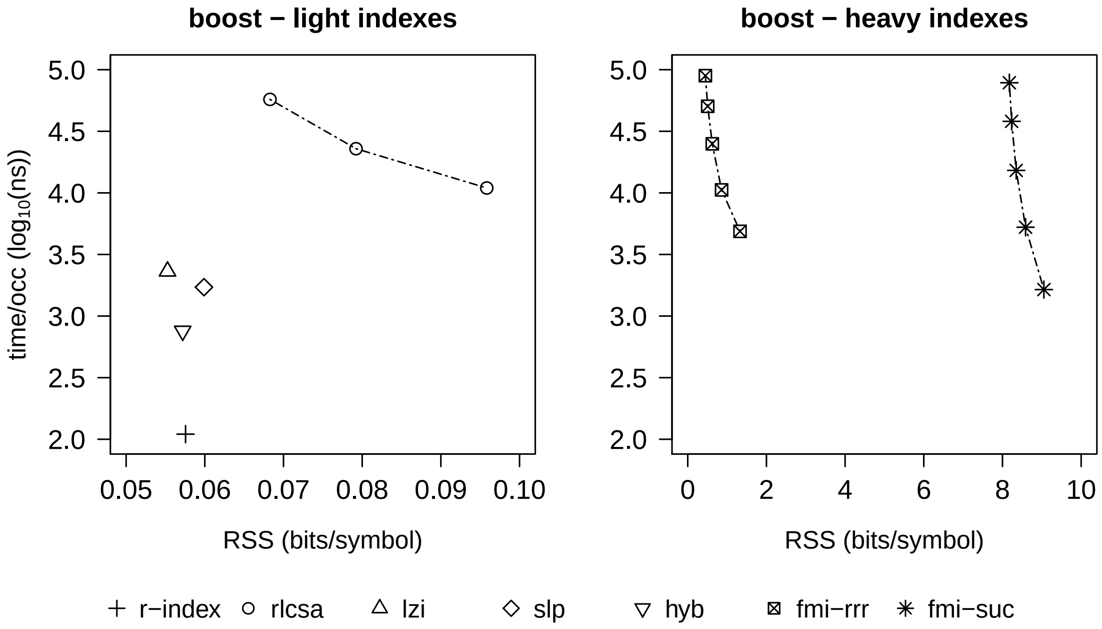

Fig. 7. Locate time per occurrence and working space (in bits per symbol) of the indexes. The y-scale measures nanoseconds per occurrence reported and is logarithmic. To improve readability, we separate light and heavy indexes in two plots, corresponding to two compression regimes. Cdawg works only on DNA, thus it is not displayed.

on all datasets except world_leaders, the least repetitive one. In all cases, rlcsa is also faster than fmi-rrr. This is because, unlike fmi-rrr, rlcsa implements heuristics to locate multiple occurrences at once. Interestingly, on most datasets fmi-rrr falls in the high-compression regime of dictionary-compressed indexes. This is because the BWT runs translate into 0/1 runs on the wavelet tree bitvectors, and those runs are compressed by a factor of $\Theta(\log\log n/\log n)$ by the RRR representation [Raman et al. 2007].[26] Instead, fmi-suc only offers global frequency compression, and thus it is up to an order of magnitude larger than fmi-rrr (while being faster by just a factor of 2 in most cases). In all cases, the comparison between the three FM-indexes and r-index shows that the regular sampling mechanism of the FM-index is completely outperformed on repetitive data.

Only cdawg is faster than r-index (almost twice as fast), but it is 60 times larger (indeed, way larger than the FM-indexes), which leaves it out of the range of “small” indexes.

7.4. Scalability

We now evaluate the space performance of the indexes on a real collection of Influenza nucleotide sequences from NCBI[27]. It is formed by 641,444 sequences, of total size 0.95 GB after removing the headers and newlines. We built the indexes on 100 prefixes of the dataset, whose sizes increased evenly from 1% to 100% of the sequences. As the prefixes grew, they became more repetitive; we measured how the bits per symbol used by the indexes decreased accordingly.

Figure 10 shows the rate between the measures r, z, g and the prefix length n. In this collection the repetitiveness is not as high as in the previous datasets (it is similar to world_leaders), but still r reaches 1% of n, and g and z reach about a half and a quarter of r, respectively. After a sharp initial reduction within the first 10% of the collection, the ratios continue decreasing slowly but steadily. These ratios are likely to stabilize at some function of the mutation rate of the sequences.

[26] One of the original RLFM-index variants, called RLBW, aimed to exploit this property, but it was outperformed by the variant implemented here as rlcsa. Still, it performed better than the original RLFM+ [Mäkinen et al. 2010].

[27] ftp://ftp.ncbi.nih.gov/genomes/INFLUENZA/influenza.fna.gz, the description is in the parent directory.

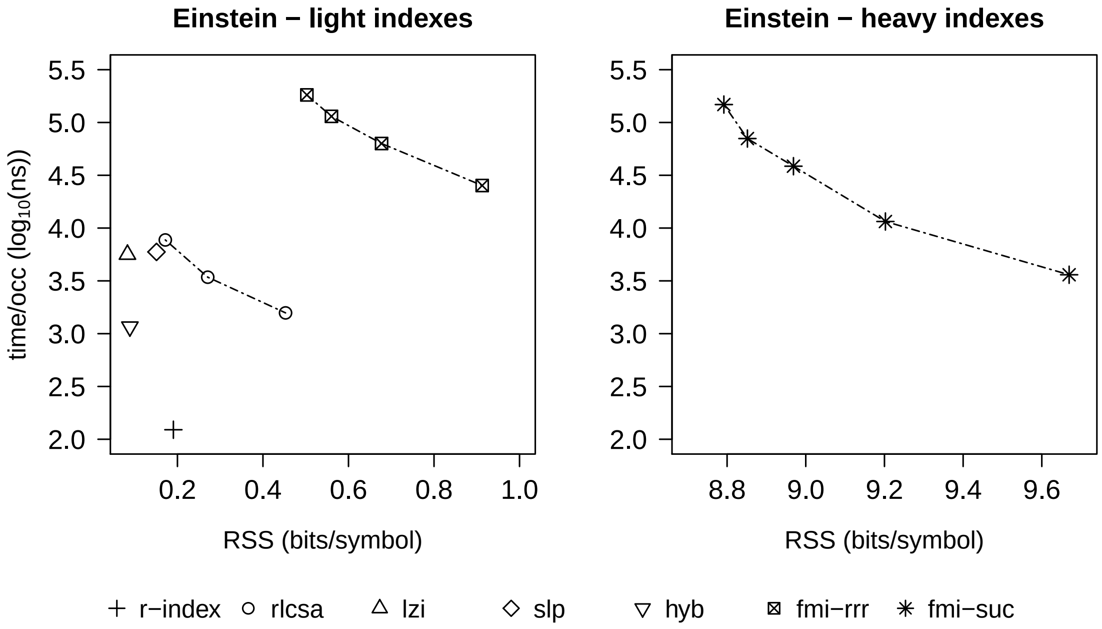

Fig. 8. Locate time per occurrence and working space (in bits per symbol) of the indexes. The y-scale measures nanoseconds per occurrence reported and is logarithmic. To improve readability, we separate light and heavy indexes in two plots, corresponding to two compression regimes. Cdawg works only on DNA, thus it is not displayed.

Figure 11 shows the evolution of the index sizes. As a repetition-insensitive variant, we also include a classical succinct FM-index (fm-index), with a typical sampling rate of $\lceil \lg n \rceil$ positions for locating, plain bitvectors for the wavelet trees and for marking the sampled SA positions, and a rank implementation using 1.25 bits per input bit. As we add more and more similar sequences, all the indexes (except the FM-index) decrease in relative size (bps), as expected. On the complete collection, fm-index still uses 4.75 bits per symbol (bps), whereas r-index has decreased to 0.88 bps (about 2.4 words per run), hyb to 0.52 bps (about 5.5 words per phrase, 60% of r-index), slp to 0.49 bps (about 1.9 words per symbol, 56% of r-index), and lzi to 0.22 bps (about 2.3 words per phrase, 25% of r-index). We remind that, in exchange, the r-index is 10–100 times faster than those indexes, and that it uses 18% of the space of the classic fm-index (a factor that decreases as the collection grows).

This not-so repetitive collection shows that r (and thus r-index) is more sensitive than g and z to the decrease in repetitiveness. In particular, g and z are always $O(n/\log_\sigma n)$, and thus the related indexes always use $O(n \log \sigma)$ bits. Instead, r can be as large as n [Prezza 2016], so in the worst case r-index can use $\Theta(n \log n)$ bits. Note, in particular, that the other indexes are below the 2 bps of the raw data after processing just 3% of the collection; r-index breaks this barrier only after 8%.

7.5. Comparison with state-of-the-art bioinformatic software

There are currently two bioinformatics groups collaborating on the further development of the r-index: those of Christina Boucher at the University of Florida and of Ben Langmead at Johns Hopkins University. We note that Ben Langmead was the first author of Bowtie [Langmead et al. 2009; Langmead and Salzberg 2012], one of the most popular short-read aligners based on the FM-index. Their results show that the r-index can index much larger DNA datasets than those we considered here, such as a 120-GB file containing 2000 human chromosome 19 haplotypes, and demonstrate the r-index's potential as a tool in bioinformatics.

For example, Kuhnle et al. [2019] study practical methods to build the r-index. They show that Bowtie can index up to 250 chromosome 19 haplotypes (around 14.5 billion bases, each chromosome being around 58 million bases long), whereas the r-index can index over 1000 (59 billion bases). The

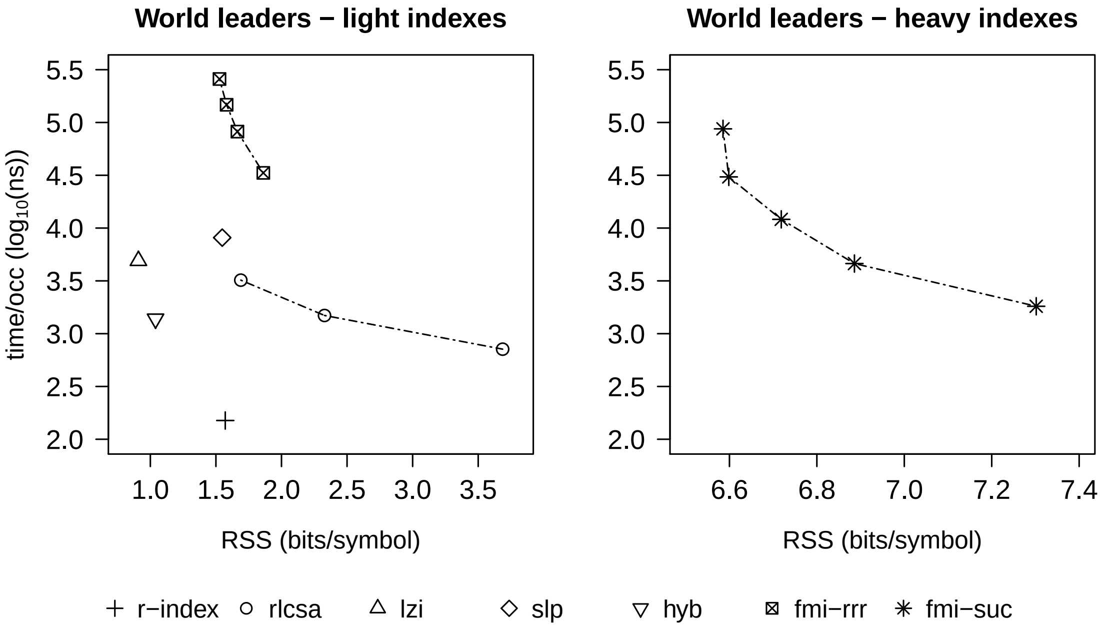

Fig. 9. Locate time per occurrence and working space (in bits per symbol) of the indexes. The y-scale measures nanoseconds per occurrence reported and is logarithmic. To improve readability, we separate light and heavy indexes in two plots, corresponding to two compression regimes. Cdawg works only on DNA, thus it is not displayed.

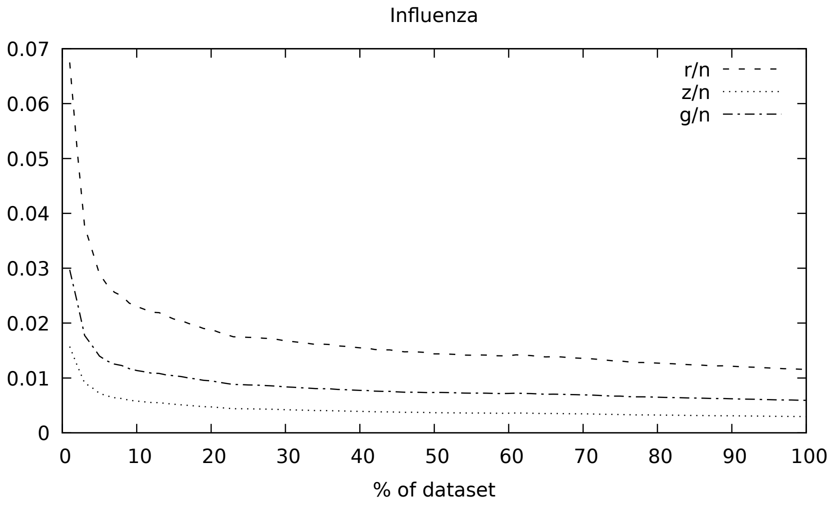

Fig. 10. Rates between the measures z, r, g (g stands for the size of the RePair grammar) and the text length n, for increasing prefixes of a repetitive collection of genomic data.

r-index takes from 250 MB to index 1 chromosome (35 bps, no repetitiveness) to 550 MB to index 1000 chromosomes (0.08 bps). That is, the r-index roughly doubles its space as the collection becomes 1000 times larger. Instead, Bowtie grows linearly: it uses 25 MB to index 1 chromosome (3.46 bps) and 7 GB to index 250 (4 bps). Both indexes reach the same size on 10 chromosomes, and from then on the

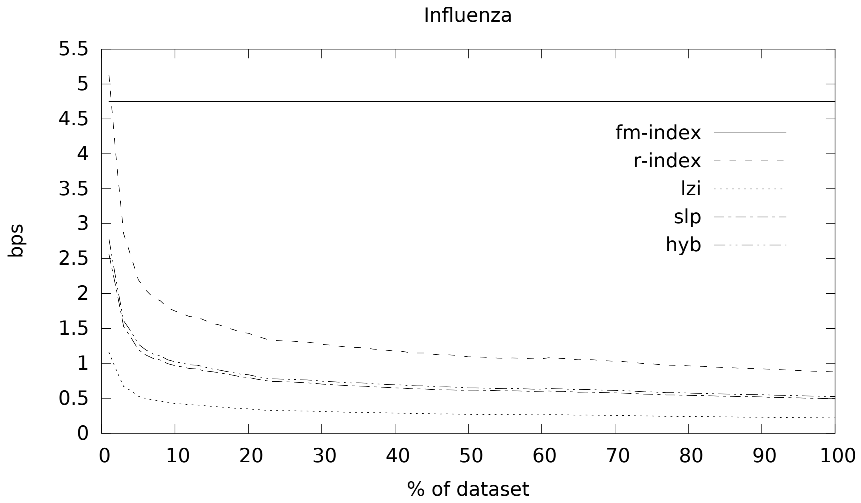

Fig. 11. Index sizes (in bits per symbol, bps) for the same prefixes of Figure 10.

`r-index` takes over. The `r-index` is also built using 1–2 orders of magnitude less space and time than `Bowtie`.

Note that the bits per symbol reached by the `r-index` in this collection, 0.08 bps, is much lower than the 0.88 bps achieved on the Influenza dataset of our experiments above. This corresponds to the much lower mutation rates of human genomes compared to bacteria.

The `r-index` is also faster at locating when indexing more than 100 chromosomes. On 250 chromosomes, for example, all the occurrences of a random substring of length 100 appearing in all the chromosomes are located in 900 microseconds by `Bowtie` and in 500 microseconds by the `r-index`.

8. CONCLUSIONS

We have closed the long-standing problem of efficiently locating the occurrences of a pattern in a text using an index whose space is bounded by the number of equal-letter runs in the Burrows-Wheeler transform (BWT) of the text. The occ occurrences of a pattern $P[1..m]$ in a text $T[1..n]$ over alphabet $[1..\sigma]$ whose BWT has r runs can be counted in time $O(m \log\log_w(\sigma + n/r))$ and then located in $O(occ \log\log_w(n/r))$ time, on a w-bit RAM machine, using an $O(r)$-space index. Using space $O(r \log\log_w(\sigma + n/r))$, the counting and locating times are reduced to $O(m)$ and $O(occ)$, respectively, which is optimal in the general setting. Further, using $O(rw \log_\sigma \log_w n)$ space we can also obtain optimal time in the packed setting, replacing $O(m)$ by $O(\lceil m \log(\sigma)/w \rceil)$ in the counting time. Our findings also include $O(r \log(n/r))$-space structures to access consecutive entries of the text, suffix array, inverse suffix array, and longest common prefix array, in optimal time plus a per-query penalty of $O(\log(n/r))$. We upgraded those structures to a full-fledged compressed suffix tree working in $O(r \log(n/r))$ space and carrying out most navigation operations in time $O(\log(n/r))$. All the structures are built in times ranging from $O(n)$ worst-case to $O(nw^{1+\epsilon})$ expected time and $O(n)$ space, and many can be built in the same asymptotic space of the final structure with a single pass over the text.

The number of runs in the BWT is an important measure of the compressibility of highly repetitive text collections, which can be compressed by orders of magnitude by exploiting the repetitiveness. While the first index of this type [Mäkinen et al. 2009; Mäkinen et al. 2010] managed to exploit the BWT runs, it was not able to locate occurrences efficiently. This gave rise to many other indexes based on other measures, like the size z of a Lempel-Ziv parse [Lempel and Ziv 1976], the size g of a context-free grammar [Kieffer and Yang 2000], the size e of the smallest compact automaton recognizing the text substrings [Blumer et al. 1987], etc. While the complexities are not always comparable [Gagie et al. 2018a], the experimental results show that our proof-of-concept implementation outperforms all the space-efficient alternatives by one or two orders of magnitude in locating time.

This work triggered several other lines of research. From the idea of cutting the text into phrases defined by the BWT run ends, we showed that a run-length context-free grammar (RLCFG) of size $O(r\log(n/r))$ can be built on the text by using locally consistent parsing [Jeż 2015]. This was generalized to a RLCFG built on top of any bidirectional macro scheme (BMS) [Storer and Szymanski 1982], which allowed us to prove bounds on the Lempel-Ziv approximation to the optimal BMS, as well as several other related bounds between compressibility measures [Gagie et al. 2018a; Navarro and Prezza 2018]. Also, the idea that at least one occurrence of any text substring must cross a phrase boundary led Kempa and Prezza [2018] to the concept of *string attractor*, a set of γ text positions with such a property. They prove that string attractors subsume the other measures of repetitiveness (i.e., $\gamma \le \min(r,z,g,e)$), and design universal data structures of size $O(\gamma\log(n/\gamma))$ for accessing the compressed text, analogous to ours. Navarro and Prezza [2019] then extend these ideas to the first self-index on attractors, of size $O(\gamma\log(n/\gamma))$, locating in time $O(m\log n + occ\log^\epsilon n)$. Very recently, Christiansen et al. [2019] obtained, within $O(\gamma\log(n/\gamma))$ space, counting and locating time $O(m+\log^{2+\epsilon} n)$ and $O(m+(occ+1)\log^\epsilon n)$, respectively. Further, they obtained optimal times $O(m)$ and $O(m+occ)$, by raising the space to $O(\gamma\log(n/\gamma)\log n)$ and $O(\gamma\log(n/\gamma)\log^\epsilon n)$, respectively. We obtain such optimal times within space $O(r\log_\sigma\log_w n)$ or $O(r\log(n/r))$, which are incomparable with those (we can only ensure $O(\gamma\log(n/\gamma)) \subseteq O(r\log(n/r))$). We note that the optimal time in the packed setting we achieve in $O(r\log_\sigma\log_w n)$ space had been obtained only in $\Theta(n)$ space.

On the other hand, some questions remain open, in particular regarding the operations that can be supported within $O(r)$ space. We have shown that this space is not only sufficient to represent the text, but also to efficiently count and locate pattern occurrences. We required, however, $O(r\log(n/r))$ space to provide random access to the text. This raises the question of whether efficient random access is possible within $O(r)$ space. For example, recalling Table I, random access in sublinear time is possible within $O(g)$ or $O(z\log(n/z))$ space; we can also access within space $O(\gamma\log(n/\gamma))$, as said above [Kempa and Prezza 2018]. All these spaces are incomparable with r. A more specific question, but still intriguing, is whether we can provide random access to the suffix array of the text in $O(r)$ space: note that we can return the cells *that result from a pattern search* within this space, but accessing an arbitrary cell requires $O(r\log(n/r))$ space, and this translates into the size required by a suffix tree. On the other hand, the recent result $r = O(\gamma\log^2 n)$ [Kempa and Kociumaka 2019] implies that we offer suffix tree functionality in space $O(\gamma\log^3 n) \subseteq O(z\log^3 n)$.

Finally, we have worked with the groups of Christina Boucher and Ben Langmead in order to use the `r-index` on real bioinformatic software, which involves several practical challenges. The cooperation has already produced results: a paper [Boucher et al. 2018] at the 2018 Workshop on Algorithms in Bioinformatics (WABI) that was invited to, and has now been published [Boucher et al. 2019] in, a special issue of *Algorithms in Molecular Biology*; and a paper [Kuhnle et al. 2019] at the 2019 Conference on Research in Computational Molecular Biology (RECOMB) that was invited as two articles — a standard academic article and a brief manual for the software, both of which are now accepted — to a special issue of the *Journal of Computational Biology*. Some of these results are already mentioned in Section 7.5. We keep collaborating to add more functionality to their implementation of the `r-index`, such as combining it with variation graphs and other graph-based pan-genomic indexes, extending it to find maximal exact matches, or adding efficient techniques to insert new sequences in an existing index; there is already some progress in the last two directions [Bannai et al. 2018]. Another important practical aspect is, as explained in Section 7, making the index less sensitive to lower repetitiveness scenarios, as it could be the case of indexing short sequences (e.g., sets of reads) or metagenomic collections. We are working on a hybrid with the classic FM-index to handle in different ways the areas with higher and lower repetitiveness. Finally, extending our index to enable full suffix tree functionality will require, despite our theoretical achievements in Section 6, a significant amount of algorithm engineering to obtain good practical space figures.

Acknowledgements

Part of this work was done during Dagstuhl Seminar 19241, "25 Years of the Burrows-Wheeler Transform". We thank Ben Langmead for his pointers to the prevalence of the FM-index in bioinformatics software, and for sharing the results of his implementation of our index on large datasets. We also thank Meg Gagie for checking our grammar.

REFERENCES

ABELIUK, A., CÁNOVAS, R., AND NAVARRO, G. 2013. Practical compressed suffix trees. *Algorithms 6,* 2, 319–351.

BANNAI, H., GAGIE, T., AND I, T. 2018. Online LZ77 parsing and matching statistics with RLBWTs. In *Proc. 29th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 7:1–7:12.

BELAZZOUGUI, D., BOLDI, P., PAGH, R., AND VIGNA, S. 2009a. Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 785–794.

BELAZZOUGUI, D., BOLDI, P., PAGH, R., AND VIGNA, S. 2011. Theory and practice of monotone minimal perfect hashing. *ACM Journal of Experimental Algorithmics 16,* 3, article 2.

BELAZZOUGUI, D., BOLDI, P., PAGH, R., AND VIGNA, S. 2018. Fast prefix search in little space, with applications. *CoRR 1804.04720*.

BELAZZOUGUI, D., BOTELHO, F. C., AND DIETZFELBINGER, M. 2009b. Hash, displace, and compress. In *Proc. 17th Annual European Symposium (ESA)*. 682–693.

BELAZZOUGUI, D. AND CUNIAL, F. 2017a. Fast label extraction in the CDAWG. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*. 161–175.

BELAZZOUGUI, D. AND CUNIAL, F. 2017b. Representing the suffix tree with the CDAWG. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 7:1–7:13.

BELAZZOUGUI, D., CUNIAL, F., GAGIE, T., PREZZA, N., AND RAFFINOT, M. 2015a. Composite repetition-aware data structures. In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 26–39.

BELAZZOUGUI, D., GAGIE, T., GAWRYCHOWSKI, P., KÄRKKÄINEN, J., ORDÓÑEZ, A., PUGLISI, S. J., AND TABEI, Y. 2015b. Queries on LZ-bounded encodings. In *Proc. 25th Data Compression Conference (DCC)*. 83–92.

BELAZZOUGUI, D., GAGIE, T., GOG, S., MANZINI, G., AND SIRÉN, J. 2014. Relative FM-indexes. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*. 52–64.

BELAZZOUGUI, D. AND NAVARRO, G. 2014. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms 10,* 4, article 23.

BELAZZOUGUI, D. AND NAVARRO, G. 2015. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms 11,* 4, article 31.

BELAZZOUGUI, D., PUGLISI, S. J., AND TABEI, Y. 2015c. Access, rank, select in grammar-compressed strings. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*. 142–154.

BILLE, P., ETTIENNE, M. B., GØRTZ, I. L., AND VILDHØJ, H. W. 2018. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theoretical Computer Science 713*, 66–77.

BILLE, P., LANDAU, G. M., RAMAN, R., SADAKANE, K., RAO, S. S., AND WEIMANN, O. 2015. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing 44,* 3, 513–539.

BLUMER, A., BLUMER, J., HAUSSLER, D., MCCONNELL, R. M., AND EHRENFEUCHT, A. 1987. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM 34,* 3, 578–595.

BOUCHER, C., GAGIE, T., KUHNLE, A., LANGMEAD, B., MANZINI, G., AND MUN, T. 2019. Prefix-free parsing for building big bwts. *Algorithms for Molecular Biology 14,* 1, 13:1–13:15.

BOUCHER, C., GAGIE, T., KUHNLE, A., AND MANZINI, G. 2018. Prefix-free parsing for building big BWTs. In *Proc. 18th International Workshop on Algorithms in Bioinformatics (WABI)*. 2:1–2:16.

BURROWS, M. AND WHEELER, D. 1994. A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation.

CÁCERES, M. AND NAVARRO, G. 2019. Faster repetition-aware compressed suffix trees based on block trees. In *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*. To appear.

CHARIKAR, M., LEHMAN, E., LIU, D., PANIGRAHY, R., PRABHAKARAN, M., SAHAI, A., AND SHELAT, A. 2005. The smallest grammar problem. *IEEE Transactions on Information Theory 51,* 7, 2554–2576.

CHEN, S., VERBIN, E., AND YU, W. 2012. Data structure lower bounds on random access to grammar-compressed strings. *CoRR 1203.1080*.

CHRISTIANSEN, A. R. AND ETTIENNE, M. B. 2018. Compressed indexing with signature grammars. In *Proc. 13th Latin American Symposium on Theoretical Informatics (LATIN)*. 331–345.

CHRISTIANSEN, A. R., ETTIENNE, M. B., KOCIUMAKA, T., NAVARRO, G., AND PREZZA, N. 2019. Optimal-time dictionary-compressed indexes. *CoRR 1811.12779v3*.

CLAUDE, F., FARIÑA, A., MARTÍNEZ-PRIETO, M., AND NAVARRO, G. 2016. Universal indexes for highly repetitive document collections. *Information Systems 61*, 1–23.

CLAUDE, F. AND NAVARRO, G. 2010. Self-indexed grammar-based compression. *Fundamenta Informaticae 111,* 3, 313–337.

CLAUDE, F. AND NAVARRO, G. 2012. Improved grammar-based compressed indexes. In *Proc. 19th International Symposium on String Processing and Information Retrieval (SPIRE)*. 180–192.

DO, H. H., JANSSON, J., SADAKANE, K., AND SUNG, W.-K. 2014. Fast relative Lempel-Ziv self-index for similar sequences. *Theoretical Computer Science 532*, 14–30.

FARACH-COLTON, M., FERRAGINA, P., AND MUTHUKRISHNAN, S. 2000. On the sorting-complexity of suffix tree construction. *Journal of the ACM 47,* 6, 987–1011.

FARRUGGIA, A., GAGIE, T., NAVARRO, G., PUGLISI, S. J., AND SIRÉN, J. 2018. Relative suffix trees. *The Computer Journal 61,* 5, 773–788.

FERRADA, H., GAGIE, T., HIRVOLA, T., AND PUGLISI, S. J. 2013. Hybrid indexes for repetitive datasets. *CoRR 1306.4037*.

FERRADA, H., KEMPA, D., AND PUGLISI, S. J. 2018. Hybrid indexing revisited. In *Proc. 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 1–8.

FERRAGINA, P. AND MANZINI, G. 2005. Indexing compressed texts. *Journal of the ACM 52,* 4, 552–581.

FERRAGINA, P., MANZINI, G., MÄKINEN, V., AND NAVARRO, G. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms 3,* 2, article 20.

FISCHER, J. 2010. Wee LCP. *Information Processing Letters 110,* 8-9, 317–320.

FISCHER, J. AND HEUN, V. 2010. Finding range minima in the middle: Approximations and applications. *Mathematics in Computer Science 3,* 1, 17–30.

FISCHER, J. AND HEUN, V. 2011. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing 40,* 2, 465–492.

FISCHER, J., MÄKINEN, V., AND NAVARRO, G. 2009. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science 410,* 51, 5354–5364.

FREDMAN, M. L., KOMLÓS, J., AND SZEMERÉDI, E. 1984. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM 31,* 3, 538–544.

FREDMAN, M. L. AND WILLARD, D. E. 1993. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences 47,* 3, 424–436.

FRITZ, M. H.-Y., LEINONEN, R., COCHRANE, G., AND BIRNEY, E. 2011. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, 734–740.

GAGIE, T., GAWRYCHOWSKI, P., KÄRKKÄINEN, J., NEKRICH, Y., AND PUGLISI, S. J. 2012. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*. 240–251.

GAGIE, T., GAWRYCHOWSKI, P., KÄRKKÄINEN, J., NEKRICH, Y., AND PUGLISI, S. J. 2014. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th Latin American Symposium on Theoretical Informatics (LATIN)*. 731–742.

GAGIE, T., GAWRYCHOWSKI, P., AND PUGLISI, S. J. 2015. Approximate pattern matching in LZ77-compressed texts. *Journal of Discrete Algorithms 32*, 64–68.

GAGIE, T., NAVARRO, G., AND PREZZA, N. 2017. Optimal-time text indexing in BWT-runs bounded space. *CoRR 1705.10382v4*.

GAGIE, T., NAVARRO, G., AND PREZZA, N. 2018a. On the approximation ratio of Lempel-Ziv parsing. In *Proc. 13th Latin American Symposium on Theoretical Informatics (LATIN)*. 490–503.

GAGIE, T., NAVARRO, G., AND PREZZA, N. 2018b. Optimal-time text indexing in BWT-runs bounded space. In *Proc. 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1459–1477.

GOG, S., BELLER, T., MOFFAT, A., AND PETRI, M. 2014. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*. 326–337.

GOG, S. AND OHLEBUSCH, E. 2013. Compressed suffix trees: Efficient computation and storage of LCP-values. *ACM Journal of Experimental Algorithmics 18*, article 2.1.

GOLYNSKI, A., MUNRO, J. I., AND RAO, S. S. 2006. Rank/select operations on large alphabets: A tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 368–373.

GONZÁLEZ, R. AND NAVARRO, G. 2007. Compressed text indexes with fast locate. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 216–227.

GONZÁLEZ, R., NAVARRO, G., AND FERRADA, H. 2014. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics 19,* 1, article 1.

GROSSI, R. AND VITTER, J. S. 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

JANSON, S. 2017. Tail bounds for sums of geometric and exponential variables. *CoRR 1709.08157v1*.

JEŻ, A. 2015. Approximation of grammar-based compression via recompression. *Theoretical Computer Science 592*, 115–134.

JEŻ, A. 2016. A really simple approximation of smallest grammar. *Theoretical Computer Science 616*, 141–150.

KÄRKKÄINEN, J., KEMPA, D., AND PUGLISI, S. J. 2013. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 189–200.

KÄRKKÄINEN, J., MANZINI, G., AND PUGLISI, S. J. 2009. Permuted Longest-Common-Prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 181–192.

KÄRKKÄINEN, J., SANDERS, P., AND BURKHARDT, S. 2006. Linear work suffix array construction. *Journal of the ACM 53,* 6, 918–936.

KASAI, T., LEE, G., ARIMURA, H., ARIKAWA, S., AND PARK, K. 2001. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 181–192.

KEEL, B. N. AND SNELLING, W. M. 2018. Comparison of Burrows-Wheeler transform-based mapping algorithms used in high-throughput whole-genome sequencing: Application to Illumina data for livestock genomes. *Frontiers in Genetics 9*, article 35.

KEMPA, D. 2019. Optimal construction of compressed indexes for highly repetitive texts. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1344–1357.

KEMPA, D. AND KOCIUMAKA, T. 2019. Resolution of the Burrows-Wheeler Transform conjecture. *CoRR 1910.10631*.

KEMPA, D. AND PREZZA, N. 2018. At the roots of dictionary compression: String attractors. In *Proc. 50th Annual ACM Symposium on the Theory of Computing (STOC)*. 827–840.

KIEFFER, J. C. AND YANG, E.-H. 2000. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory 46,* 3, 737–754.

KIM, D. K., SIM, J. S., PARK, H., AND PARK, K. 2005. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms 3,* 2-4, 126–142.

KO, P. AND ALURU, S. 2005. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms 3,* 2-4, 143–156.

KODAMA, Y., SHUMWAY, M., AND LEINONEN, R. 2012. The sequence read archive: Explosive growth of sequencing data. *Nucleic Acids Research 40,* D1, D54–D56.

KOPELOWITZ, T. AND LEWENSTEIN, M. 2007. Dynamic weighted ancestors. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 565–574.

KREFT, S. AND NAVARRO, G. 2013. On compressing and indexing repetitive sequences. *Theoretical Computer Science 483*, 115–133.

KUHNLE, A., MUN, T., BOUCHER, C., GAGIE, T., LANGMEAD, B., AND MANZINI, G. 2019. Efficient construction of a complete index for pan-genomics read alignment. In *Proc. 23rd International Conference on Research in Computational Molecular Biology (RECOMB)*. 158–173.

KURUPPU, S., PUGLISI, S. J., AND ZOBEL, J. 2010. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*. 201–206.

LANGMEAD, B. AND SALZBERG, S. L. 2012. Fast gapped-read alignment with bowtie 2. *Nature methods 9,* 4, 357–359.

LANGMEAD, B., TRAPNELL, C., POP, M., AND SALZBERG, S. L. 2009. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology 10,* 3, R25.

LARSSON, J. AND MOFFAT, A. 2000. Off-line dictionary-based compression. *Proceedings of the IEEE 88,* 11, 1722–1732.

LEMPEL, A. AND ZIV, J. 1976. On the complexity of finite sequences. *IEEE Transactions on Information Theory 22,* 1, 75–81.

MÄKINEN, V., BELAZZOUGUI, D., CUNIAL, F., AND TOMESCU, A. I. 2015. *Genome-Scale Algorithm Design*. Cambridge University Press.

MÄKINEN, V. AND NAVARRO, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12,* 1, 40–66.

MÄKINEN, V., NAVARRO, G., SIRÉN, J., AND VÄLIMÄKI, N. 2009. Storage and retrieval of individual genomes. In *Proc. 13th Annual International Conference on Computational Molecular Biology (RECOMB)*. 121–137.

MÄKINEN, V., NAVARRO, G., SIRÉN, J., AND VÄLIMÄKI, N. 2010. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology 17,* 3, 281–308.

MANBER, U. AND MYERS, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing 22,* 5, 935–948.

MANZINI, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

MCCREIGHT, E. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM 23,* 2, 262–272.

MUNRO, J. I., NAVARRO, G., AND NEKRICH, Y. 2017. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 408–424.

NA, J. C., PARK, H., CROCHEMORE, M., HOLUB, J., ILIOPOULOS, C. S., MOUCHARD, L., AND PARK, K. 2013a. Suffix tree of alignment: An efficient index for similar data. In *Proc. 24th International Workshop on Combinatorial Algorithms (IWOCA)*. 337–348.

NA, J. C., PARK, H., LEE, S., HONG, M., LECROQ, T., MOUCHARD, L., AND PARK, K. 2013b. Suffix array of alignment: A practical index for similar data. In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*. 243–254.

NAVARRO, G. 2016. *Compact Data Structures – A practical approach*. Cambridge University Press.

NAVARRO, G. 2017. A self-index on block trees. In *Proc. 24th International Symposium on String Processing and Information Retrieval (SPIRE)*. 278–289.

NAVARRO, G. 2019. Document listing on repetitive collections with guaranteed performance. *Theoretical Computer Science 777*, 58–72.

NAVARRO, G. AND MÄKINEN, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, article 2.

NAVARRO, G. AND NEKRICH, Y. 2017. Time-optimal top-k document retrieval. *SIAM Journal on Computing 46,* 1, 89–113.

NAVARRO, G. AND ORDÓÑEZ, A. 2016. Faster compressed suffix trees for repetitive text collections. *Journal of Experimental Algorithmics 21,* 1, article 1.8.

NAVARRO, G. AND PREZZA, N. 2018. On the approximation ratio of greedy parsings. *CoRR 1803.09517*.

NAVARRO, G. AND PREZZA, N. 2019. Universal compressed text indexing. *Theoretical Computer Science 762*, 41–50.

NAVARRO, G. AND SADAKANE, K. 2014. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms 10,* 3, article 16.

NISHIMOTO, T., I, T., INENAGA, S., BANNAI, H., AND TAKEDA, M. 2015. Dynamic index, LZ factorization, and LCE queries in compressed space. *CoRR 1504.06954*.

NISHIMOTO, T., I, T., INENAGA, S., BANNAI, H., AND TAKEDA, M. 2016. Fully dynamic data structure for LCE queries in compressed space. In *Proc. 41st International Symposium on Mathematical Foundations of Computer Science (MFCS)*. 72:1–72:15.

OHLEBUSCH, E. 2013. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag.

OHNO, T., SAKAI, K., TAKABATAKE, Y., I, T., AND SAKAMOTO, H. 2018. A faster implementation of online RLBWT and its application to LZ77 parsing. *Journal of Discrete Algorithms 52-53*, 18–28.

POLICRITI, A. AND PREZZA, N. 2018. LZ77 computation based on the run-length encoded BWT. *Algorithmica 80,* 7, 1986–2011.

PREZZA, N. 2016. Compressed computation for text indexing. Ph.D. thesis, University of Udine.

PRZEWORSKI, M., HUDSON, R. R., AND RIENZO, A. D. 2000. Adjusting the focus on human variation. *Trends in Genetics 16,* 7, 296–302.

RAMAN, R., RAMAN, V., AND SATTI, S. R. 2007. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms 3,* 4, article 43.

RUSSO, L. M. S., NAVARRO, G., AND OLIVEIRA, A. 2011. Fully-compressed suffix trees. *ACM Transactions on Algorithms 7,* 4, article 53.

RUŽIĆ, M. 2008. Constructing efficient dictionaries in close to sorting time. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP)*. 84–95.

RYTTER, W. 2003. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science 302,* 1-3, 211–222.

SADAKANE, K. 2007. Compressed suffix trees with full functionality. *Theory of Computing Systems 41,* 4, 589–607.

SCHATZ, M. C. AND LANGMEAD, B. 2013. The DNA data deluge: Fast, efficient genome sequencing machines are spewing out more data than geneticists can analyze. *IEEE Spectrum 50,* 7, 26–33.

SIRÉN, J., VÄLIMÄKI, N., MÄKINEN, V., AND NAVARRO, G. 2008. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*. 164–175.

STHEPHENS, Z. D., LEE, S. Y., FAGHRI, F., CAMPBELL, R. H., CHENXIANG, Z., EFRON, M. J., IYER, R., SINHA, S., AND ROBINSON, G. E. 2015. Big data: Astronomical or genomical? *PLoS Biology 17,* 7, e1002195.

STORER, J. A. AND SZYMANSKI, T. G. 1982. Data compression via textual substitution. *Journal of the ACM 29,* 4, 928–951.

TAKAGI, T., GOTO, K., FUJISHIGE, Y., INENAGA, S., AND ARIMURA, H. 2017. Linear-size CDAWG: new repetition-aware indexing and grammar compression. In *Proc. 24th International Symposium of String Processing and Information Retrieval (SPIRE)*. 304–316.

UKKONEN, E. 1995. On-line construction of suffix trees. *Algorithmica 14,* 3, 249–260.

VERBIN, E. AND YU, W. 2013. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*. 247–258.

WEINER, P. 1973. Linear Pattern Matching Algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory (FOCS)*. 1–11.

WILLARD, D. E. 2000. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing 29,* 3, 1030–1049.

A. CONSTRUCTION

In this appendix we analyze the working space and time required to build all our data structures. Table VI summarizes the results. The working space does not count the space needed to read the text in online form, right-to-left. Times are worst-case unless otherwise stated. Expected cases hold with high probability (w.h.p.), which means over $1 - 1/n^c$ for any fixed constant c.

A.1. Dictionaries and predecessor structures

A dictionary mapping t keys from a universe of size u to an interval $[1\,..\,O(t)]$ can be implemented as a perfect hash function using $O(t)$ space and searching in constant worst-case time. Such a function can be built in $O(t)$ space and expected time [Fredman et al. 1984]. A construction that takes $O(t)$ time w.h.p. [Willard 2000] starts with a distributor hash function that maps the keys to an array of buckets $B[1\,..\,t]$. Since the largest bucket contains $O(\log t/\log\log t)$ keys w.h.p., we can build a fusion

Table VI. Construction time and space for our different data structures, for any constant $\epsilon > 0$. All the expected times ("exp.") hold w.h.p. as well. Variable B is the block size in the external memory model, where $Sort(n)$ denotes the I/O complexity of sorting n integers.

Structure	Construction time	Construction space
Basic counting and locating (Lem. 2.1)	$O(n \log r)$	$O(r)$
or	$O(n)$	$O(n)$
Fast locating (Thm. 3.6 + Lem. 3.7, $s = \log\log_w(n/r)$)	$O(n(\log r + \log\log_w(n/r)))$	$O(r \log\log_w(n/r))$
or	$O(n)$	$O(n)$
Optimal counting and locating (Thm. 4.10)	$O(n(\log r + \log\log_w(n/r)))$	$O(r \log\log_w(\sigma + n/r))$
or	$O(n + r(\log\log\sigma)^3)$	$O(n)$
RAM-optimal counting and locating (Thm. 4.11)	$O(n(\log r + \log\log_w(n/r)))$ $+ O(rw^2(\log\log_w n)^3)$ exp.	$O(rw \log_\sigma \log_w n)$
or	$O(n + rw^{1+\epsilon})$ exp.	$O(n)$
Text substrings (Thm. 5.1)	$O(n(\log r + \log\log_w(n/r)))$	$O(r \log(n/r))$
or	$O(n)$	$O(n)$
Accessing SA, ISA, and LCP (Thm. 5.4, 5.7 & 5.8)	$O(n(\log r + \log\log_w(n/r)))$	$O(r \log(n/r))$
or	$O(n)$	$O(n)$
Optimal counting/locating, $O(r \log(n/r))$ space (Thm. 5.9)	$O(n + r\log(n/r)(\log\log\sigma)^2)$ exp.	$O(n)$ (only if $r = \omega(n/\log_w^\epsilon \sigma)$)
Suffix tree (Thm. 6.1)	$O(n + r \log\log_w r)$	$O(n)$
with $O(Sort(n) + \log(n/r))$ I/Os	$O(n(\log r + \log\log_w(n/r)))$	$O(B + r\log(n/r))$
with $O(n/B + \log(n/r))$ I/Os, no ***TDepth*** & $\boldsymbol{LAQ_T}$	$O(n(\log r + \log\log_w(n/r)))$	$O(B + r\log(n/r))$

tree [Fredman and Willard 1993] on each bucket, which requires linear space and construction time, and constant query time.

If we are interested in deterministic construction time, we can resort to the so-called deterministic dictionaries, which use $O(t)$ space and can be built in time $O(t(\log\log t)^2)$ [Ružić 2008].

A *minimum perfect hash function (mphf)* maps the keys to the range $[1..t]$. This is trivial using $O(t)$ space (we just store the mapped value), but it is also possible to store a mphf within $O(t)$ bits, building it in $O(t)$ expected time and $O(t)$ space [Belazzougui et al. 2009b]. Such expected time holds w.h.p. as well if they use a distributor function towards $t' = O(t/\log t)$ buckets. For each bucket B_i, $i \in [1..t']$, they show that w.h.p. $O(\log t)$ trials are sufficient to find a perfect hash function $\sigma(i)$ for B_i, adding up to $O(t)$ time w.h.p. Further, the indexes $\sigma(i)$ found distribute geometrically (say, with a constant parameter p), and the construction also fails if their sum exceeds $\lambda \cdot t/p$ for some constant λ of our choice. The probability of that event is exponentially decreasing with t' for any $\lambda > 1$ [Janson 2017].

A *monotone mphf (mmphf)*, in addition, preserves the order of the keys. A mmphf can be stored in $O(t \log\log u)$ bits while answering in constant time. Its construction time and space is as for a mphf [Belazzougui et al. 2009a, Sec. 3] (see also Belazzougui et al. [2011, Sec. 3]). Therefore, all the expected cases we mention related to building perfect hash functions of any sort hold w.h.p. as well. Alternatively, their construction time can turn into worst-case w.h.p. of being correct.

The predecessor structure we use [Belazzougui and Navarro 2015, Thm. 14] uses $O(t)$ words and answers in time $O(\log\log_w(u/t))$. Its low-space version [Belazzougui and Navarro 2015, Sec. A.1 & A.2], using $O(t\log(u/t))$ bits, does not use hashing. It is a structure of $O(\log\log_w(u/t))$ layers, each being a bitvector of $O(t)$ bits. Its construction takes $O(t\log\log_w(u/t))$ worst-case time and $O(t)$ space.

Finally, note that if we can use $O(u)$ bits, then we can build a constant-time predecessor structure in $O(u)$ time, by means of rank queries on a bitvector.

A.2. Our basic structure

The basic structures of Section 2.5 can be built in $O(r)$ space. We start by using an $O(r)$-space construction of the run-length encoded BWT that scans T once, right to left, in $O(n \log r)$ time [Prezza 2016] (see also Ohno et al. [2018] and Kempa [2019]). The text T is not needed anymore from now on.

We then build the predecessor structure E that enables the LF-steps in time $O(\log\log_w(n/r))$. The construction takes $O(r\log\log_w(n/r)) \subseteq O(n)$ time and $O(r)$ space, as seen above. The positions p that start or end BWT runs are easily collected in $O(r)$ time from the run-length encoded BWT.

The structures to compute rank on L' in time $O(\log\log_w \sigma)$ [Belazzougui and Navarro 2015] also use predecessor structures. These are organized in r/σ chunks of size σ. Each chunk has σ lists of positions in $[1..\sigma]$ of lengths $\ell_1, \ldots, \ell_\sigma$, which add up to σ. The predecessor structure for the ith list is then built over a sample of $\ell_i/\log_w \sigma$ elements, in time $O((\ell_i/\log_w \sigma)\log\log_w(\sigma\log_w(\sigma)/\ell_i))$. Added over all the lists, this is $O((\sigma/\log_w \sigma)\log\log_w \sigma) \subseteq O(\sigma)$. The total construction time of this structure is then $O(r)$.

In total, the basic structures can be built in $O(n\log r)$ time and $O(r)$ space. Of course, if we can use $O(n)$ construction space, then we easily obtain $O(n)$ construction time, by building the suffix array in linear time and then computing the structures from it.

A.3. Fast locating

Structure E collects the starts of runs. In Section 3 we build two extended versions that collect starts and ends of runs. The first is a predecessor structure R (Lemma 3.2), which organizes the $O(r)$ run starts and ends separated by their character, on a universe of size σn. The second uses two predecessor structures (Lemmas 3.5 and 3.7), called P^+ and P^- in Lemma 3.7, which contain the BWT positions at distance at most s from run borders.

To build both structures, we simulate a backward traversal of T (using LF-steps from the position of the symbol \$) to collect the text positions of all the run starts and ends (for R), or all the elements at distance at most s from a run start or end (for P^+ and P^-). We use predecessor and successor queries on E (the latter are implemented without increasing the space of the predecessor structure) and accesses to L' to determine whether the current text position must be stored, and where. The traversal alone takes time $O(n\log\log_w(n/r))$ for the LF-steps.

The predecessor structure R is built in $O(r)$ space and $O(r\log\log(\sigma n/r)) \subseteq O(n\log\log\sigma) \subseteq O(n\log r)$ time (since $\sigma \le r$). The structures P^+ and P^- contain $O(rs)$ elements in a universe of size n, and thus are built in $O(rs)$ space and time $O(rs\log\log(n/(rs))) \subseteq O(n)$ (we never index more than n elements).

Overall, the structure of Theorem 3.6, enhanced as in Lemma 3.7, can be built in $O(rs)$ space and $O(n\log r + n\log\log_w(n/r))$ time. If we can use $O(n)$ space for the construction, then the LF-steps can be implemented in $O(1)$ time and the traversal requires $O(n)$ time. In this case, the structures R, P^+ and P^- can also be built in $O(n)$ time, since the predecessor searches can be implemented with bitvectors.

For the structure of Lemma 3.8 we follow the same procedure, building the structures P^+ and P^-. The classical algorithm to build the base LCP array [Kasai et al. 2001] uses $O(n)$ time and space. Within this space we can also build the predecessor structures in $O(n)$ time, as before. Note that this structure is not needed for Theorem 3.6, but in later structures. Using those, we will obtain a construction of LCP using less space (see Section A.6).

A.4. Optimal counting and locating

The first step of this construction is to build the compact trie that contains all the distinct substrings of length s of T. All these lie around sampled text positions, so we can simulate a backward traversal of T using E and L', as before, while maintaining a window of the last s symbols seen. Whenever we hit a run start or end in L, we collect the next $s-1$ symbols as well, forming a substring of length $2s-1$, and from there we restart the process, remembering the last s symbols seen.[28] This traversal costs $O(n\log\log_w(n/r))$ time as before.

The memory area where the edges of the compact trie will point is simply the concatenation of all the areas of length $2s-1$ we obtained. We now collect the s substrings of length s from each of these areas, and radix-sort the $O(rs)$ resulting strings of length s, in time $O(rs^2)$. After the strings are sorted, if we remove duplicates (getting σ^* distinct strings) and compute the longest common prefix of the consecutive strings, we easily build the compact trie structure in a single $O(\sigma^*)$-time pass. We then assign consecutive mapped values to the σ^* leaves and also assign the values $v_{\min}$ and $v_{\max}$ to the internal nodes. By recalling the suffix array and text positions each string comes from, we can also assign the values p and $SA[p]$ (or $SA^*[p]$) to the trie nodes. Further, we precompute the queries for $SA[p]$ in the structures P^+ and P^- in time $O(\sigma^*\log\log_w(n/(rs)) \subseteq O(rs\log\log_w(n/(rs)) \subseteq O(n)$.

To finish, we must create the perfect hash functions on the children of each trie node. There are $O(rs)$ children in total but each set stores at most σ children, so the total deterministic time to create the dictionaries is $O(rs(\log\log\sigma)^2)$. In total, we create the compact trie in time $O(n\log\log_w(n/r) + rs^2 + rs(\log\log\sigma)^2)$ and space $O(rs)$.

[28] If we hit other run starts or ends when collecting the $s-1$ additional symbols, we form a single longer text area including both text samples; we omit the details.

The construction of the RLFM-index of T^* can still be done within this space, without explicitly generating T^*, as follows. For each position $L[i]$, the BWT of T, we perform s LF-steps to obtain the metasymbol corresponding to $L^*[i]$, which we use to traverse the compact trie in order to find the mapped symbol $L^*[i]$. Since the values of L^* are obtained in increasing order, we can easily compress its runs on the fly, in $O(rs)$ space. The BWT of T^* is then obtained in time $O(ns \log\log_w(n/r))$. We can improve the time by obtaining this BWT run by run instead of symbol by symbol: We start from each run $L[x_1, y_1]$ and compute $x_2 = LF(x_1)$. From it, we find the end y_2 of the run x_2 belongs in L. The run for $s = 2$ is then $L[x_1, x_1 + \min(y_1 - x_1, y_2 - x_2)]$. We repeat the process s times until obtaining all x_k and y_k, $1 \le k \le s$. The next run of L^* is then $L^*[x_1, x_1 + \min_{1\le k\le s}(y_k - x_k)]$. The computation of each y_k from $x_k = LF^{k-1}(x_1)$ can be done by finding the predecessor of x_k in E and associating with each element in E the length of the run it heads, which is known when building E. In this way, the cost to compute the BWT of T^* decreases to $O(r^* s \log\log_w(n/r)) \subseteq O(rs^2 \log\log_w(n/r))$.

From the BWT, the other structures of the RLFM-index of T^* are built as in Section A.2, in time $O(r^* \log\log_w(n/r^*)) \subseteq O(n)$ and space $O(r^*) \subseteq O(rs)$. The array C^* is also built in $O(r^*) \subseteq O(rs)$ time and $O(\sigma^*) \subseteq O(rs)$ space.

To finish, we need to build the structure of Lemma 3.7, which as seen in Section A.3 is built in $O(rs)$ space and $O(n \log\log_w(n/r))$ time. With $s = \log\log_w(\sigma + n/r)$, the total construction time is upper bounded by $O(n \log\log_w(n/r) + r \log\log(\sigma + n/r)^3) \subseteq O(n \log\log_w(n/r) + r(\log\log\sigma)^3)$ and the construction space by $O(rs)$. When added to the $O(n \log r)$ time to build the BWT of T, the total simplifies to $O(n(\log r + \log\log_w(n/r)))$ because $\sigma \le r$.

If we can use $O(n)$ space for the construction, the LF-steps can be implemented in constant time. We can generate T^* explicitly and use linear-time and linear-space suffix array construction algorithms, so all the structures are built in time $O(n)$. The compact trie can be built by pruning at depth s the suffix tree of T, which is built in $O(n)$ time. We still need to build the perfect hash functions for the children, in deterministic time $O(rs(\log\log\sigma)^2)$. Added to $O(n)$, the total simplifies to $O(n + r(\log\log\sigma)^3)$.

When building the RAM-optimal version, the value of s grows to $w \log_\sigma \log_w n$. Further, the compact trie must be changed by the structure of Navarro and Nekrich [2017, Sec. 2]. In their structure, they jump by $\log_\sigma n$ symbols, whereas we jump by $w/\log\sigma$ symbols. Their perfect hash functions, involving $O(rs)$ elements, can be built in time $O(rs \log\log(rs)^2)$, whereas their weak prefix search structures [Belazzougui et al. 2018, Thm. 6] are built in expected time $O(rsw^\epsilon)$ for any constant $\epsilon > 0$. For the value of s used in this case, the time can be written as $O(rw^{1+\epsilon})$. Overall, the total construction time is in $O(n(\log r + \log\log_w(n/r)) + rw^2(\log\log_w n)^3)$. The construction space stays in $O(rs)$. If we can use $O(n)$ space, the expected construction time becomes $O(n + rs(\log\log(rs)^2) + rw^{1+\epsilon}) \subseteq O(n + rw^{1+\epsilon})$.

A.5. Access to the text

The structure of Theorem 5.1 can be built as follows. We first collect the text positions of starts and ends of BWT runs. Each sampled position induces a constant number of half-blocks at each of the $O(\log(n/r))$ levels (there are also r blocks of level 0). For each block or half-block, we must find its primary occurrence. We first find all their text endpoints in BWT with an LF-guided scan of T of time $O(n \log\log_w(n/r))$, after which we can read each block or half-block backwards in $O(\log\log_w(n/r))$ time per symbol. For each of them, we follow the method described in Lemma 4.5 to find its primary occurrence in $O(\log\log_w(\sigma + n/r))$ time per symbol, doing the backward search as we extract its symbols backwards too. Since at level l there are $O(r)$ blocks or half-blocks of length $O(n/(r \cdot 2^{l-1}))$, the total length of all the blocks and half-blocks adds up to $O(n)$, and the total time to find the primary occurrences is $O(n \log\log_w(\sigma + n/r))$.

We also need to fill in the text at the leaves of the structure. In the last level, then, we traverse the blocks in order to store their symbols in the structure, not to find their primary occurrence.

Therefore, the structure of Theorem 5.1 is built in $O(n \log\log_w(\sigma + n/r))$ time and $O(r \log(n/r))$ working space, once the basic structure of Lemma 2.1 is built.

In case of having $O(n)$ space for construction, we can replace predecessor structures with rank queries on bitvectors of n bits, but we still have the $O(\log\log_w \sigma)$ time for rank on L'. Thus the total time is $O(n \log\log_w \sigma)$. This is the most intuitive construction, yet we will slightly improve it in Section A.6.

A.6. Suffix array access

The other structures of Section 5 give access to cells of the suffix array (SA), its inverse (ISA), and the longest common prefix array (LCP).

The structure of Theorem 5.4 is analogous to that of Theorem 5.1: it has $O(\log(n/r))$ levels and $O(r)$ blocks or half-blocks of length $s_l = n/(r \cdot 2^{l-1})$ at each level l. However, its domain is the suffix array cells and the way to find a primary occurrence of each block is different. At each level, we start with any interval of length s_l and compute LF on its left extreme. This leads to another interval of length s_l. We repeat the process until completing the cycle and returning to the initial interval. Along the way, we collect all the intervals that correspond to blocks or half-blocks of this level. Each time the current interval contains or immediately follows a sampled BWT position in E, we make it the primary occurrence of all the blocks or half-blocks collected so far (all those must coincide with the content of the current block or half-block), and reinitialize an empty set of collected blocks. This process takes $O(n \log\log_w(n/r))$ time for a fixed level. We can perform a single traversal for all the levels simultaneously, storing all the blocks in a dictionary using the left extreme as their search key. As we traverse BWT, we collect the blocks of all lengths starting at the current position p. Further, we find the successor of $p-1$ in E to determine the minimum length of the blocks that cover or follow the nearest sampled position, and all the sufficiently long collected blocks find their primary occurrence starting at p. The queries on E also amount to $O(n \log\log_w(n/r))$ time.

This multi-level process requires a dictionary of all the $O(r\log(n/r))$ blocks and half-blocks. If we implement it as a predecessor structure, it takes $O(r\log(n/r))$ space, it is constructed in $O(r\log(n/r)\log\log_w(n/r))$ time, and answers the $O(n)$ queries in time $O(n\log\log_w(n/r))$. The collected segments can be stored separated by length, and the $O(\log(n/r))$ lengths having collected blocks can be marked in a small bitvector, where we find the nonempty sets over some length in constant time.

We also need to fill the DSA cells of the leaves of the structure. This can be done with an additional traversal of BWT, filling in the SA values at the required positions whenever we reach them in the traversal. We can then easily convert SA to DSA values in the leaves. This does not add extra time or space, asymptotically.

A.7. Inverse suffix array, and again text access

The construction of the structures of Theorem 5.7 is analogous to that of Section A.6. This time, the domain of the blocks and half-blocks are the text positions and, instead of traversing with LF, we must use ϕ. This corresponds to traversing BWT right to left, keeping track of the corresponding position in T. We can maintain the text position using our basic structure of Lemma 3.5. Then, if the current text position is i, we can use the predecessor structures on T to find the first sampled position following $i-1$, to determine which collected blocks have found their primary occurrence. We can similarly fill the required values $DISA$ by traversing BWT right-to-left and writing the appropriate ISA values. Therefore, we can build the structures within the same cost as Theorem 5.4.

In both cases, if we have $O(n)$ space available for construction, we can build the structures in $O(n)$ time, since LF can be computed in constant time and all the dictionaries and predecessor structures can be implemented with bitvectors. We can also use these ideas to obtain a slightly faster construction for the structures of Theorem 5.1, which extract substrings of T.

LEMMA A.1. *Let $T[i-1\,..\,i]$ be within a phrase. Then it holds that $\phi(i-1) = \phi(i)-1$ and $T[i-1] = T[\phi(i)-1]$.*

PROOF. The fact that $\phi(i-1) = \phi(i)-1$ is already proved in Lemma 5.5. From that proof it also follows that $T[i-1] = BWT[p] = BWT[p-1] = T[j-1] = T[\phi(i)-1]$. □

LEMMA A.2. *Let $T[i-1\,..\,i+s]$ be within a phrase, for some $1 < i \le n$ and $0 \le s \le n-i$. Then there exists $j \ne i$ such that $T[j-1\,..\,j+s-1] = T[i-1\,..\,i+s-1]$ and $[j-1\,..\,j+s]$ contains the first position of a phrase.*

PROOF. The proof is analogous to that of Lemma 5.6. By Lemma A.1, it holds that $T[i'-1\,..\,i'+s-1] = T[i-1\,..\,i+s-1]$, where $i' = \phi(i)$. If $T[i'-1\,..\,i'+s]$ contains the first position of a phrase, we are done. Otherwise, we apply Lemma A.1 again on $[i'-1\,..\,i'+s]$, and repeat until we find a range that contains the first position of a phrase. This search eventually terminates because ϕ is a permutation with a single cycle. $\square$

We can then find the primary occurrences for all the blocks in Theorem 5.1 analogously as for $DISA$ (Theorem 5.7). We traverse T with ϕ (i.e., we traverse BWT right to left, using Lemma 3.5 to compute ϕ each time). This time we index the blocks and half-blocks using their right extreme, collecting all those that end at the current position i of T. Then, at each position i, we use the predecessor structures on T to find the nearest sampled position preceding $i+1$, to determine which collected blocks and half-blocks have found their primary occurrence. We can similarly fill the required values of T with a final traversal of BWT, accessing L'. Therefore, we can build these structures within the same cost of Theorem 5.7.

A.8. Longest common prefix array

Finally, the construction for LCP access in Theorem 5.8 is a direct combination of Theorem 5.4 (i.e., SA) and Lemma 3.8 (i.e., ***PLCP*** extension, with $s = \log(n/r)$). In Section A.3 we saw how to build the latter in $O(n)$ time and space. Within $O(n)$ space, we can also build the structure of Theorem 5.8 in $O(n)$ time. We can, however, build the structure of Lemma 3.8 within $O(r\log(n/r) + rs)$ space if we first build SA, ISA, and the extraction structure. The classical linear-time algorithm [Kasai et al. 2001] is as follows: we compare $T[SA[2]\,..]$ with $T[SA[1]\,..]$ until they differ; the number ℓ of matching symbols is $LCP[2]$. Now we jump to compute $LCP[\Psi(2)]$, where $\Psi(p) = ISA[(SA[p] \bmod n) + 1]$ is the inverse of LF [Grossi and Vitter 2006]. Note that $LCP[\Psi(2)] = lcp(T[SA[\Psi(2)]\,..], T[SA[\Psi(2)-1]\,..]) = lcp(T[SA[2]+1\,..], T[SA[\Psi(2)-1]\,..])$ and, if $\ell > 0$, this is at least $\ell-1$ because $T[SA[2]+1\,..]$ already shares the first $\ell - 1$ symbols with some lexicographically smaller suffix, $T[SA[1]+1\,..]$. Thus the comparison starts from the position ℓ onwards: $LCP[\Psi(2)] = \ell-1+lcp(T[SA[\Psi(2)]+\ell-1\,..], T[SA[\Psi(2)-1]+\ell-1\,..])$. This process continues until the cycle Ψ visits all the positions of LCP.

We can simulate this algorithm, traversing the whole virtual array $LCP[1\,..\,n]$ but writing only the $O(rs)$ relevant cells, that is, those at distance s from a run border. We first build P^+ and P^- as for Lemma 3.8. We then traverse T backwards virtually, using LF, in time $O(n \log\log_w(n/r))$, spotting the positions in $P^\pm = P^+ \cup P^-$. Say we find $p \in P^\pm$ and the previous $p' \in P^\pm$ was found d steps ago. This means that $p' = \Psi^d(p)$ is the next relevant suffix after p along the LCP algorithm. We store $next[f(p)] = \langle p', d\rangle$, where $next$ is a table aligned with LCP'. Once this pass is complete, we simulate the algorithm starting at the last relevant p value we found: we compute $LCP[p] = \ell$ and store $LCP'[f(p)] = \ell$. Then, if $next[f(p)] = \langle p', d\rangle$, we set $p = p'$ and $\ell = \max(1, \ell - d)$. Along the process, we carry out $O(rs)$ string comparisons for a total of $O(n)$ symbols. Each string comparison takes time $O(\log(n/r))$ in order to compute ISA. We extract the desired substrings of T by chunks of $\log(n/r)$ symbols, so that comparing ℓ symbols costs $O(\ell + \log(n/r))$. Overall, the traversal takes time $O(n + rs\log(n/r))$, plus the $O(n\log\log_w(n/r))$ time to compute $next$. Added to the $O(n\log r + n\log\log_w(n/r))$ time needed in Section A.3 to build the sampling structures, we have a total time of $O(n\log r + n\log\log_w(n/r) + rs\log(n/r))$, within $O(r\log(n/r) + rs)$ space. For $s = \log(n/r)$, as required in Theorem 5.8, the space is $O(r\log(n/r))$ and the time is $O(n\log r + n\log\log_w(n/r))$, because $O(rs\log(n/r)) = O(r\log^2(n/r)) \subseteq O(n)$.

A.9. Optimal counting and locating in space $O(r\log(n/r))$

To obtain optimal counting and locating in space $O(r\log(n/r))$, we only need to care about the case $r \geq n/\log n$, so the allowed space becomes $\Omega(n\log\log n)$ bits.[29] In this case we use an $O(n)$-bit compressed suffix tree enriched with the structures of Belazzougui and Navarro [2014, Lem. 6]. This requires, essentially, the suffix tree topology represented with parentheses, edge lengths (capped to $O(\log\log n)$ bits), and mmphfs on the first letters of the edges towards the nodes' children. The parentheses and edge lengths are obtained directly left-to-right, with a sequential pass over LCP [Kasai et al. 2001; Sadakane 2007]. If we use $O(n)$ space for the construction, the first letters are obtained directly from the suffix array and the text, all in $O(n)$ time. The construction of the mmphfs on (overall) $O(n)$ elements can be done in $O(n)$ expected time. The compressed suffix tree includes, in addition,

[29] In fact, the condition is $r = \omega(n/\log_w^\epsilon \sigma)$, which would allow us using any space in $O(n\log^{1-\epsilon} n)$, for example, but we do not know of a representation larger and better than the one we are using.

the structures to extract substrings of T and entries of SA, and a compact trie on the distinct strings of length $\log(n/r)$ in T. With $O(n)$ space, these and the other structures of Section A.4 are built in $O(n + rs(\log\log\sigma)^2) = O(n + r\log(n/r)(\log\log\sigma)^2)$ expected time.

A.10. Suffix tree

The suffix tree needs the compressed representations of SA, ISA, and LCP. While these can be built in $O(r\log(n/r))$ space, the suffix tree construction will be dominated by the $O(n)$ space used to build the RLCFG on $DLCP$ in Lemmas 6.5 and 6.6. Thus, we build SA, ISA, and LCP in $O(n)$ time and space.

Starting from the plain array $DLCP[1..n]$, the RLCFG is built in $O(\log(n/r))$ passes of the $O(n)$-time algorithm of Jeż [2015]. This includes identifying the repeated pairs, which can also be done in $O(n)$ time via radix sort. The total time is also $O(n)$, because the lengths of the strings compressed in the consecutive passes decrease exponentially.

All the fields l, d, p, m, n, etc. stored for the nonterminals are easily computed in $O(r\log(n/r)) \subseteq O(n)$ time, that is, $O(1)$ per nonterminal. The arrays L, A, and M are computed in $O(r)$ time and space. The structure RMQ_M is built in $O(r)$ time and bits [Fischer and Heun 2011]. Finally, the structures that solve PSV' and NSV' queries on $DLCP'$ (construction of the tree for the weighted level-ancestor queries [Fischer and Heun 2011], the data structure on this tree [Kopelowitz and Lewenstein 2007], and the simplification for $\mathrm{PSV}/\mathrm{NSV}$ with large r [Fischer et al. 2009]), as well as the approximate median of the minima [Fischer and Heun 2010], are built in $O(r)$ time and space, as shown by their authors.

This does not count the construction of the predecessor data structures for the weighted level-ancestor queries, however. This requires creating several structures with $O(r)$ elements in total, on universes of size n, having at least n/r elements in each structure. The total construction time is then $O(r\log\log_w r)$. Note that these predecessor structures are needed for $\mathrm{PSV}'/\mathrm{NSV}'$, but also for $\mathrm{PSV}/\mathrm{NSV}$; the special $O(\log(n/r))$-time solution we use for the latter applies only when r is large.

In addition, the suffix tree requires the construction of the compressed representation of $PTDE$ [Fischer et al. 2009]. This is easily done in $O(n)$ space and time by traversing a classical suffix tree.

External memory. We note that, with $O(n/B + \log(n/r))$ I/Os (where B is the external memory block size), we can build most of the suffix tree in main memory space $O(B + r\log(n/r))$. The main bottleneck is the algorithm of Jeż [2015]. The algorithm starts with two sequential passes on $DLCP$, first identifying runs of equal cells (to collapse them into one symbol using a rule of the form $X \rightarrow Y^t$) and second collecting all the distinct pairs of consecutive symbols (to create some rules of the form $X \rightarrow YZ$). Both kinds of rules will add up to $O(r)$ per pass, so the distinct pairs can be stored in a balanced tree in main memory using $O(r)$ space. Once the pairs to replace are defined (in $O(r)$ time [Jeż 2015]), the algorithm traverses the text once again, doing the replacements. The new array is of length at most $(3/4)n$; repeating this process $O(\log(n/r))$ times will yield an array of size $O(r)$, and then we can finish. By streaming the successively smaller versions of the array to external memory, we obtain the promised I/Os and main memory space. The computation time is dominated by the cost of building the structures SA, ISA, and LCP in $O(r\log(n/r))$ space: $O(n(\log r + \log\log_w(n/r))$. The balanced tree operations add another $O(n\log r)$ time to this complexity.

The other obstacle is the construction of $PTDE$, needed for the operations ***TDepth*** and $\boldsymbol{LAQ}_T$. This array can be built in additional $O(Sort(n))$ I/Os, $O(n)$ computation, and $O(r)$ main memory space by emulating the linear-time algorithm to build the suffix tree topology from the LCP array [Kasai et al. 2001]. This algorithm traverses LCP left to right, and maintains a stack of the internal nodes in the current rightmost path of the suffix tree, each with its string depth (the stack is easily maintained on disk with $O(n/B)$ I/Os). Each new $LCP[p]$ cell represents a new suffix tree leaf. For each such leaf, we pop nodes from the stack until we find a node whose string depth is $\le LCP[p]$. The sequence of stack sizes is the array TDE. We write those TDE entries to disk as they are generated, left to right, in the format $\langle TDE[p], SA[p]\rangle$. Once this array is generated on disk, we sort it by the second component, and then the sequence of first components is the array $PTDE$. This array is then read from disk left to right, as we simultaneously fill the run-length compressed bitvector that represents it in $O(r)$ space [Fischer et al. 2009]. The left-to-right traversal of LCP and SA is done in $O(n)$ time by accessing their compressed representation by chunks of $\log(n/r)$ cells, using Theorems 5.4 and 5.8 with $s = \log(n/r)$.