Noname manuscript No.
(will be inserted by the editor)

Efficient Indexes for Jumbled Pattern Matching with Constant-Sized Alphabet

Tomasz Kociumaka · Jakub Radoszewski · Wojciech Rytter

Received: date / Accepted: date

Abstract We introduce efficient indexes for a problem in non-standard stringology: jumbled pattern matching. An index is a data structure constructed for a text of length n over an alphabet of size σ that can answer queries asking if the text contains a fragment which is jumbled (Abelian) equivalent to a pattern, specified by its so-called Parikh vector. We denote the length of the pattern by m. Moosa and Rahman [Journal of Discrete Algorithms, 2012] gave an index for the case of binary alphabets with $\mathcal{O}(\frac{n^2}{(\log n)^2})$-time construction in the word-RAM model. Several earlier papers stated as an open problem the existence of an efficient solution for larger alphabets. In this paper we develop an index for any constant-sized alphabet. The construction involves a trade-off parameter, which in particular lets us achieve the following complexities: $\mathcal{O}(n^{2-\delta})$ space and $\mathcal{O}(m^{(2\sigma-1)\delta})$ query time for any $0 < \delta < 1$, or $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ space and polylogarithmic, $o(\log^{2\sigma-1} m)$, query time. The construction time in both cases is subquadratic: $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ in the word-RAM model (using bit-parallelism). Our construction algorithms are randomized (Las Vegas, running time w.h.p.), which is due to the usage of perfect hashing. On the other hand, all queries are answered deterministically.

A preliminary version of this work appeared at ESA 2013 [18]. Here we improve it in several ways. We achieve $o(n^2)$-time construction of the index with $\mathcal{O}(n^{2-\delta})$ space and $\mathcal{O}(m^{(2\sigma-1)\delta})$ query time, which was not present in [18]. We also extend the index so that the position of the leftmost occurrence of the query pattern is provided at no additional cost in the complexity; this required rather nontrivial changes in the construction algorithm.

Keywords jumbled indexing · jumbled pattern matching · Abelian equivalence · histogram indexing

Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter
Institute of Informatics, University of Warsaw, Poland
E-mail: [kociumaka,jrad,rytter]@mimuw.edu.pl
Corresponding author: Tomasz Kociumaka, Tel.: +48-22-55-44-579, Fax: +48-22-55-44-400

1 Introduction

The problem of *jumbled pattern matching* is a variant of the standard pattern matching problem. The *match* between a given pattern and a factor of the text is defined by their commutative (Abelian) equivalence: one word can be obtained from the other by permuting its symbols. This relation can be conveniently described using *Parikh vectors*, which show frequency of each symbol of the alphabet in a word: u and v are commutatively equivalent (denoted as $u \approx v$) if and only if their Parikh vectors are equal. We consider the following problem of constructing an index for jumbled pattern matching; see Fig. 1.

> Indexing for jumbled pattern matching
>
> **Preprocessing Input:** a text x of length n over an alphabet of size σ.
>
> **Query:** for a given Parikh vector (jumbled pattern) p of length m, decide whether p occurs in the text x and, if so, find its leftmost occurrence.

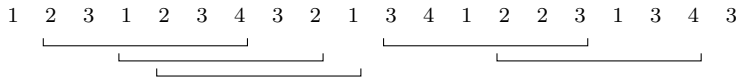

Fig. 1 All occurrences of a jumbled pattern $p = (1, 2, 2, 1)$ in the text $x = 1\,2\,3\,1\,2\,3\,4\,3\,2\,1\,3\,4\,1\,2\,2\,3\,1\,3\,4\,3$. The leftmost occurrence is at position 2.

The binary case. Most results related to indexes for jumbled pattern matching so far have been obtained for binary words. Cicalese et al. [11] proposed an index with $\mathcal{O}(n)$ size and $\mathcal{O}(1)$ query time and gave an $\mathcal{O}(n^2)$-time construction algorithm for the index. The key observation used in this index is that if a binary word contains two factors of length ℓ containing i and j ones respectively, then it must contain a factor of length ℓ with any intermediate number of ones. The index provides only a yes/no answer for a query pattern; additional $\mathcal{O}(\log n)$ time can be used to restore a witness occurrence [12]. The construction time was improved independently by Burcsi et al. [6] (see also [7, 8]) and Moosa and Rahman [20] to $\mathcal{O}(\frac{n^2}{\log n})$, and then by Moosa and Rahman [21] to $\mathcal{O}(\frac{n^2}{(\log n)^2})$. All these results work in the word-RAM model. For trees vertex-labeled with $\{0, 1\}$ an index, with $\mathcal{O}(\frac{n^2}{(\log n)^2})$ construction time, $\mathcal{O}(n)$ size and $\mathcal{O}(1)$ query time was given in [15]. Hermelin et al. [17] reduced binary jumbled indexing to all-pairs shortest paths problem and used the latest results of Williams for the latter problem [23] to obtain preprocessing time of $\mathcal{O}(\frac{n^2}{2^{\Omega((\log n/\log\log n)^{0.5})}})$ for binary jumbled indexing on both words and trees (a similar reduction was shown by Bremner at el. [5]). The general problem of computing an index for jumbled pattern matching in graphs is known to be NP-complete [13, 19] but fixed-parameter tractable by the pattern size [13] (see also [4]).

Indexes for larger alphabets. For arbitrary alphabets, Amir et al. [1] presented an index with $\mathcal{O}(n^{1+\varepsilon})$ space, $\mathcal{O}(n^{1+\varepsilon}\log\sigma)$ construction time and $\mathcal{O}(m^{\frac{1}{\varepsilon}}+\log\sigma)$ query time for any positive $\varepsilon<1$. Nevertheless, this query time is $o(n)$ only for $m=o(n^{\varepsilon})$. Jumbled pattern matching in a run-length encoded text over arbitrary alphabet was considered in [9].

Amir et al. [2] presented hardness results for jumbled indexing over large alphabets. They showed that, under 3SUM-hardness assumption, for $\sigma=\omega(1)$ jumbled indexing requires $\Omega(n^{2-\epsilon})$ preprocessing time or $\Omega(n^{1-\delta})$ query time for every $\epsilon,\delta>0$. Furthermore, under strong 3SUM-hardness assumption, for $\sigma\geq 3$ jumbled indexing requires $\Omega(n^{2-\epsilon_\sigma})$ preprocessing time or $\Omega(n^{1-\delta_\sigma})$ query time, where $\epsilon_\sigma,\delta_\sigma<1$ are computable constants. Recall that the 3SUM problem asks if one can choose elements $a\in A$, $b\in B$, and $c\in C$ from given integer sets A,B,C so that $a+b=c$. It is believed that this problem cannot be solved in strongly subquadratic time; for precise formulations of the related hardness assumptions, see [2].

Several researchers (see, e.g., [8,20,21]) posed an open problem asking for a construction of an $o(n^2)$ indexing scheme with $o(n)$ query time for general alphabets. In particular, even for a ternary alphabet none was known, since the basic observation used to obtain a binary index is not applicable to any larger alphabet.

Our results. We prove that the answer for the open problem asking for a subquadratic jumbled index with sublinear-time queries is positive for any constant-sized alphabet. We show an index of size $\mathcal{O}(\frac{n^2}{L})$ answering queries in $\mathcal{O}(L^{2\sigma-1})$ time where L is a trade-off parameter which can attain any given value between 1 and n. For some choices of L we also improve the query time so that it depends on the pattern size m only. More precisely, we show an index of size $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ which enables queries in $\mathcal{O}\big(\big(\frac{\log m}{(\log\log m)^2}\big)^{2\sigma-1}\big)$ time, and for any $0<\delta<1$ an index of size $\mathcal{O}(n^{2-\delta})$ with $\mathcal{O}(m^{\delta(2\sigma-1)})$-time queries. Both these variants take $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ time to construct, and the query algorithm provides the leftmost occurrence of the pattern if it exists. Our index works in the word-RAM model with word size $\Omega(\log n)$ [16], and the construction algorithm uses bit-parallelism. The construction algorithm is randomized (Las Vegas, running time w.h.p.) due to perfect hashing. On the other hand, all query algorithms are deterministic.

After the submission of this journal paper, Chan and Lewenstein [10] presented a breakthrough work where they improved the construction time of the binary index to $\mathcal{O}(n^{1.859})$. They also obtained an index with strongly subquadratic construction and strongly sublinear queries for larger alphabets. Moreover, Chan and Lewenstein provide an extension of our idea of heavy and light factors which improves query time in the index.

Organization of the paper. Section 2 is devoted mostly to combinatorial properties of Parikh vectors. In Section 3 we describe the basic version of the

index together with the query algorithm. The size of the index is $\mathcal{O}(\frac{n^2}{L})$ and the queries are answered in $\mathcal{O}(L^{2\sigma-1})$ time. We also present a naive $\mathcal{O}(n^2)$-time construction algorithm. In Section 4 we introduce variants of the index whose query time depends on the pattern size rather than the text size. Using the auxiliary tools based on bit parallelism in the word-RAM model that we provide in Section 5, in Section 6 we develop a subquadratic-time construction algorithm for the index. Section 7 contains some concluding remarks.

2 Preliminaries

In this paper we assume that the alphabet Σ is $\{1, 2, \ldots, \sigma\}$ for $\sigma = \mathcal{O}(1)$. Let $x \in \Sigma^n$. By x_i (for $i \in \{1, \ldots, n\}$) we denote the i-th letter of x. A word of the form $x_i \ldots x_j$, also denoted as $x[i..j]$, is called a *factor* of x. We say that the factor $x[i..j]$ *occurs* at the position i. A factor of the form $x[1..i]$ is called a *prefix* of x. If $i > j$ then $x[i..j]$ represents an empty word.

A *Parikh vector* is a vector of dimension σ with non-negative integer components. Parikh vectors can be used to describe frequencies of letters in a word. Let $\#_s(x)$ denote the number of occurrences of the letter s in x. Then the Parikh vector $\mathcal{P}(x)$ of a word $x \in \Sigma^*$ is defined as:

$$\mathcal{P}(x) \;=\; (\#_1(x),\, \#_2(x),\, \#_3(x),\, \ldots,\, \#_\sigma(x)). \tag{1}$$

Example 2.1 $\mathcal{P}(1\,2\,3\,1\,2\,3\,4\,3\,2\,1\,3\,4\,1\,2\,2\,3\,1\,3\,4\,3) \;=\; (5,\, 5,\, 7,\, 3)$.

We say that words x and y are *commutatively equivalent* (denoted as $x \approx y$) if y can be obtained from x by a permutation of its letters. Observe that we have:

$$x \approx y \iff \mathcal{P}(x) = \mathcal{P}(y).$$

Example 2.2 $1\,2\,2\,1 \approx 2\,2\,1\,1$, since $\mathcal{P}(1\,2\,2\,1) = (2, 2) = \mathcal{P}(2\,2\,1\,1)$.

For a fixed word x, we define $Occ(p)$ as the set of the starting positions of all occurrences of factors of x with Parikh vector p. For the zero Parikh vector $\bar{0}_\sigma$ we assume that $Occ(\bar{0}_\sigma) = \{1, \ldots, n+1\}$. If $Occ(p) \neq \emptyset$, we say that p *occurs* in x (at each position $i \in Occ(p)$), or that p is an *Abelian factor* of x.

Example 2.3 Let $x = 1\,2\,3\,1\,2\,3\,4\,3\,2\,1\,3\,4\,1\,2\,2\,3\,1\,3\,4\,3$. A factor $2\,3\,1\,2\,3\,4$ occurs (as a subword) at position 2. We have $\mathcal{P}(2\,3\,1\,2\,3\,4) \;=\; (1, 2, 2, 1)$, hence the Parikh vector $p = (1, 2, 2, 1)$ also occurs at position 2. There are several other occurrences of the Abelian factor p; see Fig. 1. We have:

$$Occ((1, 2, 2, 1)) = \{2, 4, 5, 11, 14\}.$$

The remainder of this section is devoted to combinatorial and algorithmic properties of Parikh vectors. We define the *norm* of a Parikh vector $p = (p_1, p_2, \ldots, p_\sigma)$ as:

$$|p| \;=\; \sum_{i=1}^{\sigma} |p_i|. \tag{2}$$

For two Parikh vectors p, q, by $p + q$ we denote their component-wise sum and by $p - q$ their component-wise difference. The latter is well-defined only if $p \geq q$, i.e., if $p_i \geq q_i$ for each $i \in \Sigma$.

For a fixed integer r we define the *extension sets* of Parikh vectors:

$$Ext^{+}_{=r}(p) = \{p + p' \,:\, |p'| = r,\ \forall i\ p'_i \geq 0\}, \tag{3}$$

$$Ext^{-}_{=r}(p) = \{p - p' \,:\, |p'| = r,\ \forall i\ p_i \geq p'_i \geq 0\}, \tag{4}$$

$$Ext^{+}_{<r}(p) = \{p + p' \,:\, |p'| < r,\ \forall i\ p'_i \geq 0\}. \tag{5}$$

In other words, $Ext^{+}_{<r}(p)$ is the open ball of radius r centered in p, and $Ext^{+}_{=r}(p)$ is the sphere of radius r centered in p (in both cases restricted to points with integer coordinates and to the non-negative hyperoctant with origin p). $Ext^{-}_{=r}(p)$ is similar to $Ext^{+}_{=r}(p)$; the only difference is that we take care to subtract only those Parikh vectors p' that yield non-negative resulting components of $p - p'$.

Lemma 2.4 *For any Parikh vector p and integer $r \geq 0$:*

(a) $|Ext^{+}_{=r}(p)|, |Ext^{-}_{=r}(p)| \leq 2r^{\sigma-1}$;
(b) $|Ext^{+}_{<r}(p)| \leq r^{\sigma}$.

Proof Both $|Ext^{+}_{=r}(p)|$ and $|Ext^{-}_{=r}(p)|$ are bounded by the number of Parikh vectors of norm r, which is $\binom{r+\sigma-1}{\sigma-1}$, since each such Parikh vector corresponds to a placement of r indistinguishable balls into σ distinguishable urns ('letters'). For $r \leq 1$ this is $\mathcal{O}(1)$, since $\sigma = \mathcal{O}(1)$, and for $\sigma = 1$ this is 1. Otherwise:

$$\binom{r+\sigma-1}{\sigma-1} \;=\; \frac{r+1}{1}\frac{r+2}{2}\cdots\frac{r+\sigma-1}{\sigma-1} \;\leq\; (r+1)r^{\sigma-2} \;\leq\; 2r^{\sigma-1}.$$

This proves part (a).

To bound $|Ext^{+}_{<r}(p)|$, it suffices to observe that

$$Ext^{+}_{<r}(p) = \bigcup_{k=0}^{r-1} Ext^{+}_{=k}(p).$$

Thus the size of the set is at most

$$\sum_{k=0}^{r-1}\binom{k+\sigma-1}{\sigma-1} \;=\; \binom{r+\sigma-1}{\sigma} \;=\; \frac{r}{1}\frac{r+1}{2}\cdots\frac{r+\sigma-1}{\sigma} \;\leq\; r^{\sigma}.$$

This proves part (b). □

Let us also introduce an efficient tool for determining Parikh vectors of factors of a given word.

Lemma 2.5 *A text x of length n can be preprocessed in $\mathcal{O}(n)$ time so that for any $1 \leq i \leq j \leq n$, the Parikh vector $\mathcal{P}(x[i..j])$ can be computed in $\mathcal{O}(1)$ time.*

Proof We precompute $\mathcal{P}(x[1..k])$ for all $k = 0, \ldots, n$ in $\mathcal{O}(n)$ time. Then

$$\mathcal{P}(x[i..j]) \;=\; \mathcal{P}(x[1..j]) - \mathcal{P}(x[1..i-1]).$$

Subtraction of Parikh vectors takes constant time, since $\sigma = \mathcal{O}(1)$. □

3 Index with sublinear-time queries

In this section we describe an index for jumbled pattern matching which has subquadratic size and allows sublinear-time queries. We also show a simple $\mathcal{O}(n^2)$-time construction of the index. Let us fix a trade-off parameter $L \in \{1, \ldots, n\}$; the space of the index and the query time will depend on L.

3.1 A sketch of the algorithm

The intuition behind the index is as follows. We explicitly store all Abelian factors of the text whose norm is a multiple of L (L-*factors*) together with all their occurrences in x. There are only $\mathcal{O}(\frac{n^2}{L})$ occurrences of such Abelian factors and thus we are aiming at $\mathcal{O}(\frac{n^2}{L})$ space. Hence, if a query Parikh vector has norm divisible by L, we can answer the query immediately. Otherwise the query Parikh vector, say q, has norm $kL + r$ for some $0 < r < L$. If q indeed occurs in the text at some position i (assume that it is the leftmost occurrence), then we may consider the Abelian factor p of length kL occurring at the same position. We say that the L-factor p *generates* the Abelian factor q; see Fig. 2. In the index we find the position i differently when the generating L-factor p turns out to have few occurrences in x (then p is a so-called *light* L-factor) and differently if it turns out to have many occurrences in x (then p is a *heavy* L-factor).

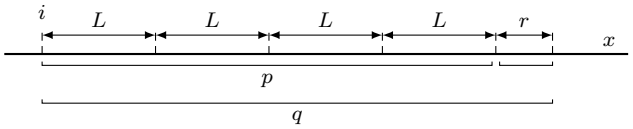

Fig. 2 An Abelian factor q occurring at position i and its generating L-factor p.

The "Light" Case. If p is a light L-factor, then we can afford to iterate through all its occurrences in x. To find the position i we use the following observation:

Observation 3.1 *$p \in Ext^{-}_{=r}(q)$ and $i \in Occ(p)$.*

Hence, the query for q is answered by iterating through all possible generating L-factors $p \in Ext^{-}_{=r}(q)$, filtering out those which are not light L-factors, and then iterating through all elements $i \in Occ(p)$. Lemma 2.4(a) limits the size of the set $Ext^{-}_{=r}(q)$ and the size of the set $Occ(p)$ is bounded due to the fact that p is light.

The "Heavy" Case. For each heavy L-factor p we store all Abelian factors generated by it (this set is denoted as $\mathcal{D}_L(p)$), together with their leftmost occurrences generated by p. To answer a query for a Parikh vector q in this case, we simply return the precomputed answer.

We need to argue that the space used here is small enough. A single L-factor p generates at most $|Ext^+_{<L}(p)|$ different Abelian factors. Even though a heavy L-factor has many occurrences in x, the total number of occurrences of heavy L-factors is still $\mathcal{O}(\frac{n^2}{L})$. Hence, we can afford $\mathcal{O}(\sum_{p\text{ heavy}} |Occ(p)|)$ space. If we pick the threshold value on the number of occurrences of light/heavy L-factors so that every heavy L-factor p satisfies the condition $|Occ(p)| \geq |Ext^+_{<L}(p)|$, then we can indeed afford to store all Abelian factors generated by each heavy L-factor.

In Subsection 3.2 we formally define the notions of light and heavy L-factors and the set $\mathcal{D}_L$. Then in Subsection 3.3 we give a full description of the data structure and the query algorithm. A simple construction of the index (subject to improvement in the later sections) is shown in Subsection 3.4. Finally in Subsection 3.5 we analyze the complexity of the index depending on the trade-off parameter L.

3.2 Combinatorial tools

By $\mathcal{F}_L$ we denote the set of all Abelian factors of the text x whose norm is divisible by L:

$$\mathcal{F}_L \;=\; \{\mathcal{P}(x[i..j]) \;:\; 1 \leq i \leq j \leq n,\, L \text{ divides } j+1-i\} \cup \{\bar{0}_\sigma\}. \qquad (6)$$

Elements of $\mathcal{F}_L$ are called *L-factors*. The following observation gives a simple property of the set of L-factors.

Fact 3.2 $\sum_{p\in\mathcal{F}_L} |Occ(p)| \leq \frac{n^2}{L} + n + 1$.

Proof The following sequence of inequalities:

$$\sum_{p\in\mathcal{F}_L} |Occ(p)| \leq n+1+\sum_{k=1}^{\lfloor n/L\rfloor} \sum_{p\in\mathcal{F}_L, |p|=kL} |Occ(p)| \leq n+1+\tfrac{n}{L}\cdot n = \tfrac{n^2}{L}+n+1.$$

proves the claimed result. □

We divide the L-factors into two sets: the set $\mathcal{L}_L$ of *light* L-factors (with few occurrences):

$$\mathcal{L}_L = \{p \in \mathcal{F}_L \,:\, |Occ(p)| \leq L^\sigma\} \qquad (7)$$

and the set of the remaining, *heavy* L-factors denoted by $\mathcal{H}_L$.

Example 3.3 Let $x = 2\,2\,2\,1\,1\,1\,2\,1\,2\,1\,1\,2\,2\,1\,1\,2\,2\,1\,2\,1$ and $L = 3$. For this text $\mathcal{H}_L = \{(0,0),(2,1),(3,3)\}$, since each of these L-factors has more than $L^\sigma = 3^2 = 9$ occurrences in x. For example, there are eleven occurrences of the heavy L-factor $(2,1)$; see Fig. 3. All the remaining L-factors in this example are light, i.e.,

$$\mathcal{L}_L = \{(0,3),(1,2),(3,0),(4,2),(4,5),(5,4),(6,3),(6,6),\ldots\}.$$

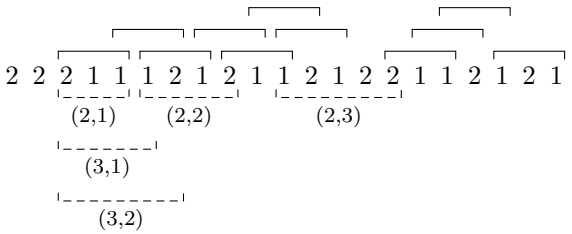

Fig. 3 Occurrences of a heavy 3-factor $(2,1)$ in $x = 2\,2\,2\,1\,1\,1\,2\,1\,2\,1\,1\,2\,2\,1\,1\,2\,2\,1\,2\,1$. We have $\mathcal{D}_3((2,1)) = \{(2,1), (3,1), (3,2), (2,2), (2,3)\}$ (underlined with dashed lines).

For a Parikh vector $p \in \mathcal{F}_L$ and its extension $q \in Ext^+_{<L}(p)$, let us define the set of common occurrences $Occ_p(q) = Occ(p) \cap Occ(q)$ and its leftmost element $Pos_p(q) = \min Occ_p(q)$ (where $\min \emptyset = \infty$). For $p \in \mathcal{H}_L$ we define $\mathcal{D}_L(p)$ as the set of all Parikh vectors $q \in Ext^+_{<L}(p)$ which have a common occurrence with p:

$$\mathcal{D}_L(p) = \{q \in Ext^+_{<L}(p) \,:\, Occ_p(q) \neq \emptyset\}. \tag{8}$$

Finally, we define the sum over all *heavy* Parikh vectors $p \in \mathcal{H}_L$:

$$\mathcal{D}_L = \bigcup_{p \in \mathcal{H}_L} \mathcal{D}_L(p) \tag{9}$$

and for each $q \in \mathcal{D}_L$

$$Pos_{\mathcal{H}_L}(q) = \min_{p \in \mathcal{H}_L} Pos_p(q). \tag{10}$$

In other words, to determine $Pos_{\mathcal{H}_L}(q)$, we consider all $p \in \mathcal{H}_L$ such that $q \in \mathcal{D}_L(p)$ and take the minimum $i \in Occ_p(q)$.

Example 3.4 For the text $x = 2\,2\,2\,1\,1\,1\,2\,1\,2\,1\,1\,2\,2\,1\,1\,2\,2\,1\,2\,1$ from Example 3.3 ($L = 3$) we have $\mathcal{D}_L((2,1)) = \{(2,1), (3,1), (3,2), (2,2), (2,3)\}$ and the corresponding positions $Pos_{(2,1)}(q)$ are: $Pos_{(2,1)}((2,1)) = Pos_{(2,1)}((3,1)) = Pos_{(2,1)}((3,2)) = 3$, $Pos_{(2,1)}((2,2)) = 6$, and $Pos_{(2,1)}((2,3)) = 11$; see Fig. 3.

3.3 Data structure and queries

Our indexing data structure, denoted as $\text{INDEX}_L(x)$, consists of three parts:

(a) a dictionary with keys $p \in \mathcal{L}_L$ and values $Occ(p)$,
(b) a dictionary with keys $q \in \mathcal{D}_L$ and values $Pos_{\mathcal{H}_L}(q)$,
(c) the data structure of Lemma 2.5 to retrieve the Parikh vectors of factors.

The dictionaries are implemented using perfect hashing [14] to obtain $\mathcal{O}(1)$-time access and construction working in linear time with high probability.

Lemma 3.5 *The size of $INDEX_L(x)$ is $\mathcal{O}(n^2/L)$.*

Proof The size of part (a) of the index is $\sum_{p\in\mathcal{L}_L} |Occ(p)|$. By Fact 3.2, this is at most $\frac{n^2}{L} + n + 1 = \mathcal{O}(\frac{n^2}{L})$.

The size of part (b) is $|\mathcal{D}_L|$. To bound the size of this set we use the following claim.

Claim For every $p \in \mathcal{H}_L$, $|\mathcal{D}_L(p)| < |Occ(p)|$.

Proof Each Parikh vector in $\mathcal{D}_L(p)$ is an extension of the heavy L-factor p. More precisely: $\mathcal{D}_L(p) \subseteq Ext^+_{<L}(p)$. Therefore for $p \in \mathcal{H}_L$ we have:

$$|\mathcal{D}_L(p)| \leq |Ext^+_{<L}(p)| \leq L^\sigma < |Occ(p)|,$$

where the second inequality is due to Lemma 2.4(b). □

Immediately from the claim we obtain:

$$|\mathcal{D}_L| \leq \sum_{p\in\mathcal{H}_L} |\mathcal{D}_L(p)| \leq \sum_{p\in\mathcal{H}_L} |Occ(p)| \leq \tfrac{n^2}{L} + n + 1.$$

Note that in the last inequality we again used Fact 3.2.

Finally, the size of part (c) of the index is $\mathcal{O}(n)$ due to Lemma 2.5. Hence, the whole index uses $\mathcal{O}(n^2/L)$ space. □

The query is realized by the following algorithm $\textsc{Query}(q)$. It uses the precomputed position $Pos_{\mathcal{H}_L}(q)$ if q turns out to be generated by a heavy L-factor and applies Observation 3.1 to account for the possibility that the leftmost occurrence of q is generated by a light L-factor.

Lemma 3.6 *The query time in $INDEX_L(x)$ is $\mathcal{O}(L^{2\sigma-1})$.*

Proof The query algorithm works as presented in the pseudocode above. Let us bound its running time. By Lemma 2.4(a), $|Ext^-_{=r}(q)| = \mathcal{O}(r^{\sigma-1}) = \mathcal{O}(L^{\sigma-1})$. All elements $p \in Ext^-_{=r}(q)$ can be listed in $\mathcal{O}(L^{\sigma-1})$ time. For each $p \in \mathcal{L}_L$, by definition, $|Occ(p)| \leq L^\sigma$. Thus, with constant-time equivalence queries of Lemma 2.5, we obtain the desired $\mathcal{O}(L^{2\sigma-1})$ query time. □

```
Algorithm QUERY(q)
    first_pos := ∞;
    if q ∈ D_L then first_pos := Pos_{H_L}(q);
    r := |q| mod L;
    foreach p ∈ Ext⁻_{=r}(q) do
        if p ∈ L_L then
            foreach i ∈ Occ(p) do
                if P(x[i..i + |q| − 1]) = q then
                    first_pos := min(i, first_pos);
    if first_pos = ∞ then return "no occurrence";
    else return first_pos;
```

3.4 Simple construction of the index

The main part of the construction of $\mathrm{INDEX}_L(x)$ is the computation of the set $\mathcal{D}_L$ together with the leftmost occurrences $Pos_{\mathcal{H}_L}(q)$. A simple quadratic-time implementation is provided in the pseudocode below. We define $pref_L(v)$ as the longest prefix of a word v whose length is divisible by L.

Lemma 3.7 *$INDEX_L(x)$ can be constructed in $\mathcal{O}(n^2)$ time w.h.p.*

Proof As the first step we construct $\mathcal{F}_L$ together with $Occ(p)$ for each $p \in \mathcal{F}_L$. We store $\mathcal{F}_L$ as a dictionary (hash table with perfect hashing). This component can be constructed in $\mathcal{O}(\frac{n^2}{L})$ time using Lemma 2.5. Moreover, it allows to compute $\mathcal{L}_L$ and $\mathcal{H}_L$ in the same time.

```
Algorithm COMPUTE-D_L(x, n)
    D_L := ∅;
    compute H_L;
    for i := n downto 1 do
        for j := i to n do
            ▷ process factors of x starting with rightmost occurrences
            q := P(x[i..j]);
            p := P(pref_L(x[i..j]));
            if p ∈ H_L then
                insert q into D_L;
                Pos_{H_L}(q) := i;
```

We construct $\mathcal{D}_L$ using the algorithm from the pseudocode. With the aid of Lemma 2.5 it runs in $\mathcal{O}(n^2)$ time. Randomization of the construction is due to perfect hashing used to implement the dictionaries in the index. □

3.5 Complexity of the index

The following theorem summarizes the results of the previous subsection.

Theorem 3.8 *For any text of length n and any integer $1 \leq L \leq n$ there exists an index for jumbled pattern matching of size $\mathcal{O}(\frac{n^2}{L})$ and with $\mathcal{O}(L^{2\sigma-1})$ query time. The index can be constructed in $\mathcal{O}(n^2)$ time w.h.p.*

For some particular values of the trade-off parameter L we obtain particularly useful indexes.

Corollary 3.9 *For any text of length n one can construct in $\mathcal{O}(n^2)$ time an index for jumbled pattern matching of size $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ which answers queries in $\mathcal{O}\big(\big(\frac{\log n}{(\log\log n)^2}\big)^{2\sigma-1}\big)$ time.*

Proof We take $L = \lceil \frac{\log n}{(\log\log n)^2} \rceil$ and apply Theorem 3.8. □

Corollary 3.10 *For any text of length n and any $0 < \delta < 1$ one can construct in $\mathcal{O}(n^2)$ time an index for jumbled pattern matching of size $\mathcal{O}(n^{2-\delta})$ which answers queries in $\mathcal{O}(n^{(2\sigma-1)\delta})$ time.*

Proof We take $L = \lceil n^\delta \rceil$ and apply Theorem 3.8. □

In Section 6 we show that the data structure from Theorem 3.8 can be constructed in $\mathcal{O}\big(\max(\frac{n^2}{L}, \frac{n^2(\log\log n)^2}{\log n})\big)$ time. This yields $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$-time construction in the special cases of both corollaries. However, first we improve the query time.

4 Faster queries for small patterns

While $\mathcal{O}(n^{(2\sigma-1)\delta})$ is sublinear in n for small δ, it is still rather large, and, especially for very small patterns, might be considered unsatisfactory. We modify the data structure to handle such patterns much more efficiently, in $\mathcal{O}(m^{(2\sigma-1)\delta})$ time for patterns of norm m. We start with an auxiliary data structure.

Lemma 4.1 *For any text of length n and any integers L, k such that $1 \leq L \leq k \leq n$, there exists an index for jumbled pattern matching of size $\mathcal{O}(\frac{nk}{L})$ and with $\mathcal{O}(L^{2\sigma-1})$ query time for patterns of norm at most k. The index can be constructed in $\mathcal{O}(nk)$ time w.h.p.*

Proof We repeat the proof of Theorem 3.8 but we restrict to L-factors of norm at most k. Let $\mathcal{F}_{L,k}$ denote the set of these factors. Similarly as in the case of $\mathcal{F}_L$, we obtain $|\mathcal{F}_{L,k}| \leq \frac{n \cdot k}{L}$. The rest of the construction is the same as before. □

Theorem 4.2 *For any text of length n and any $0 < \delta < 1$ there exists an index for jumbled pattern matching of size $\mathcal{O}(n^{2-\delta})$ and with $\mathcal{O}(m^{(2\sigma-1)\delta})$ query time, where m is the norm of the pattern. The index can be constructed in $\mathcal{O}(n^2)$ time w.h.p.*

Proof Let

$$K \;=\; \{2^i \;:\; 0 \leq i \leq \lfloor \log n \rfloor\} \cup \{n\}.$$

We build the data structures from Lemma 4.1 for each $k \in K$ with $L_k = \lfloor k^\delta \rfloor$. The size of a single data structure is $\mathcal{O}(\frac{n \cdot k}{L_k})$, which is $\mathcal{O}(n \cdot k^{1-\delta})$.

The total size is of order:

$$\begin{aligned} n^{2-\delta} + \sum_{i=0}^{\lfloor \log n \rfloor} n \cdot 2^{i(1-\delta)} \;&=\; n^{2-\delta} + n \sum_{i=0}^{\lfloor \log n \rfloor} (2^{1-\delta})^i \\ &=\; n^{2-\delta} + n \frac{(2^{1-\delta})^{\lfloor \log n \rfloor + 1} - 1}{2^{1-\delta} - 1} \\ &=\; n^{2-\delta} + n \cdot \mathcal{O}(2^{(1-\delta)\log n}) = \mathcal{O}(n^{2-\delta}). \end{aligned}$$

To answer a query about a pattern p of size m we take

$$k \;=\; \min\{j \in K \;:\; j \geq m\}.$$

Then we apply the query algorithm from Lemma 4.1 (using only the part of the data structure relevant to k). The query works in $\mathcal{O}(m^{(2\sigma-1)\delta})$ time, since $k \leq 2m$ and $(2\sigma - 1)\delta$ is a constant. The total construction time sums up to $\mathcal{O}(n^2)$. □

A similar argument gives an improvement of Corollary 3.9.

Theorem 4.3 *For any text of size n there exists an index for jumbled pattern matching of $\mathcal{O}\big(\frac{n^2 (\log\log n)^2}{\log n}\big)$ size and with $\mathcal{O}\big(\big(\frac{\log m}{(\log\log m)^2}\big)^{2\sigma-1}\big)$ query time, where m is the norm of the pattern. The index can be constructed in $\mathcal{O}(n^2)$ time w.h.p.*

Proof We proceed as in the proof of Theorem 4.2. We take

$$K \;=\; \{2^{2^i} \;:\; 0 \leq i \leq \lfloor \log\log n \rfloor\} \cup \{n\}.$$

Define $f(x) = \frac{\log x}{(\log\log x)^2}$. We combine the instances of the data structure for Lemma 4.1 with $L_k = \lceil f(k) \rceil$ for each $k \in K$.

To answer a query for a pattern of norm m, we pick $k = \min\{j \in K \;:\; j \geq m\}$. Then the query takes $\mathcal{O}(f(k)^{2\sigma-1})$ time. Note that

$$f(k) \leq f(m^2) = \tfrac{\log m^2}{(\log\log m^2)^2} = \tfrac{2\log m}{(\log\log m^2)^2} \leq \tfrac{2\log m}{(\log\log m)^2} = 2f(m),$$

so the query time is indeed $\mathcal{O}(f(m)^{2\sigma-1})$. Moreover, we have

$$\sum_{k\in K\setminus\{n\}} \frac{nk}{f(k)} = \sum_{i=0}^{\lfloor \log\log n\rfloor} \frac{n2^{2^i}}{f(2^{2^i})} = n \sum_{i=0}^{\lfloor \log\log n\rfloor} \frac{i^2 2^{2^i}}{2^i}.$$

Note that the i-th summand in the above sum is at least two times greater than the $(i-1)$-th one, so the whole expression is bounded from above by

$$2n \cdot \frac{(\log\log n)^2 2^{2^{\log\log n}}}{2^{\log\log n}} = \mathcal{O}(\tfrac{n^2}{f(n)}).$$

The index size and the construction time are as stated. □

In Section 6 we show that the data structures of Theorems 4.2 and 4.3 can actually be constructed in $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ time.

5 Efficient element location in packed lists

In this section we consider an auxiliary problem of finding the first occurrence for each distinct value occurring in a given list. We focus on lists of length greater than the size of the universe of values and we aim at sublinear time in the length of the lists, which requires a suitable compact representation of these lists. The algorithm developed in this section is later used to obtain a subquadratic-time construction of our index for jumbled pattern matching.

Let w be a lower bound on the machine word size of the word-RAM machine and let $U = \{0, \ldots, N-1\}$ for $N \leq 2^w$ be the *universe*. Each element of the universe can be stored in binary using $B = \lceil \log N \rceil$ bits, and therefore a single machine word can fit up to $M = \lfloor \frac{w}{B} \rfloor$ elements one after the other. Such a sequence of up to M elements stored in a single word is called a *short list*, often denoted as ℓ in this section. If the universe U is binary, i.e., if $U = \{0, 1\}$, we refer to short lists as *short bitmasks*.

Note that the short list does *not* store its length and thus in general we cannot, for example, tell the difference between a given list of length $m < M$ and one of the valid lists of length M. Consequently, we need to know the length m to interpret the encoding. The following fact describes how short lists can be used to store lists of larger length m in roughly $\frac{m}{M}$ machine words. The resulting data structure is called here a *packed list*.

Fact 5.1 (Packed List) *One can store a list* $\mathbf{L}$ *that consists of* m *elements of an* N*-element universe* U *using* $\mathcal{O}\big(1 + \frac{m \log N}{w}\big)$ *machine words, so that the following operations can be performed in constant time:*

Push *append a given short list at the end of* $\mathbf{L}$*,*
Pop *return a short list of* m'*,* $m' \leq \min(m, M)$*, first elements of* $\mathbf{L}$ *and remove those elements from* $\mathbf{L}$
Length *return the length* m *of* $\mathbf{L}$*.*

Proof First, observe that bit-wise operations let us implement two basic operations on short lists in constant time:

- concatenate two short lists of lengths m and m' (with $m + m' \leq M$),
- retrieve a short list of m' heading or trailing elements of a given short list of length m ($m' \leq m \leq M$).

Our implementation of a packed list consists of a queue of short lists. We also store the length m of the whole list $\mathbf{L}$, as well the as the lengths m_b and m_e of the short lists ℓ_b and ℓ_e at the beginning and at the end of the queue, respectively. We maintain as an invariant that the remaining short lists in the queue have exactly M elements each; see Fig. 4. Consequently, the size of the queue is $\frac{m}{M} + \mathcal{O}(1)$ and thus the space consumption of the packed list is $\mathcal{O}(1 + \frac{m \log N}{w})$ as desired.

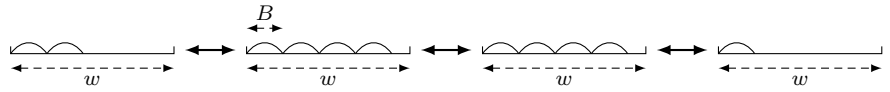

Fig. 4 A schematic view of an example packed list with $m = 11$ elements. Here $M = 4$, $m_b = 2$ and $m_e = 1$.

The implementation of the **Length** query is trivial and so is updating the length m under **Push** and **Pop** operations.

To append a short list ℓ of m' elements, we extract the first $\min(M - m_e, m')$ elements of ℓ and concatenate the resulting short list with the tail ℓ_e of the queue. If $m' > M - m_e$, we also extract the remaining elements of ℓ and append the resulting short list as the new tail of the queue. The length m_e needs to be updated accordingly (so needs m_b if the queue had length at most one).

The implementation of the operation *Pop* is similar. First, we extract the first $\min(m', m_b)$ elements from the head ℓ_b and then replace ℓ_b with its remaining elements. If $m' < m_b$, we are done and we return the extracted short list. Otherwise, ℓ_b becomes empty, so we drop it from the queue and remove $m' - m_b$ first elements from the new head. The resulting list is formed by concatenation of the two extracted short lists. □

For a packed list $\mathbf{L}$ we define a bitmask $\mathit{FirstOcc}_{\mathbf{L}}$ of *first occurrences*. Its length is the same as the length of $\mathbf{L}$ and the i-th bit of $\mathit{FirstOcc}_{\mathbf{L}}$ is set if the value at the i-th position in $\mathbf{L}$ has no earlier occurrence in $\mathbf{L}$. The bitmask is stored as a packed list over the universe $U = \{0, 1\}$.

Example 5.2 If the packed list $\mathbf{L}$ over $U = \{0, 1, 2, 3\}$ contains the following elements:

$$1, 0, 0, 3, 0, 1, 1, 1, 3, 2, 3, 2$$

then the corresponding bitmask $\mathit{FirstOcc}_{\mathbf{L}}$ is

$$1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0$$

The following lemma is the main result of this section.

Lemma 5.3 *After $\mathcal{O}(2^{2w}w)$-time preprocessing, for any packed list $\mathbf{L}$, the bitmask $FirstOcc_{\mathbf{L}}$ can be computed in $\mathcal{O}\big(1+\frac{m(\log N)^2}{w}\big)$ time, where $m = Length(\mathbf{L})$.*

Proof We apply a divide-and-conquer algorithm to solve the problem. We partition $\mathbf{L}$ into sublists of elements not exceeding p ($\mathbf{L}_{\leq p}$) and larger than p ($\mathbf{L}_{>p}$) for some *pivot* value p, recurse on $\mathbf{L}_{\leq p}$ and $\mathbf{L}_{>p}$, and retrieve $FirstOcc_{\mathbf{L}}$ from $FirstOcc_{\mathbf{L}_{\leq p}}$ and $FirstOcc_{\mathbf{L}_{>p}}$. This approach is similar to efficient wavelet tree construction for sequences over a small universe; see [3, 22].

For a short list ℓ and an integer $p \in U$ we define an operation of partitioning ℓ with p as a pivot. We denote $Partition(\ell, p) = (\ell_{\leq p}, \ell_{>p}, b_{\ell,p})$, where $\ell_{\leq p}$ and $\ell_{>p}$ are both short lists, $\ell_{\leq p}$ is a sublist of ℓ consisting of elements not exceeding p, while $\ell_{>p}$ is the complement sublist of ℓ, and $b_{\ell,p}$ is a short bitmask of the same length as ℓ in which zeroes correspond to elements of ℓ not exceeding p and ones to the remaining elements. For the $Partition(\ell, p)$ queries we additionally assume that length of ℓ is given in the input, and lengths of $\ell_{\leq p}$ and $\ell_{>p}$ are returned as a part of the output. The answer to a single query can be naively computed in linear time with respect to the length of the list, i.e., in $\mathcal{O}(M) = \mathcal{O}(w)$ time. Since the number of distinct pairs (ℓ, p) does not exceed $\mathcal{O}(2^w \cdot 2^w) = \mathcal{O}(2^{2w})$, the answers to all $Partition(\ell, p)$ queries can be precomputed in $\mathcal{O}(2^{2w}w)$ time.

Our next goal is to implement LargePartition, that is, the extension of *Partition* to packed lists of arbitrary length. We scan the input list $\mathbf{L}$ and at each step we *pop* a short list consisting of the first $M = \lfloor \frac{w}{B} \rfloor$ elements of $\mathbf{L}$, partition it with respect to p, and append (*push*) two resulting short lists to the output packed lists $\mathbf{L}_{\leq p}$ and $\mathbf{L}_{>p}$. Additionally, we concatenate the bitmasks obtained from *Partition* operations to obtain the resulting bitmask for LargePartition. See the pseudocode below for details.

Observation 5.4 LargePartition$(\mathbf{L}, p)$ *works in $\mathcal{O}\big(1+\frac{|\mathbf{L}|\log N}{w}\big)$ time.*

```
Algorithm LargePartition(L, p)
    Initialize L_{≤p} and L_{>p} as empty packed lists;
    Initialize B as an empty bitmask;
    while Length(L) > 0 do
        ℓ := Pop(L, min(M, Length(L)));
        (ℓ_{≤p}, ℓ_{>p}, b) := Partition(ℓ, p);
        Push(L_{≤p}, ℓ_{≤p});
        Push(L_{>p}, ℓ_{>p});
        Push(B, b);
    return (L_{≤p}, L_{>p}, B);
```

Algorithm LargeMerge($\mathbf{B}, \mathbf{F}_0, \mathbf{F}_1$)
Initialize F as an empty bitmask;
while *Length*($\mathbf{B}$) > 0 **do**
$m := \min(w, \mathit{Length}(\mathbf{B}))$; ▷ number of bits
$b := \mathit{Pop}(\mathbf{B}, m)$; ▷ a chunk of $\mathbf{B}$
$m_1 := \mathit{PopCount}(b)$; ▷ number of ones in b
$f_0 := \mathit{Pop}(\mathbf{F}_0, m - m_1)$; ▷ first m_0 elements of $\mathbf{F}_0$
$f_1 := \mathit{Pop}(\mathbf{F}_1, m_1)$; ▷ first m_1 elements of $\mathbf{F}_1$
$f := \mathit{Merge}(b, f_0, f_1)$; ▷ small merge
$\mathit{Push}(\mathbf{F}, f)$; ▷ appending the result
return $\mathbf{F}$;

Next, we show how to retrieve the desired $\mathit{FirstOcc}_{\mathbf{L}}$ from $\mathit{FirstOcc}_{\mathbf{L}_{\le p}}$ and $\mathit{FirstOcc}_{\mathbf{L}_{>p}}$. Conceptually, $\mathit{FirstOcc}_{\mathbf{L}}$ can be easily obtained using the auxiliary bitmask $\mathbf{B}$: it suffices to transform $\mathbf{B}$ so that the i-th 0 in $\mathbf{B}$ is replaced by the i-th bit from $\mathit{FirstOcc}_{\mathbf{L}_{\le p}}$, while the i-th 1 in $\mathbf{B}$ is replaced by the i-th bit from $\mathit{FirstOcc}_{\mathbf{L}_{>p}}$.

For an efficient implementation we use the following auxiliary operation $\mathit{Merge}(b, f_0, f_1)$: given a short bitmask b (of length $m \le w$), with m_1 set bits, and a pair of bitmasks: f_0 of length $m - m_1$ and f_1 of length m_1, replace zeroes in b with consecutive bits of f_0, and ones in b with consecutive bits of f_1. Note that there are $\mathcal{O}(2^{2w})$ possible inputs, so the answers can be precomputed in $\mathcal{O}(2^{2w}w)$ time. We also precompute for each short bitmask b the number of set bits (denoted as $\mathit{PopCount}(b)$).

In order to extend *Merge* to bitmasks of arbitrary length, i.e., to compute LargeMerge($\mathbf{B}, \mathbf{F}_0, \mathbf{F}_1$), we scan $\mathbf{B}$ in chunks of at most w bits, obtained using *pop* operation. For such a chunk b we compute the numbers of zeroes m_0 and ones m_1. We pop m_i bits from each $\mathbf{F}_i$ to obtain f_i, compute $f = \mathit{Merge}(b, f_0, f_1)$ and append it at the end of the output.

Observation 5.5 LargeMerge($\mathbf{L}, p$) *works in* $\mathcal{O}\big(1 + \frac{|\mathbf{B}|}{w}\big)$ *time.*

Finally, we note that it is straightforward to precompute in $\mathcal{O}(2^w w^2) = o(2^{2w}w)$ time all answers to the $\mathit{SmallFirstOcc}(\ell)$ queries asking for $\mathit{FirstOcc}_{\ell}$ for short lists ℓ. Combined with the LargePartition and LargeMerge developed above, this lets us design a recursive procedure computing $\mathit{FirstOcc}_{\mathbf{L}}$ for arbitrary packed lists. We extend the input with a range $\{r_b, \ldots, r_e\} \subseteq U$ guaranteed to contain all members of $\mathbf{L}$; in the initial call we have $r_b = 0$, $r_e = N - 1$.

We conclude the proof with the analysis of the running time. The preprocessing time is $\mathcal{O}(2^{2w}w)$ as required. If the initial list is short, the procedure clearly runs in $\mathcal{O}(1)$ time. In the discussion below we ignore this special case and assume the initial length m is greater than M.

Algorithm LargeFirstOcc($\mathbf{L}, r_b, r_e$)
  if *Length*($\mathbf{L}$) $\leq M$ **then**
    return *SmallFirstOcc*($\mathbf{L}$);
  if $r_b = r_e$ **then**
    return a bitmask with 1 one followed by *Length*($\mathbf{L}$) $- 1$ zeroes;
  $p := \lfloor \frac{r_b + r_e}{2} \rfloor$;
  $(\mathbf{L}_{\leq p}, \mathbf{L}_{>p}, \mathbf{B}) :=$ LargePartition($\mathbf{L}, p$);
  $\mathbf{F}_0 :=$ LargeFirstOcc($\mathbf{L}_{\leq p}, r_b, p$);
  $\mathbf{F}_1 :=$ LargeFirstOcc($\mathbf{L}_{>p}, p+1, r_e$);
  return LargeMerge($\mathbf{B}, \mathbf{F}_0, \mathbf{F}_1$);

A single call to the LargeFirstOcc procedure, excluding the recursive calls it makes, takes $\mathcal{O}\big(1 + \frac{|\mathbf{L}| \log N}{w}\big)$ time. The $\mathcal{O}(1)$ term may dominate only in the leaves of the recursion tree. Therefore, unless the root of the tree is a leaf, we account the $\mathcal{O}(1)$ time in the amortized running time of the parent call. The procedure makes at most two recursive calls, so the amortized running time becomes $\mathcal{O}\big(\frac{|\mathbf{L}| \log N}{w}\big)$. The depth of the recursion is bounded by $\lceil \log N \rceil$ since the interval $\{r_b, \ldots, r_e\}$ is halved at each step. Moreover, the lists in a single level of recursion tree are of total length m. This gives $\mathcal{O}\big(\frac{m \log N}{w}\big)$ amortized time per level and $\mathcal{O}\big(\frac{m(\log N)^2}{w}\big)$ time in total. Accounting for the $\mathcal{O}(1)$-time in the border case, we get the announced $\mathcal{O}\big(1 + \frac{m(\log N)^2}{w}\big)$ time complexity. □

Corollary 5.6 *After $\mathcal{O}(2^{2w}w)$-time preprocessing, given a packed list $\mathbf{L}$ of length m we can compute, for each value $j \in U$, the position of the first occurrence of j in $\mathbf{L}$ (or ∞ if no such position exists) in $\mathcal{O}\big(N + \frac{m(\log N)^2}{w}\big)$ time.*

Proof We apply Lemma 5.3 to obtain $\mathbf{F} = \mathit{FirstOcc}_{\mathbf{L}}$ in $\mathcal{O}\big(1 + \frac{m(\log N)^2}{w}\big)$ time. Next, we initialize the output array in $\mathcal{O}(N)$ time to ∞'s, and iterate through the set bits of $\mathbf{F}$ in $\mathcal{O}\big(N + \frac{m}{w}\big)$ time. Whenever we encounter a set bit at position j, we retrieve $i = \mathbf{L}[j]$ and set the i-th position of the output array to j. □

6 Reducing preprocessing time

In Section 3 we presented an index of subquadratic size allowing for sublinear-time queries. However, the construction time was quadratic. Here, we slightly improve this parameter.

Recall that the only bottleneck of the simple construction algorithm, developed in Section 3.4, is computing the set $\mathcal{D}_L$ (of Abelian factors generated

by heavy L-factors) and the witness positions $Pos_{\mathcal{H}_L}(q)$. Our improved solutions process each $p \in \mathcal{H}_L$ separately, determining $\mathcal{D}_L(p)$ and $Pos_p(q)$ for all $q \in \mathcal{D}_L(p)$.

First, in Theorem 6.1, we actually deal with small values of L using bit-parallelism of the word-RAM model. Our approach is as follows: We assign each $q \in Ext^+_{<L}(p)$ a short integer identifier of $\mathcal{O}(\sigma \log L)$ bits and for each $i \in Occ(p)$ we compute a short list representing those extensions $q \in Ext^+_{<L}(p)$ for which $i \in Occ_p(q)$. Then, we concatenate these short lists into a single packed list, apply Corollary 5.6, and translate the first occurrences of each identifier in the packed list into the leftmost occurrences $Pos_p(q)$ of each $q \in Ext^+_{<L}(p)$. Provided that $L = \mathcal{O}(\frac{\log n}{(\log\log n)^2})$, this gives a factor-$L$ speed-up.

For larger L, in Theorem 6.3, we use a two-level procedure, which introduces an auxiliary parameter $\ell \approx \frac{\log n}{(\log\log n)^2}$ and generates $\mathcal{D}_L(p)$ in $\frac{L}{\ell}$ phases, using the same techniques as before to obtain a factor-ℓ speedup.

Theorem 6.1 *The index of Theorem 3.8 admits a construction algorithm running in $\mathcal{O}(\frac{n^2}{L} + \frac{n^2(\log L)^2}{\log n})$ time w.h.p.*

Proof First, observe that if $\log L \geq \sqrt{\log n}$ the claimed construction time is quadratic since $\frac{n^2(\log L)^2}{\log n} = \Omega(n^2)$. Thus, in the following we assume $\log L < \sqrt{\log n}$, which in particular implies $\log L = o(\log n)$.

We apply the results of Section 5 with $w = \alpha \log n$ for some constant $\alpha < \frac{1}{2}$, so that the preprocessing time is $\mathcal{O}(2^{2w}w) = \mathcal{O}(n^{2\alpha}\log n) = o(n)$.

As described in the proof of Lemma 3.7, all parts of the preprocessing excluding the computation of $\mathcal{D}_L$ work in $\mathcal{O}(\frac{n^2}{L})$ time. The missing component is constructed using Corollary 5.6.

Let us consider $Ext^+_{<L}(\bar{0}_\sigma)$, the set of Parikh vectors e satisfying $|e| < L$. These Parikh vectors can be interpreted as lists of length σ over $\{0,\dots,L-1\}$, and thus they can be represented as integers within $U = \{0,\dots,N-1\}$ for $N = L^\sigma$, which shall be the universe for the packed lists. Here, entries of the Parikh vector correspond to digits in the base-L representation of its identifier.

Note that these integer identifiers fit into a machine word since $L^\sigma = 2^{\sigma\log L} = 2^{o(\log n)}$ while $w = \Theta(\log n)$. For each position i, $1 \leq i \leq n$, we construct a packed list $\mathbf{L}_i$ of length $\min(L, n+2-i)$ whose j-th element (0-based) is the identifier of $\mathcal{P}(x[i..i+j-1])$. A single list can be constructed in $\mathcal{O}(L)$ time, which gives $\mathcal{O}(nL)$ time in total. Each $\mathbf{L}_i$ is stored in $\mathcal{O}(1+\frac{L\log N}{w})$ space.

Now sets $\mathcal{D}_L(p)$ are computed separately for each $p \in \mathcal{H}_L$. Note that along with any $q \in \mathcal{D}_L(p)$ we need to find the leftmost occurrence $Pos_p(q)$. Observe that for any $i \in Occ_p(q)$ we have $\mathcal{P}(x[i+|p|..i+|p|+r-1]) = q - p$ where $r = |q| - |p| \in \{0,\dots,L-1\}$.

We consider all positions $i \in Occ(p)$, and for each such position i we take the list $\mathbf{L}_{i+|p|}$. To compute $\mathcal{D}_L(p)$, it suffices to find all the distinct elements in these lists $\mathbf{L}_{i+|p|}$ and add p to each of the corresponding Parikh vectors. For this, we concatenate the corresponding lists $\mathbf{L}_{i+|p|}$ into a single packed list $\mathbf{L}$

of length not exceeding $|Occ(p)| \cdot L$, and run the algorithm of Corollary 5.6. For each identifier occurring in $\mathbf{L}$, we retrieve its first position in $\mathbf{L}$ which can be translated into the position $Pos_p(q)$ of the corresponding occurrence of $q \in \mathcal{D}_L(p)$. The latter is easy if we store $Occ(p)$ as a sorted array and concatenate $\mathbf{L}_{i+|p|}$ in this order.

Consequently, for fixed p the set $\mathcal{D}_L(p)$ together with the witness occurrences $Pos_p(q)$ can be computed in

$$\mathcal{O}\Big(N + |Occ(p)| \cdot \Big(1 + \tfrac{L(\log N)^2}{w}\Big)\Big) = \mathcal{O}\Big(L^\sigma + |Occ(p)| \cdot \Big(1 + \tfrac{L(\log L)^2}{\log n}\Big)\Big)$$

time. By the definition of a heavy L-factor, $|Occ(p)| > L^\sigma$, so the above running time can be bounded by $\mathcal{O}\big(|Occ(p)| \cdot \big(1 + \frac{L(\log L)^2}{\log n}\big)\big)$. By Fact 3.2, across all $p \in \mathcal{H}_L$ this sums up to $\mathcal{O}\big(\frac{n^2}{L} + \frac{n^2(\log L)^2}{\log n}\big)$. □

For small L we obtain a corollary which basically states that we have an optimal construction time of our data structure, since its running time matches the space complexity of the index.

Corollary 6.2 *If $L = \mathcal{O}\left(\frac{\log n}{(\log\log n)^2}\right)$, then the index of Theorem 3.8 can be constructed in $\mathcal{O}(\frac{n^2}{L})$ time w.h.p. In particular, the index of Corollary 3.9 can be constructed in $\mathcal{O}(\frac{n^2(\log\log n)^2}{\log n})$ time w.h.p.*

Next, we generalize the algorithm to extend the scope of its usefulness for larger L.

Theorem 6.3 *If $L = \Omega\big(\frac{\log n}{(\log\log n)^2}\big)$, then the index of Theorem 3.8 can be constructed in $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ time w.h.p. In particular, the index of Corollary 3.10 can be constructed in $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ time w.h.p.*

Proof Let $\ell = \lceil \frac{\log n}{(\log\log n)^2} \rceil$. If $L < \ell^{\sigma+1}$, then already Theorem 6.1 gives the desired complexity bound. Thus, in the following we assume that $L \geq \ell^{\sigma+1}$.

Again, we shall concentrate on computing sets $\mathcal{D}_L(p)$ for each $p \in \mathcal{H}_L$ and the witness occurrences $Pos_p(q)$ for each $q \in \mathcal{D}_L(p)$. As in the proof of Theorem 6.1, we assign integer identifiers to short Abelian factors, and precompute lists $\mathbf{L}_i$ for each position i. This time, however, we perform these operations for ℓ instead of L. Consequently, we set $N = \ell^\sigma$ and the lists $\mathbf{L}_i$ take $\mathcal{O}(n\ell)$ time to construct.

We construct $\mathcal{D}_L(p)$ in $\mathcal{O}(\frac{L}{\ell})$ phases. During the k-th phase, we consider extensions of p by a length within $\{(k-1)\ell+1, \ldots, k\ell\}$. We begin each phase with constructing for all $q \in Ext^+_{=(k-1)\ell}(p)$ lists $Occ_p(q)$. For this, we scan $Occ(p)$ and for each position i we append i to the appropriate list $Occ_p(q)$ (for $q = \mathcal{P}(x[i..i+|q|-1])$). Using Lemma 2.5 and a dictionary (hash table) to map Parikh vectors into lists, this can be achieved in $\mathcal{O}(|Ext^+_{=(k-1)\ell}(p)| + |Occ(p)|)$ time, which reduces to $\mathcal{O}(|Occ(p)|)$ since

$$|Ext^+_{=(k-1)\ell}(p)| = \mathcal{O}\left(((k-1)\ell)^{\sigma-1}\right) = \mathcal{O}\left(L^{\sigma-1}\right)$$

by Lemma 2.4(a), and $p \in \mathcal{H}_L$. Next, we process each occurrence list $Occ_p(q)$.

For each such list of length at most ℓ^σ we naively scan the occurrences and the respective extensions. This takes $|Occ_p(q)|\ell = \mathcal{O}(\ell^{\sigma+1})$ time per list, which is at most $\mathcal{O}(|Ext^+_{=(k-1)\ell}(p)|\ell^{\sigma+1}) = \mathcal{O}(L^{\sigma-1}\ell^{\sigma+1})$ for the whole phase.

The remaining lists $Occ_p(q)$ are processed similarly as in the proof of Theorem 6.1: we concatenate $\mathbf{L}_{i+|q|}$ for $i \in Occ_p(q)$ to obtain $\mathbf{L}$, retrieve the first occurrences of elements in $\mathbf{L}$ using Corollary 5.6, add q to the Parikh vectors represented by the identifiers obtained, and translate positions in $\mathbf{L}$ into positions in the text. For a single list $Occ_p(q)$ this takes

$$\mathcal{O}\Big(N + |Occ_p(q)| \cdot \Big(1 + \tfrac{\ell(\log N)^2}{w}\Big)\Big) = \mathcal{O}\Big(\ell^\sigma + |Occ_p(q)| \cdot \Big(1 + \tfrac{\ell(\log \ell)^2}{w}\Big)\Big)$$

time, which by the lower bound on $|Occ_p(q)|$ is of order

$$\mathcal{O}\Big(|Occ_p(q)| \cdot \Big(1 + \tfrac{\ell(\log \ell)^2}{w}\Big)\Big).$$

Each $i \in Occ(p)$ occurs in at most one $Occ_p(q)$, so for the whole k-th phase, this is

$$\mathcal{O}\Big(|Occ(p)| \cdot \Big(1 + \tfrac{\ell(\log \ell)^2}{w}\Big)\Big).$$

Combining both parts and summing up over phases, we conclude that a single $\mathcal{D}_L(p)$ can be computed in time

$$\mathcal{O}\Big(L^\sigma \ell^\sigma + |Occ(p)| \cdot \Big(\tfrac{L}{\ell} + \tfrac{L(\log \ell)^2}{w}\Big)\Big).$$

By definition of a heavy L-factor, we can bound L^σ by $|Occ(p)|$, and, by the assumed lower bound on L, we can bound $\ell^\sigma \le \frac{L}{\ell}$. Consequently, the time required to compute $\mathcal{D}_L(p)$ reduces to

$$\mathcal{O}\Big(|Occ(p)| \cdot L \cdot \Big(\tfrac{1}{\ell} + \tfrac{(\log \ell)^2}{w}\Big)\Big) = \mathcal{O}\Big(|Occ(p)| \cdot L \cdot \tfrac{(\log\log n)^2}{\log n}\Big).$$

By Fact 3.2, across all $p \in \mathcal{H}_L$ this sums up to the desired upper bound of $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$. □

Finally, we observe that the construction of the index of Lemma 4.1 can also be improved. We only need to make sure that the $\mathcal{O}(nL)$ term (hidden in the construction of Theorem 6.1) or the $\mathcal{O}(n\ell)$ term (in the proof of Theorem 6.3) do not dominate the running time. For this, it suffices to allow for additional $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ term in the running time, since we use the approach of Theorem 6.1 only for $L < \log^{\sigma+1} n$, and in Theorem 6.3 we have $l < \log^{\sigma+1} n$.

Lemma 6.4 *The index of Lemma 4.1 can be constructed in* $\mathcal{O}\big(n \log^{\sigma+1} n + \max\big(\frac{nk}{L}, \frac{nk(\log\log n)^2}{\log n}\big)\big)$ *time w.h.p.*

Corollary 6.5 *The indexes of Theorems 4.2 and 4.3 can be constructed in* $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ *time w.h.p.*

Proof In the proofs of Theorems 4.2 and 4.3, we use only $\mathcal{O}(\log n)$ instances of the data structure of Lemma 4.1, so the additional $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ term is dominated by the claimed construction time. The $\mathcal{O}(\frac{nk}{L})$ term sums up to the data structure size, which is also dominated. Finally, it suffices to note that $\sum_{k \in K} k = \mathcal{O}(n)$, so the $\mathcal{O}\big(\frac{nk(\log\log n)^2}{\log n}\big)$ terms sum up to $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$, as desired. □

7 Conclusions

We presented several versions of an index for jumbled pattern matching in a text over a constant-sized alphabet. The index admits a size vs query time trade-off, which in particular gives a data structure of size $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ with $\mathcal{O}\big(\big(\frac{\log m}{(\log\log m)^2}\big)^{2\sigma-1}\big)$ query time, and a solution of size $\mathcal{O}(n^{2-\delta})$ with $\mathcal{O}(m^{(2\sigma-1)\delta})$ query time for any $0 < \delta < 1$. Thus the index is able to provide polylogarithmic query time and subquadratic space, or strongly sublinear query time along with strongly subquadratic space. Both versions of the index can be constructed in $\mathcal{O}\big(\frac{n^2(\log\log n)^2}{\log n}\big)$ time with high probability under the word-RAM model. Moreover, the query algorithm computes the leftmost occurrence of the query pattern if it exists.

Recall that for a constant alphabet of size $\sigma \geq 3$, in [2] it is shown that, under strong 3SUM-hardness assumption, jumbled indexing requires $\Omega(n^{2-\epsilon_\sigma})$ preprocessing time or $\Omega(n^{1-\delta_\sigma})$ query time, where $\epsilon_\sigma, \delta_\sigma < 1$ are computable constants. This leaves room for improvement of the construction time of an index, and also does not apply to the space vs query time trade-off of an index.

Acknowledgements The authors would like to thank several researchers present at the Stringmasters 2013 workshop in Verona for introducing the problem and comments on the preliminary solution: Péter Burcsi, Ferdinando Cicalese, Gabriele Fici, Travis Gagie, Arnaud Lefebvre and Zsuzsanna Lipták. We are especially grateful to Ferdinando Cicalese and Travis Gagie for very valuable remarks.

The authors thank anonymous reviewers for numerous comments that helped significantly improve the presentation of the article.

Tomasz Kociumaka is supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program, grant no 0179/DIA/2013/42. Jakub Radoszewski is supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' program in 2015-2016, grant no 0392/IP3/2015/73. He also receives financial support of Foundation for Polish Science. Wojciech Rytter is supported by the Polish National Science Center, grant no 2014/13/B/ST6/00770.

References

1. Amir, A., Butman, A., Porat, E.: On the relationship between histogram indexing and block-mass indexing. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences **372**(2016) (2014). ID 20130132.
2. Amir, A., Chan, T.M., Lewenstein, M., Lewenstein, N.: On hardness of jumbled indexing. In: J. Esparza, P. Fraigniaud, T. Husfeldt, E. Koutsoupias (eds.) Automata,

Languages, and Programming, ICALP 2014, Part I, *LNCS*, vol. 8572, pp. 114–125. Springer (2014)
3. Babenko, M., Gawrychowski, P., Kociumaka, T., Starikovskaya, T.: Wavelet trees meet suffix trees. In: P. Indyk (ed.) 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, pp. 572–591. SIAM (2015)
4. Björklund, A., Kaski, P., Kowalik, L.: Constrained multilinear detection and generalized graph motifs. Algorithmica **74**(2), 947–967 (2016)
5. Bremner, D., Chan, T.M., Demaine, E.D., Erickson, J., Hurtado, F., Iacono, J., Langerman, S., Patrascu, M., Taslakian, P.: Necklaces, convolutions, and X+Y. Algorithmica **69**(2), 294–314 (2014)
6. Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: On table arrangements, scrabble freaks, and jumbled pattern matching. In: P. Boldi, L. Gargano (eds.) Fun with Algorithms, FUN 2010, *LNCS*, vol. 6099, pp. 89–101. Springer (2010)
7. Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: Algorithms for jumbled pattern matching in strings. Int. J. Found. Comput. Sci. **23**(2), 357–374 (2012)
8. Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: On approximate jumbled pattern matching in strings. Theory Comput. Syst. **50**(1), 35–51 (2012)
9. Butman, A., Eres, R., Landau, G.M.: Scaled and permuted string matching. Inf. Process. Lett. **92**(6), 293–297 (2004)
10. Chan, T.M., Lewenstein, M.: Clustered integer 3SUM via additive combinatorics. In: R.A. Servedio, R. Rubinfeld (eds.) 47th Annual ACM on Symposium on Theory of Computing, STOC 2015, pp. 31–40. ACM (2015)
11. Cicalese, F., Fici, G., Lipták, Z.: Searching for jumbled patterns in strings. In: J. Holub, J. Žďárek (eds.) Prague Stringology Conference 2009. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague (2009)
12. Cicalese, F., Gagie, T., Giaquinta, E., Laber, E.S., Lipták, Z., Rizzi, R., Tomescu, A.I.: Indexes for jumbled pattern matching in strings, trees and graphs. In: O. Kurland, M. Lewenstein, E. Porat (eds.) String Processing and Information Retrieval, SPIRE 2013, *LNCS*, vol. 8214, pp. 56–63. Springer (2013)
13. Fellows, M.R., Fertin, G., Hermelin, D., Vialette, S.: Upper and lower bounds for finding connected motifs in vertex-colored graphs. J. Comput. Syst. Sci. **77**(4), 799–811 (2011)
14. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with $O(1)$ worst case access time. J. ACM **31**(3), 538–544 (1984)
15. Gagie, T., Hermelin, D., Landau, G.M., Weimann, O.: Binary jumbled pattern matching on trees and tree-like structures. Algorithmica **73**(3), 571–588 (2015)
16. Hagerup, T.: Sorting and searching on the word RAM. In: M. Morvan, C. Meinel, D. Krob (eds.) Symposium on Theoretical Aspects of Computer Science, STACS 1998, *LNCS*, vol. 1373, pp. 366–398. Springer, Berlin Heidelberg (1998)
17. Hermelin, D., Landau, G.M., Rabinovich, Y., Weimann, O.: Binary jumbled pattern matching via all-pairs shortest paths. CoRR **abs/1401.2065** (2014)
18. Kociumaka, T., Radoszewski, J., Rytter, W.: Efficient indexes for jumbled pattern matching with constant-sized alphabet. In: H.L. Bodlaender, G.F. Italiano (eds.) Algorithms, ESA 2013, *LNCS*, vol. 8125, pp. 625–636. Springer (2013)
19. Lacroix, V., Fernandes, C.G., Sagot, M.: Motif search in graphs: Application to metabolic networks. IEEE/ACM Transactions on Computational Biology and Bioinformatics **3**(4), 360–368 (2006)
20. Moosa, T.M., Rahman, M.S.: Indexing permutations for binary strings. Inf. Process. Lett. **110**(18-19), 795–798 (2010)
21. Moosa, T.M., Rahman, M.S.: Sub-quadratic time and linear space data structures for permutation matching in binary strings. J. Discrete Algorithms **10**, 5–9 (2012)
22. Munro, J.I., Nekrich, Y., Vitter, J.S.: Fast construction of wavelet trees. In: E.S. de Moura, M. Crochemore (eds.) String Processing and Information Retrieval, SPIRE 2014, *LNCS*, vol. 8799, pp. 101–110. Springer (2014). Accepted to Theor. Comput. Sci. (10.1016/j.tcs.2015.11.011).
23. Williams, R.: Faster all-pairs shortest paths via circuit complexity. In: D.B. Shmoys (ed.) 46th Annual ACM on Symposium on Theory of Computing, STOC 2014, pp. 664–673. ACM (2014)