Theoretical Computer Science 532 (2014) 14–30

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Fast relative Lempel–Ziv self-index for similar sequences

Huy Hoang Do [a,*], Jesper Jansson [b], Kunihiko Sadakane [c], Wing-Kin Sung [a]

[a] National University of Singapore, COM 1, 13 Computing Drive, Singapore 117417, Singapore
[b] Laboratory of Mathematical Bioinformatics, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto 611-0011, Japan
[c] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

ARTICLE INFO

Keywords:
String decomposition
Textual substitution
Exact pattern searching
FM-index
Suffix range
Rank and select

ABSTRACT

Recent advances in biotechnology and web technology are continuously generating huge collections of similar strings. People now face the problem of storing them compactly while supporting fast pattern searching. One compression scheme called *relative Lempel–Ziv compression* uses textual substitutions from a reference text to represent each string in S as a concatenation of substrings from a reference string R. This basic scheme gives a good compression ratio when every string in S is similar to R, but does not provide any pattern searching functionality. Here, we describe a new data structure based on relative Lempel–Ziv compression that is space-efficient and also supports fast pattern searching.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

There is an increasing need for indexing methods that can store collections of *similar* strings (or repetitive text) compactly while supporting fast pattern searching queries. For example, in genomic applications, the sequencing of individual genomes is becoming a feasible task. The “1000 Genomes Project” [1], aimed at characterizing common human genetic variations, has already sequenced the partial genomes of a large number of persons from various populations. Aligning a read from a sample to multiple human genomes has been proven to be useful for identifying polymorphisms [38]. In the near future, researchers will face the problem of storing those individual (and highly similar) genomic sequences compactly and indexing them efficiently. As another example, Wikipedia documents are modified and snapshots are taken every day to remember older versions of the data. Typically, changes between versions are small. Hence, fast indexing methods for compressed similar texts may allow people to search archived versions of Wikipedia documents quickly. The above applications motivate the following general task:

The string set self-indexing problem: Given a set of strings $\mathcal{S} = \{S_1, \dots, S_t\}$, construct a data structure that can subsequently report all exact occurrences in $\mathcal{S}$ of any query pattern without using $\mathcal{S}$.

This paper is concerned with the case where the given strings are similar. Before stating our new results, we survey some existing data compression methods and compressed indexes that are suitable for sets of similar sequences in the next two subsections. Throughout the paper, we use the terms “string”, “sequence”, and “text” synonymously.

1.1. Similar text compression methods

To store a single string S of length n, modern compression methods often guarantee to use less than or equals to $nH_k(S)$ bits where $H_k(S)$ is the k-th order empirical entropy of the string S. However, this entropy measurement may not be a

* Corresponding author.
E-mail addresses: hoang@comp.nus.edu.sg (H.H. Do), jj@kuicr.kyoto-u.ac.jp (J. Jansson), sada@nii.ac.jp (K. Sadakane), ksung@comp.nus.edu.sg (W.-K. Sung).

0304-3975/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.tcs.2013.07.024

Base compression method	Popular in	Effective(*)	Search time(**)	Reference
LZ78	GIF image	No	linear	[3,13,35]
BWT-transform	bzip2	No	linear	[30]
LZ77	zip	Yes	quadratic	[23]
Grammar based		Yes	quadratic	[10,17]
Restricted structure		Yes	quadratic	[21]
RLZ		Yes	linear	this paper

Fig. 1. Summary of the compressed indexing structures. (*): Effective for similar sequences. (**): The search time is expressed in terms of the pattern length.

good upper bound for repetitive texts whose repeats are longer than k. For example, the storage based on entropy bound of the text SS (where $|S| \gg k$) is $2nH_k(S)$ bits. On the other hand, one can easily encode the text in $nH_k(S) + O(\log n)$ bits. Thus, there are methods that achieve the k-th order empirical entropy, yet perform poorly for repetitive texts [39]. As a consequence, compression methods have been designed for specific types of repetitive texts in biology. For example, GenCompress [8] compresses a text considering approximate repeats. Christley et al. [9] and Kuruppu et al. [25] compressed DNA sequences with respect to a reference sequence. BioCompress [20], XM [6], and COMRAD [24] are other repetitive compressors designed specifically for DNA. Alternative approaches include methods based on grammar compression (for example, Re-pair [28] was one of the first effective grammar-based compression methods) and LZ77 compression [41] for general repetitive texts. Cfact [34] and Offlines [2] greedily replace duplicate text with shorter codes.

The compression methods above can store repetitive texts compactly, but do not allow random access to the compressed text directly. Previous work has addressed this issue. Kreft and Navarro [22] provided the first efficient random access operations for the LZ77 method. Bille et al. [4] built additional data structures on top of an existing grammar-based compression scheme to allow random access of any region with only logarithmic extra time per query.

1.2. Compressed indexes for similar text

Although the above compression schemes can compress similar sequences, they do not allow us to search for the occurrences of an arbitrary pattern quickly. Below, we survey some specialized data structures for indexing repetitive texts. In a pioneering paper of Mäkinen et al. [30], a repetitive text is defined as a collection of strings of total length N, where the strings are assumed to be highly similar, each string length is approximately n, and the strings share an alphabet of size σ. They employed run-length encoding to reduce the redundancy of a suffix array structure. Their approach shrinks the total index size greatly, but the space of the index is still proportional to the number of strings. In another paper, Huang et al. [21] assumed that every string contains at most m' point mutations with respect to a reference string. They designed a space-efficient data structure of size $O(n\log\sigma + m'\log m')$ bits to encode all such strings. Although the resulting data structure is small, their approach cannot index certain other types of similar strings such as *genome rearrangements*, formed by swapping substrings in genomic sequences, efficiently. (When only a few such rearrangements have occurred, long substrings of the genomic sequences will be preserved; they just occur in a different order.) Kreft and Navarro [23] built a self-index based on LZ77 compression. If the text of length N can be compressed using m LZ77 phrases, their data structure is of size $2m\log N + m\log m + 5m\log\sigma + O(m) + o(N)$ bits, but the query time is $O(\ell^2 h + (\ell + occ)\log N)$, i.e., quadratic in the pattern length ℓ, and also dependent on a variable h, which equals to m in the worst case. In another line of research, Claude and Navarro [10] proposed a self-index for grammar-based compression methods. It uses $O(r\log r) + r\log N$ bits, where r is the number of rules generated by their grammar compression, h is the height of the grammar tree, and the resulting query time is quadratic $(O((\ell^2 + h(\ell + occ))\log r))$. Using another technique for constructing a grammar from the LZ77 phrasing, Gagie et al. [17] obtained a data structure of size $2r\log r + O(m(\log n + \log m \log\log m))$ and query time $O(\ell^2 + (\ell + occ)\log\log N)$. Some results for LZ78 compression and FM-index were given in [3,13,35]. They have good query time but require $O(NH_k)$ bit-space in the worst case. They may not be good enough to index a repetitive text in practice [39] or in theory [36]. In summary, existing indexes for a set of similar strings either require: (1) a lot of space, (2) the text to have some special structure, or (3) quadratic query time (for a summary, see the table in Fig. 1).

1.3. Our results

Our main contribution is a compressed static indexing data structure with two alternative space–time trade-offs. The smaller alternative can store a set of strings $\mathcal{S}$ relatively to a reference string R in asymptotically optimal space. The larger alternative improves the query time at the expense of using more space. The results are summarized as follows:

Theorem 1.1. *Given a reference string R of length n over an alphabet Σ of size $\sigma = O(\log^a n)$ for some constant a and a set of strings $\mathcal{S} = \{S_1, \ldots, S_t\}$ over Σ, let m be the smallest possible number of substrings of R (a.k.a. factors) to represent $\mathcal{S}$. All exact occurrences of any query pattern P of length ℓ can be reported using either of the following alternatives. Their complexity specifications are*:

(a) $(2 + \frac{1}{\epsilon})nH_k(R) + O(n) + O(m\log n)$ *bits and* $O(\ell\log^\epsilon n + occ \cdot (\log_\sigma^\epsilon n + \frac{\log m}{\log n}))$ *query time*;

(b) $(2 + \frac{1}{\epsilon})nH_k(R) + O(n) + O(m\log n\log\log n)$ *bits and* $O(\ell\log\log n + occ \cdot (\log_\sigma^\epsilon n + \frac{\log m}{\log n}))$ *query time,*

R = ACGTGATAG
S_1 = TGATAGACG = TGATAG, ACG = 8 2
S_2 = GAGTACTA = GA, GT, AC, TA = 5 6 1 7
S_3 = GTACGT = GT, ACGT = 6 3
S_4 = AGGA = AG, GA = 4 5

(a)

$T[..]$	Factor	Pos. in R
1	AC	1..2
2	ACG	1..3
3	ACGT	1..4
4	AG	8..9
5	GA	5..6
6	GT	3..4
7	TA	7..8
8	TGATAG	4..9

(b)

$\overline{T}[..]$	Factor (rev.)	Pos. in R
1	GA	5..6
2	TA	7..8
3	AC	1..2
4	AG	8..9
5	TGATAG	4..9
6	ACG	1..3
7	GT	3..4
8	ACGT	1..4

(c)

Fig. 2. (a) A reference string R and a set of strings $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$ decomposed into the smallest possible number of factors from R. (b) The array $T[1..8]$ (to be defined in Section 2) consists of the distinct factors sorted in lexicographical order. (c) The array $\overline{T}[1..8]$.

where occ is the number of occurrences of P, k is any positive integer less than $\log_\sigma n$, and $\epsilon \leqslant 1$ is a constant. For both alternatives, the data structure can be constructed in $O(\sum_{i=1}^{t} |S_i| + (n+m)\log(n+m))$ time.

Our compression scheme is based on a variant of the relative Lempel–Ziv (RLZ) compression scheme from [25]. It represents each $S_i \in \mathcal{S}$ as a concatenation of substrings of R (referred to as *factors*) obtained from the LZ77-like factorization of R. See Fig. 2 for an example. Experiments on large scale genomic data in [25] have shown that this method yields good compression ratios for repetitive texts even when parts of the sequence are rearranged.

In this paper, we assume that the reference R is given. In case where R is not available, we can apply the method of Kuruppu et al. [26] to find a suitable one. We also assume the alphabet size σ is in polylogarithmic of the word length (i.e., $\sigma = O(\log^a n)$ for some constant a). For larger alphabets, e.g. $\sigma = \Omega(n^\alpha)$, the query time needs an additional term of $O(\ell \log\sigma / \log\log n + occ \cdot \log\sigma)$ and the space needs an additional term of $O(n \log\sigma \log\log n / \log n) = o(n \log\sigma)$.

Both alternatives in Theorem 1.1 use the same pattern searching algorithm. The algorithm considers two cases: Case 1, where the pattern P is a substring of a single factor; and case 2, where P crosses at least one boundary between two factors. (See Fig. 4.) For case 2, the pattern is partitioned into two parts: left and right. The left part ends at the end of the first factor, while the right part begins at the start of the second factor. For each possible partition of the pattern, the left part and right part are searched independently and then joined together by an appropriate 2D range query data structure. However, to avoid the quadratic pattern search time, we use multiple tricks to reuse results between the searches in each partition.

We remark that recently, Gagie et al. [17] independently proposed a similar method to index a set of sequences. Their space complexity is $O(nH_k(R) + n + m(\log n + \log m \log\log m))$ bits, and the query time is $O((\ell + occ)\log^\epsilon n)$, where $\epsilon > 0$. Thus, compared to the method in our Theorem 1.1(a), their method always uses more space while having similar time. Compared to that in Theorem 1.1(b), theirs is slower while having asymptotically comparable space. Also note that in their method, the reference sequence is restricted. It must be one of the sequences in $\mathcal{S}$ (otherwise false occurrences may be reported).

The paper is organized as follows. Section 2 defines the notation used throughout the paper and outlines the framework of our new data structures. Section 3 describes some auxiliary data structures used in our construction. These data structures are known in the literature; however, we also present some improvements which may be of independent interest. Section 3.4 presents a new data structure for answering a restricted type of 2D range queries. Sections 4–7 describe further technical details of our main data structure.

2. Data structure framework

2.1. The relative Lempel–Ziv (RLZ) compression scheme

Let R be a reference sequence of length n over an alphabet Σ and let $\mathcal{S} = \{S_1, \ldots, S_t\}$ be a given set of strings over Σ. Each sequence $S_i \in \mathcal{S}$ is compressed based on R by relative Lempel–Ziv (RLZ) compression [25]. Precisely, given two strings S and R, where R contains all the symbols in S, the *Lempel–Ziv factorization* (or *parsing*) of S relative to R, denoted by $LZ(S|R)$,

Input: A string S and the BWT of $\overline{R}$
Output: A decomposition of S, i.e., $S_1 \ldots S_k$
1: $i = 1$; $k = 1$:
2: **while** $i \leqslant |S|$ **do**
3: By backward search on $\overline{R}$, identify the longest prefix $S_k = S[i..j]$ of $S[i..|S|]$ such that S_k is a substring of R.
4: $k = k + 1$; $i = j + 1$;
5: **end while**
6: Report $S_1 \ldots S_k$

Fig. 3. Algorithm to decompose a string into RLZ factors.

is a way to express S as a concatenation of substrings of the form $S = w_0 w_1 w_2 \ldots w_z$ such that: (1) w_0 is an empty string; and (2) w_i for $i > 0$ is a non-empty substring of S and w_i is the longest prefix of $S[(|w_0..w_{i-1}| + 1)..|S|]$ that occurs in R. Each substring w_i is called a *factor* (or *phrase*), and can be represented by a pair of numbers (p_i, l_i), where p_i is a starting position of w_i in R and l_i denotes the length of w_i.

$LZ(S|R)$ was suggested in [25]. The algorithm, which runs in linear time, is summarized in Fig. 3. By definition, the decomposition guarantees that no factor can be expanded any further to the right. Furthermore, the RLZ compression scheme has the following property:

Lemma 2.1. *$LZ(S|R)$ represents S using the smallest possible number of factors.*

Proof. Consider the algorithm to decompose a string into RLZ factors in Fig. 3. Let $dist_R(S)$ denote the minimal number of factors of R to represent S. We prove the property by induction. First, any string S of length 1 has a decomposition using $dist_R(S) = 1$ factor of R.

Next, by induction, for any string X of length less than ℓ, we assume X can be constructed using $dist_R(X)$ factors of R. Now, consider a string S of length ℓ and assume the algorithm reports $S = S_1 \ldots S_k$. To obtain a contradiction, suppose the optimal decomposition is $S'_1 \ldots S'_{k'}$ where $k' < k$. Since the algorithm always finds the longest string, we know that $|S_1| \geqslant |S'_1|$. Note that for $S[|S_1| + 1..\ell]$, the algorithm will decompose it into $S_2 \ldots S_k$, which consists of $k - 1$ factors. The induction hypothesis states that $dist_R(S[|S_1| + 1..\ell]) = k - 1$. As $S[|S'_1| + 1..\ell]$ is longer than $S[|S_1| + 1..\ell]$, we have $dist_R(S[|S'_1| + 1..\ell]) \geqslant dist_R(S[|S_1| + 1..\ell])$. Hence, $k' - 1 = dist_R(S[|S'_1| + 1..\ell]) \geqslant dist_R(S[|S_1| + 1..\ell]) = k - 1$. Contradiction. □

For every $S_i \in \mathcal{S}$, denote the Lempel–Ziv factorization of each S_i relative to R by $S_i = S_{i1} S_{i2} \ldots S_{ic_i}$. Define $m = \sum_{i=1}^{t} c_i$. By Lemma 1, m is in fact the smallest possible number of factors to represent $\mathcal{S}$. Next, take all the s *distinct factors* that appear in the factorizations for $\mathcal{S}$ and let $T[1..s]$ be an array containing these factors sorted in lexicographical order (see Fig. 2(b)). Note that $s \leqslant \min\{n^2, m\}$. Our data structure stores $T[1..s]$ in $O(s \log n)$ bits by encoding each $T[j]$ by its starting and ending positions in the reference string R, and the set $\mathcal{S}$ in $O(m \log s) = O(m \log n)$ bits by representing each $S_i \in \mathcal{S}$ as a list of indices from $T[1..s]$ (see Fig. 2(a)).

Let $F[1..m]$ be the lexicographically sorted array of all non-empty suffixes in $\mathcal{S}$ that start with a factor; i.e., each element $F[y]$ is of the form $S_{ip} S_{i(p+1)} \ldots S_{ic_i}$, and is called a *factor suffix* from here on. See Fig. 12(a) for an example. Importantly, our data structure does not store $F[1..m]$ explicitly. For any string x, $\bar{x}$ denotes its reverse. Let $\overline{T}[1..s]$ be an array of all reversed distinct factors $\overline{S}_{ij}$ sorted lexicographically. By using the relative Lempel–Ziv decomposition, each sequence S_i can be viewed as a new sequence S'_i based on the alphabet of all the distinct factors in $T[1..s]$ (see Fig. 2).

2.2. Pattern searching

To find the occurrences of a query pattern P in $\mathcal{S}$, we follow the basic strategy outlined in Section 1.3. Suppose P is a query pattern of length ℓ. Each occurrence of P in $S_1, \ldots, S_t$ belongs to one of the following two main cases; see Fig. 4:

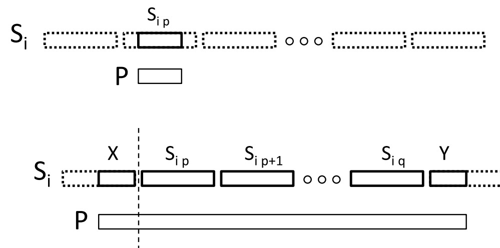

Fig. 4. When P occurs in string S_i, there are two possibilities, referred to as case 1 and case 2. In case 1 (shown on the left), P is contained inside a single factor S_{ip}. In case 2 (shown on the right), P stretches across two or more factors $S_{i(p-1)}, S_{ip}, \ldots, S_{i(q+1)}$.

	AC-TA	ACG	ACGT	AG-GA	GA-GT-AC-TA	GA	GT-AC-TA	GT-ACGT	TA	TGATAG-ACG
$				1	1			1		1
GA							1			
TA										
AC									1	
AG						1				
TGATAG		1								
ACG										
GT	1		1							
ACGT										

Fig. 5. Each row represents the string $\overline{T}[i]$ in reverse; each column corresponds to a factor suffix $F[i]$ (with dashes to mark factor boundaries). The locations of the number "1" in the matrix mark the factor in the row preceding the suffix in the column. Consider an example pattern "AGTA". There are 5 possible partitions of the pattern: "-AGTA", "A-GTA", "AG-TA", "AGT-A" and "AGTA-". Using the index of the sequences in Fig. 2, the big shaded box is a 2D query for "A-GTA" and the small shaded box is a 2D query for "AG-TA".

- **Case 1:** P lies completely inside one factor, denoted by S_{ip}.
- **Case 2:** P is not a substring of a single factor, i.e., $P = XS_{ip} \dots S_{iq}Y$, where X is a suffix of $S_{i(p-1)}$ and Y is a prefix of $S_{i(q+1)}$.

(Observe that the case $P = XY$ is an instance of case 2.) To locate all occurrences of P, our data structure uses a number of auxiliary data structures (explained in Subsection 2.3), to report all occurrences of P in $\mathcal{S}$ according to case 1 and case 2 separately. Let occ_1 and occ_2 be the number of occurrences of P as in case 1 and case 2, respectively.

Case 1: [P occurs inside a factor] Since all the factors are substrings of the reference R, the pattern is first searched for in the reference. Then, the factors that cover an occurrence of the pattern in the previous step are reported as the result. This case takes $O(\ell + occ_1 \log^{\epsilon} n)$ time, as discussed in detail in Section 4.

Case 2: [P is not a substring of a single factor] As illustrated in Fig. 4, in this case, every occurrence of P can be divided into two parts: the *left part* ($P[1..j]$ matches some suffix of a factor), and the *right part* ($P[j+1..\ell]$ matches a factor suffix). To find the occurrences of this case, we try to match all the $(\ell+1)$ possible partitions ($P[1..j]$, $P[j+1..\ell]$) of the pattern. For each partition, the left parts are matched against the set of reversed factors in $\overline{T}$. The successful matches are represented by a range in $\overline{T}$. The right parts are matched against the set of factor suffixes in F. The results are also represented by a range in F. Then, the successful matches of the left part $P[1..j]$ and right part $P[j+1..\ell]$ are combined and validated using a 2D-range query data structure. (See Fig. 5 for an example.)

2.3. Overview of our main data structure

The data structure for case 1 is called $\mathcal{I}(T)$ and is defined in Section 4. It finds all occurrences of P corresponding to case 1 in $O(\ell + occ_1 \log^{\epsilon} n)$ time and uses $2n + o(n) + O(s \log n)$ bits (Theorem 4.1).

The data structures that facilitate the searching for case 2 are more complicated and consist of three components: (i) $\mathcal{X}(\overline{T})$ to match the left parts; (ii) $\mathcal{Y}(F, T)$ to match the right parts; and (iii) $\mathcal{M}$ to report the correct combinations of the left parts and right parts. Further technical details of $\mathcal{X}(\overline{T})$, $\mathcal{Y}(F, T)$, and $\mathcal{M}$ are given in Sections 5, 6, and 3.4, respectively. Note that each of the two alternatives in Theorem 1.1 uses the same components for (i) and (ii). Their space and time trade-offs result from using different versions of (iii). The usage of each component is summarized as follows:

(i) First, $\mathcal{X}(\overline{T})$ in Section 5 uses $O(s \log n) + o(n)$ bits space. It finds all occurrences of prefixes of P that are equal to a suffix of a factor $S_{i(p-1)}$ in $O(\ell \log\log n)$ time. More precisely, $\mathcal{X}(\overline{T})$ returns, for every j, the maximal range $st_j..ed_j$ in $\overline{T}$ such that $\overline{P[1..j]}$ is a prefix of every element in $\overline{T}[st_j], \dots, \overline{T}[ed_j]$.

(ii) Second, $\mathcal{Y}(F, T)$ in Section 6 uses $(2 + 1/\epsilon)nH_k(R) + 2.55n + o(n \log \sigma) + O(m \log n)$ bits space. It finds all occurrences of suffixes of P that are equal to a prefix of a factor suffix in F, i.e., $S_{ip} \dots S_{iq}Y$, where Y is a prefix of $S_{i(q+1)}$, in $O(\ell(\log \sigma / \log\log n + \log\log n))$ time. More precisely, $\mathcal{Y}(F, T)$ returns, for every j, the maximal range $st'_j..ed'_j$ such that $P[(j+1)..\ell]$ is a prefix of every element in $F[st'_j], \dots, F[ed'_j]$.

(iii) Third, we encode all combinations of $S_{i(p-1)}$ and $S_{ip} \dots S_{iq} Y$ as follows: Define M to be a binary $(s \times m)$-matrix where $M[x, y] = 1$ if and only if $\overline{T}[x]$ is the preceding factor of the suffix $F[y]$, i.e., $F[y] = S_{ip} S_{i(p+1)} \dots S_{ic_i}$ and $\overline{S}_{i(p-1)} = \overline{T}[x]$ is the x-th lexicographically smallest in $\overline{T}$. Note that each column of the matrix M contains exactly one 1. (See Fig. 5.)

Lemma 2.2. *All case 2 occurrences of P can be found by listing the entries equal to 1 in the rectangles $[st_j, ed_j] \times [st'_j, ed'_j]$ in M, for all j.*

Proof. ($\rightarrow$) Consider an occurrence of case 2 of P in S_i, that is, $S_i[s..e] = P$ and $S_i[a..t]$ is a factor and $a \leqslant s < t < e$. Let $\overline{T}[p]$ be the entry in $\overline{T}$ that represents the factor $S_i[a..t]$. We have $S_i[t+1..|S_i|]$ is a factor suffix. Let $F[p']$ be the entry in F that represents $S_i[t+1..|S_i|]$. By the definition of the matrix M, there is an entry 1 in $M[p, p']$. Consider $j = (t - s + 1)$, we have $P[1..j] = S_i[a..s]$ is a suffix of $\overline{T}[p]$, and therefore, $st_j \leqslant p \leqslant ed_j$. Similarly, $P[j..|P|]$ is prefix of $F[p']$, and therefore, $st'_j \leqslant p' \leqslant ed'_j$. Therefore, the occurrence $S_i[s..e] = P$ implies a number 1 in the specified rectangle.

($\leftarrow$) Consider a number 1 in the region $[st_j, ed_j] \times [st'_j, ed'_j]$. The position of the occurrence can be found as follows. Let (i', j') be the position of the number 1. Let $S_i[p..|S_i|]$ be the factor suffix of $F[j']$. We have $P[j+1..|P|] = S_i[p..p+|P|-j]$ Since $\overline{T}[i']$ is the previous factor of $F[j']$, $S_i[p-j..p-1] = P[1..j]$. Therefore, P occurs in S_i from position $p - j$ to position $|P|$. □

For each pattern P of length ℓ, we may need up to ℓ queries in the matrix M to find all the results. (See Fig. 12(b) for an example.) Section 3 gives two alternative 2D range query data structures $\mathcal{M}$ that support the operation query_2d$(M, [st, ed], [st', ed'])$ on M for finding these entries: If $\mathcal{M}$ is of size $O(m \log s \log\log s)$ bits, all entries equal to 1 can be found in $O((1 + occ)\log\log s)$ time for each query, and if $\mathcal{M}$ is of size $O(m \log s)$ bits, the query takes $O((1 + occ) \cdot \log^{\epsilon} s)$ time.

As a final step, we need a data structure to decode all occurrences of cases 1 and 2 to find their actual locations in $\mathcal{S}$. This simple data structure, called $\mathcal{D}$, is used to convert indices of F to their exact locations in $\mathcal{S}$. It requires $O(m \log n)$ bits, and $O(\log m/\log n + \log\log n)$ time for decoding each occurrence. The details are presented in Section 7.

Note that the data structure is designed as a static database. Once the reference sequence is given, the input strings can be factorized in linear time using the algorithm in Section 2.1, i.e., using $O(N)$ time where N is the total length of the input. The construction algorithms for $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{I}$ are not complicated. The internal components can be directly constructed based on their definitions using only constant number of sort and scan operations on some arrays. All the external components (Section 3) can be built in $O(m \log m)$ time. The whole data structure can be constructed in $O(N + (n+m)\log(n+m))$ time.

The total space equals the sum of the spaces of all components, namely: arrays T and $\overline{T}$; data structures: $\mathcal{I}(T)$, $\mathcal{X}(\overline{T})$, $\mathcal{Y}(F, T)$, $\mathcal{M}$ and $\mathcal{D}$. (Note that the FM-indexes of R although counted only inside $\mathcal{Y}(F, T)$, it is shared with $\mathcal{I}(T)$ and $\mathcal{X}(\overline{T})$ for looking up values in suffix array.) Putting everything together, the total space requirement is $(2 + 1/\epsilon)nH_k(R) + 5.55n + O(m \log n)$ bits, while all occurrences of P in S can be found in $O(\ell(\log\sigma/\log\log n + \log^{\epsilon} n) + occ \cdot (\log^{\epsilon}_{\sigma} n + \frac{\log m}{\log n}))$ time; or $(2 + 1/\epsilon)nH_k(R) + 5.55(n) + O(m \log n \log\log n)$ bits and $O(\ell(\log\sigma/\log\log n + \log\log n) + occ \cdot (\log^{\epsilon}_{\sigma} n + \frac{\log m}{\log n}))$ time. When σ is in $\Omega(\log^{O(1)} n)$, the term $\log\sigma/\log\log n$ becomes $O(1)$. We thus obtain Theorem 1.1 above.

3. Some useful auxiliary data structures

3.1. *rank* and *select* and integer data structures from the literature

Let $B[1..n]$ be a bit vector of length n with k ones and $n - k$ zeros. The *rank* and *select* data structure supports two operations: $\text{rank}_B(i)$ returns the number of ones in $B[1..i]$; and $\text{select}_B(i)$ returns the position in B of the ith one. Given an array $A[1..n]$ of non-negative integers, where each element is at most m, we are interested in the following operations: $\text{max_index}_A(i, j)$ returns $\arg\max_{k \in i..j} A[k]$, and $\text{range_query}_A(i, j, v)$ returns the set $\{k \in i..j\colon A[k] \geqslant v\}$. We also need one more operation for the case when $A[1..n]$ is sorted in non-decreasing order, called $\text{successor_index}_A(v)$, which returns the smallest index i such that $A[i] \geqslant v$. The data structure for this operation is called the *y-fast trie* [40]. The complexities of some existing data structures supporting the above operations are listed in the table in Fig. 6.

Operation	Extra space	Time	Reference	Remark
$\text{rank}_B(i)$, $\text{select}_B(i)$	$\log\binom{n}{k} + o(n)$	$O(1)$	[33]	
$\text{max_index}_A(i, j)$	$2n + o(n)$	$O(1)$	[16]	
$\text{range_query}_A(i, j, v)$	$O(n \log m)$	$O(1 + occ)$	[31], p. 660	
$\text{successor_index}_A(v)$	$O(n \log m)$	$O(\log\log m)$	[40]	A is sorted

Fig. 6. The time and space complexities to support the operations defined above.

3.2. Suffix array and FM-index

Consider any string R with a special terminating character \$ which is lexicographically smaller than all the other characters. The *suffix array* SA_R is the array of integers specifying the starting positions of all suffixes of R sorted lexicographically. For any string P, let st and ed be the smallest and the biggest, respectively, indexes such that P is the prefix of suffix $SA_R[i]$ for all $st \leqslant i \leqslant ed$. Then, (st, ed) is called a *suffix range* or SA_R*-range* of P, i.e., P occurs at $SA_R[st+1], \ldots, SA_R[ed]$ in R. For any given string P specified by its suffix range (st, ed) in SA_R, an *FM-index of R* supports the following operations: $\text{lookup}_R(i)$ returns the value of $SA_R[i]$; $\Psi_R(i)$ returns the index j such that $SA_R[j] = SA_R[i]+1$; and $\text{backward_search}_R(c, (st, ed))$ returns the suffix range in SA_R of the string cP, where c is any character and (st, ed) is the suffix range of P.

Lemma 3.1. *Given any string R of length n over an alphabet of size σ, the FM-index of R uses $nH_k(R) + O(n\log\sigma\log\log n/\log n)$ bits and supports* backward_search_R *in $O(\log\sigma/\log\log n)$ time and Ψ_R in $O(1)$ time (where $k < \log_\sigma n$). Given additional $(1/\epsilon)nH_k(R) + 2(\log e + 1)n + o(n)$ bits,* lookup_R *can be supported in $O(\log_\sigma^\epsilon n + \log\sigma)$ time.*

Proof. The FM-index, that uses $nH_k(R) + O(n\log\sigma\log\log n/\log n)$ bits and supports the operations backward_search_R and Ψ_R within the specified time, is described in [12,29,32]. [19] showed how to support Ψ_R and lookup_R using $(1+1/\epsilon)nH_k(R) + 2(\log e + 1)n + o(n)$ bits. If we store the FM-index and the data structure in [19] separately, it takes $(2+1/\epsilon)nH_k(R) + O(n)$ bits. Since these two data structures have some overlap, we can reduce the space when storing both of them by $nH_k(R)$ bits as follows:

Let Σ^i be an alphabet, such that for any character $c \in \Sigma^i$, $c = a_1a_2\ldots a_{2^i}$, where $a_j \in \Sigma$. Let R^i be the sequence using the alphabet Σ^i, such that $R^i[j] = R[2^i \times j]R[2^i \times j + 1]\ldots R[2^i \times j + 2^i - 1]$. Let Ψ^i be the Ψ function for sequence R^i.

In [19], a recursive data structure of $(1+1/\epsilon)$ level is stored to compute the value of SA_R. Let $h' = \log\log_\sigma n$ and $h = \log\log n$, they store Ψ^i for $i = 0, h'\epsilon, 2h'\epsilon, \ldots, h'$ and SA_{R^h}. However, since $\Psi_R = \Psi^0$, and Ψ_R is provided by the normal FM-index, we don't need additional space for Ψ^0. That saves $nH_k(R)$ bits. □

In the FM-index above, the compression technique is only effective for moderate size alphabet (i.e., $\sigma = O(\log^a n)$ for some constant a). When the alphabet is larger (e.g. $\sigma = O(n^\alpha)$), the sequence becomes more like a permutation of distinct numbers. The second term in the space complexity can surpass the main $nH_k(R)$ term; and $H_k(R)$ grows to its $\log n$ upper bound. Moreover, the running time with $\log\sigma$ is no longer a small number. A *general FM-index* is an FM-index extended to alphabets of unbounded size. It is not compressed, but its query time is only $\log\log\sigma$. The next lemma is our simple extension of the normal FM-index to the general FM-index case, obtained by applying the result from [18] and some additional arrays:

Lemma 3.2. *Given any string S of length m over an alphabet of size s, there exists a general FM-index of S that uses $m\log s + o(m\log s)$ bits and supports* backward_search_S *in $O(\log\log s)$ time and Ψ_S in $O(1)$ time. Using an additional $m\log s + o(m\log s)$ bits,* lookup_S *can be supported in $O(\log m/\log s)$ time.*

Proof. The original data structure for the FM-index uses $m\log s + o(m\log s)$ bits space and support backward_search_S and Ψ_S as described in Sections 3.1 and 3.2 of [18]. We now explain how to support lookup_S using the stated space and time complexity.

Define $\Psi_S^1(i) = \Psi_S(i)$ and $\Psi_S^k(i) = \Psi_S(\Psi_S^{k-1}(i))$ for $k > 1$. Then, we have, $SA_S[\Psi_S^k(i)] = SA_S[i] + k$. Let $t = \log m/\log s$. Use the additional bits to store:

- A succinct bit vector $B[1..m]$ such that $B[i] = 1$ if and only if $SA_S[i] \bmod t \equiv 0$.
- An array of integers $V[1..m/t]$ such that $V[i] = SA_S[\text{select}_B(i)]$.

Since $SA_S[1..m]$ contains all values from 1 to m, there exists a value i' such that $SA_S[i'] \bmod t \equiv 0$ and $SA_S[i] - SA_S[i'] < t$. From the definitions of B and Ψ_S, we have $B[i'] = 1$ and $i' = \Psi_S^k(i)$ and $k < t$. To find i', iteratively compute $\Psi_S^k(i)$ until $B[\Psi_S^k(i)]$ is 1; this enumeration takes at most $O(t)$ time. The value of $SA_S[i']$ can be looked up from V. Then, $SA_S[i] = SA_S[i'] - k$. Therefore, the value of $SA_S[i]$ takes $O(\log m/\log s)$ time to compute.

For the extra space complexity, the bit vector B uses $O(m)$ bits. The array V uses $m \cdot \log s/(\log m/\log s) \leqslant m\log s$ bits. In total, we use $2m\log s + o(m\log s)$ additional bits. □

3.3. Bi-directional FM-index

Recall that, given a suffix array SA_R, a pattern P can always be represented by an SA-range (st, ed). The traditional FM-index can only extend the search pattern to the left by one character using backward search (i.e., given SA-range of P, *backward_search* returns the SA-range of cP). However, computing the value of array $A[1..|P|]$ in Section 6 requires us to modify the search pattern at both the left end and the right end. A trivial solution is to use a heavier data structure called

the suffix tree. However, even using the existing compressed suffix tree [14,37], the modification at one end of the pattern will take $O(\log n)$ time, which is too much for our requirement in Theorem 1.1 (b).

Therefore, we use a data structure called *bi-directional FM-index* which allows us to extend and delete one character at either end of the pattern in $O(\log\sigma/\log\log n)$ time (where σ is the size of the alphabet).

Consider a sequence R over an alphabet Σ of size σ, the suffix array SA_R, and the suffix array of the reversed sequence $SA_{\overline{R}}$. Given a string X, we let $st(X)$ and $ed(X)$ be the start and the end of the suffix range of X in SA_R, respectively. Similarly, $\overline{st}(X)$ and $\overline{ed}(X)$ denote the start and the end of the suffix range of $\overline{X}$ in $SA_{\overline{R}}$, respectively. Given a pattern $P[1..\ell]$, let r_R be the suffix range of P in SA_R, i.e., $r_R = (st(P), ed(P))$. Let $r_{\overline{R}} = (\overline{st}(P), \overline{ed}(P))$. Let c be any character in Σ. The bi-directional FM-index is a data structure supporting the following four operations:

- *forward_search*$(r_R, r_{\overline{R}}, c)$: returns the new suffix range of pattern Pc in SA_R, and the suffix range of $\overline{Pc}$ in $SA_{\overline{R}}$.
- *backward_search*$(r_R, r_{\overline{R}}, c)$: returns the suffix ranges of cP and $\overline{cP}$ in SA_R and $SA_{\overline{R}}$, respectively.
- *delete_back*$(r_R, r_{\overline{R}})$: returns the suffix range of $P[2..\ell]$ in SA_R, and the suffix range of $\overline{P[2..\ell]}$ in $SA_{\overline{R}}$.
- *delete_front*$(r_R, r_{\overline{R}})$: returns the suffix ranges of $P[1..\ell-1]$ and $\overline{P[1..\ell-1]}$ in SA_R and $SA_{\overline{R}}$, respectively.

In other words, the operation *forward_search* extends the searching pattern by one character to the right, while the operation *backward_search* extends it one character to the left. The operation *delete_back* deletes the leftmost character from the searching pattern, the *delete_front* deletes the rightmost one.

To implement the bi-directional FM-index, we use: the BWT of R, the BWT of $\overline{R}$, the topology of the suffix tree of R, the topology of the suffix tree of $\overline{R}$. (Note that the topology of each tree can be stored in $2.55n + o(n)$ bits each according to [15].) The operations *forward_search* and *backward_search* have been considered before:

Lemma 3.3. *(See Lam et al. [27].) Using the FM-index of R and the FM-index of $\overline{R}$, we can compute operation* forward_search *and operation* backward_search.

We will now present how to implement the operation *delete_back*. For *delete_front*, since R and $\overline{R}$ are symmetric by a string reversal operation, we can implement *delete_front* by a similar procedure but swapping the roles of R and $\overline{R}$.

Formally, the problem of computing *delete_back* is: given $st(cX)$, $ed(cX)$ $\overline{st}(cX)$ and $\overline{ed}(cX)$, compute the values of: $st(X)$, $ed(X)$, $\overline{st}(X)$ and $\overline{ed}(X)$. First, $st(X)$ and $ed(X)$ can be computed using the following lemma:

Lemma 3.4. *Given $st(cX)$ and $ed(cX)$, the values of $st(X)$ and $ed(X)$ can be computed using the suffix link of R in $O(1)$ time.*

Proof. Computing the suffix range of $P[2..\ell]$ in SA_R from that of $P[1..\ell]$ is identical to computing the suffix link in the compressed suffix tree [15,37]. The operation can be done in constant time if the topology of the suffix tree and the function Ψ_R are given. Here Ψ_R is the inverse of the backward search of the FM-index, and can be computed in constant time (See Lemma 3.2). □

Then, the values of $\overline{st}(X)$ and $\overline{ed}(X)$ can be computed using the following lemma:

Lemma 3.5. *$\overline{st}(X)$ and $\overline{ed}(X)$ can be computed by using the following equations*:

$$\overline{st}(X) = \overline{st}(cX) - \sum_{a<c}\big[ed(aX) - st(aX) + 1\big]$$

$$\overline{ed}(X) = \overline{ed}(cX) + \sum_{z>c}\big[ed(zX) - st(zX) + 1\big]$$

Proof. As $\overline{cX} = \overline{X}c$, $\overline{X}$ is prefix of $\overline{cX}$. Note that $(\overline{st}(cX), \overline{ed}(cX))$ is the suffix range of $\overline{cX}$ in $SA_{\overline{R}}$. Therefore, $\overline{st}(X) \leqslant \overline{st}(cX) \leqslant \overline{ed}(cX) \leqslant \overline{ed}(X)$.

Let $\Delta_{\overline{st}} = \overline{st}(cX) - \overline{st}(X)$. Because c is the last character of $\overline{X}c$, hence, $\Delta_{\overline{st}}$ equals the number of occurrences of substrings $\overline{X}a$ in $\overline{R}$ for all character a in Σ and $a < c$. Thus, $\Delta_{\overline{st}}$ equals the number of occurrences of aX in R for all $a < c$. Note that the number of occurrences of aX in R is $[ed(aX) - st(aX) + 1]$. That is, $\overline{st}(X) - \overline{st}(cX) = \sum_{a<c}[ed(aX) - st(aX) + 1]$.

Similarly, let $\Delta_{\overline{ed}} = \overline{ed}(X) - \overline{ed}(cX)$. $\Delta_{\overline{ed}}$ equals the number of occurrences of zX in R for all character z and $z > c$. We obtain $\overline{ed}(X) - \overline{ed}(cX) = \sum_{c<z}[ed(zX) - st(zX) + 1]$. □

From Lemma 3.4 and Lemma 3.5, we can compute *delete_front*. *delete_back* follows similarly. To summarize, we have the following theorem:

Theorem 3.6. *The bi-directional FM-index can be implemented using the BWT of R, the BWT of $\overline{R}$, and the topologies of the two suffix trees of R and $\overline{R}$. It supports all of the four operations in $O(\log\sigma/\log\log n + 1)$ time, and uses $(2 + 1/\epsilon)nH_k(S) + O(n)$ bits.*

Proof. From Lemma 3.3, we can do forward_search and backward_search. From Lemma 3.4, we can compute $st(X)$ and $ed(X)$. From the equations in Lemma 3.5, we can compute $\overline{st}(X)$ and $\overline{ed}(X)$ using the following procedure. For each character a, we compute $(st(aX), ed(aX)) = \text{backward_search}((st(X), ed(X)), a)$. Then, we substitute the values on the right-hand side of the equations. Since each operation backward_search takes $O(\log\sigma/\log\log n + 1)$ time, we can complete the operation *delete_back* in $O(\sigma \log\sigma/\log\log n + 1)$ time.

Note that if we use the wavelet tree in [5] as a component of the BWT (as in [29]), the whole term $\sum_{a<c}[ed(aX) - st(aX) + 1]$ and $\sum_{c<z}[ed(zX) - st(zX) + 1]$ can be computed in $O(\log\sigma/\log\log n + 1)$ time, because characters in the alphabet are stored in leaves of the wavelet tree in alphabetic order. By a single traversal from the root of the wavelet tree to the leaf for c, we can compute the total frequency of characters smaller than c. This improvement also applies to the *forward_search* and *backward_search* operations by Lam et al. [27] in Lemma 3.3.

The space requirement can be proven by adding up all the requirement of each component. □

3.4. A new data structure for a special case of 2D range queries

We now describe the *2D range query* data structure mentioned in Section 2 for case 2. This data structure, called $\mathcal{M}$, helps to combine the results of $\mathcal{X}(\overline{T})$ and $\mathcal{Y}(F, T)$ to form the final answers for case 2. Let M be a binary $(s \times m)$-matrix. We define $M[x, y] = 1$ if $\overline{T}[x]$ is the preceding factor of the factor suffix $F[y]$. The operation $\text{query_2d}(M, [a_1, a_2], [b_1, b_2])$ reports all points in the rectangle $[a_1, a_2] \times [b_1, b_2]$ in M whose values are 1. Here, $[a_1, a_2]$ and $[b_1, b_2]$ specify consecutive rows and consecutive columns of M, respectively. The next lemma summarizes known results for general binary matrices:

Lemma 3.7. *(See [7].) Let M be a given binary matrix of size $m \times m$ with n 1s. M can be stored while supporting* $\text{query_2d}(M, [a_1, a_2], [b_1, b_2])$ *as follows:*

1. *$O(n \log^{1+\epsilon} m)$ bits and $O(\log\log m + occ)$ query time,*
2. *$O(n \log m)$ bits and $O((1 + occ)\log^{\epsilon} m)$ query time,*

where $\epsilon > 0$ is a constant and occ is the number of 1s inside the specified rectangle.

A proof of Lemma 3.7 was given by [7]. In this section, we improve the time for 2D range queries when M has a special form, namely when every column of $M[1..s, 1..m]$ contains exactly one 1. The corollary is as follows.

Corollary 3.8. *Let M be a given binary matrix of size $s \times m$, where $s \leqslant m$ and every column contains exactly one entry equal to 1. We can store M while supporting* $\text{query_2d}(M, [a_1, a_2], [b_1, b_2])$ *within the following space and time complexities:*

1. *$O(m \log s \log\log s)$ bits and $O((1 + occ)\log\log s)$ query time; or, alternatively,*
2. *$O(m \log s)$ bits and $O((1 + occ)\log^{\epsilon} s)$ query time,*

where $\epsilon > 0$ is a constant and occ is the number of 1s in the specified rectangle.

Proof. Suppose we have access to any data structure for storing general binary matrices of size $(s \times m)$ that uses $O(m \cdot \alpha(m))$ bits space and supports $\text{query_2d}(M, [a_1, a_2], [b_1, b_2])$ in $O(\beta(m) + \gamma(m)occ)$ time, where α, β, and γ are polylogarithmic functions (e.g. $\log^k m$), and $\alpha(m)$ is in $\Omega(\log m)$, and $s \leqslant m$. Then we can construct another data structure for the special case in which each column has exactly one 1 that uses $O(m \cdot \alpha(s))$ bits and with $O(\beta(s) + \gamma(s)occ)$ query time.

Let M be a binary matrix of size $(s \times m)$ in which each column has exactly one 1 and $s \leqslant m$. We partition M into $\kappa = \frac{m}{s^2}$ vertical blocks of size $s \times s^2$ arranged from left to right. For $\ell = 1, 2, \ldots, \kappa$, define $st_\ell = 1 + s^2(\ell - 1)$ and $ed_\ell = s^2\ell$, and let block ℓ be the submatrix $M[1..s, s_\ell..e_\ell]$. Any $\text{query_2d}(P, [a_1, a_2], [b_1, b_2])$ can be classified into one of two types: (i) $st_\ell \leqslant b_1 \leqslant b_2 \leqslant ed_\ell$ for some $\ell = 1, \ldots, \kappa$; and (ii) otherwise.

For (i), the query rectangle lies within a single block and we just use either data structure from Lemma 3.7 for $M[1..s, st_\ell..ed_\ell]$ for $\ell = 1, \ldots, \kappa$. Since every block has s^2 ones, the total space needed to support queries of type (i) in $O(\beta(s) + \gamma(s)occ)$ time is $O(\kappa s^2 \alpha(s^2)) = O(m\alpha(s))$ bits.

To handle queries of type (ii), we store $\binom{s}{2}$ *rank* and *select* data structures for $\binom{s}{2}$ bit vectors $B_{ij}[1..\kappa]$, defined as follows. For $1 \leqslant i \leqslant j \leqslant s$ and $1 \leqslant \ell \leqslant \kappa$, let $B_{ij}[\ell] = 1$ if and only if there exists some $M[p, q] = 1$ where $i \leqslant p \leqslant j$ and $1 + \frac{s^2}{m}(\ell - 1) \leqslant q \leqslant \frac{s^2}{m}\ell$. By Section 3.1, the total space to store the *rank* and *select* data structures is $O(\binom{s}{2}\kappa) = O(m)$ bits. Furthermore, for each $1 \leqslant \ell \leqslant \kappa$, we store a list L_ℓ of the s^2 ones in $M[1..s, st_\ell..ed_\ell]$ in sorted order according to their column numbers. We also store s pointers $Ptr_\ell[1..s]$, where $Ptr_\ell[i]$ points to the first entry in the list L_ℓ whose column number is at least i. All lists L_ℓ and Ptr_ℓ can be stored in $O(s^2\kappa \log s) = O(m \log s)$ bits.

Using $B_{ij}[\ell]$, L_ℓ, and Ptr_ℓ, we answer $\text{query_2d}(M, [a_1, a_2], [b_1, b_2])$ as follows. Let $st_{i_{min}}$ and $ed_{i_{max}}$ be the smallest st_i and the biggest ed_i such that both of them lie in the interval $b_1..b_2$. Then the answer to the query equals the union of:

(1) query_2d$(M, [a_1, a_2], [b_1, st_{i_{min}} - 1])$;
(2) query_2d$(M, [a_1, a_2], [st_{i_{min}}, ed_{i_{max}}])$; and
(3) query_2d$(M, [a_1, a_2], [ed_{i_{max}}, b_2])$.

Now, (1) and (3) can be computed in $O(\beta(s) + \gamma(s)occ)$ time by querying in inside blocks (case (i) using data structure in Lemma 3.7). For (2), we use the *rank* and *select* data structure for $B_{a_1 a_2}$ to find all entries $B_{a_1 a_2}[j] = 1$ for $i_{min} \leqslant j \leqslant i_{max}$, and for each such j, we report all points in L_j within $Ptr_j[a_1]$ and $Ptr_j[a_2 + 1]$. The running time is $O(1 + occ)$ time.

In conclusion, we can build a data structure of size $O(m\alpha(s))$ bits that supports the operation query_2d$(M, [a_1, a_2], [b_1, b_2])$ on M in $O(\beta(s) + \gamma(s)occ)$ time. Combining this result and Lemma 3.7, we have Corollary 3.8. □

4. The data structure $\mathcal{I}(T)$ for case 1

Recall from Section 2 that the array $T[1..s]$ stores the s distinct factors of R that occur in the factorizations of $\mathcal{S}$ in lexicographical order. Here, we define a data structure named $\mathcal{I}(T)$ and apply it to locate all occurrences of a query pattern P that lie entirely inside single factors in $T[1..s]$ (case 1 in Section 2). The main result of this section is summarized in the following theorem:

Theorem 4.1. *The data structure $\mathcal{I}(T)$ uses $2n + o(n) + O(s \log n)$ bits. Given the suffix range $st..ed$ of a query pattern P in SA_R, it reports all occurrences of P inside factors stored in $T[1..s]$ using $O(occ_1(\log^{\epsilon} n + \log \sigma))$ time, where occ_1 is the number of answers.*

A naive solution is to concatenate all the factors in $T[1..s]$ and then build a suffix tree or an FM-index, but the space used by such an approach would be proportional to the total size of $\mathcal{S}$. Instead, we formulate the problem as a variant of an interval cover problem. Each factor will be represented as an interval on the reference sequence R. The pattern P is first searched in the reference R, then the factors that cover the locations that P occurs at are reported.

Note that we cannot simply enumerate all the occurrences of P in R to find the covering factors. This is because we assume that the reference R may be independent of the sequences $\mathcal{S}$, and there may be occurrences of P in R but not in $\mathcal{S}$ (we call these occurrences *false positives*). The number of false positives of P in R can be $O(n)$. If we enumerate them, the search time cannot be bounded by ℓ and occ_1.

To avoid checking the false positive occurrences of the pattern, we impose an order on the occurrences of P in R. Each location in R is implicitly annotated with the length of the longest factor that covers over it. The searching algorithm prioritizes the occurrences of P in R with longer covers. It stops when the longest possible cover factor is shorter than the pattern length. In this way, we can ensure that the false positive occurrences are not enumerated.

We need the following definitions: for each $i \in \{1, 2, \ldots, s\}$, define sp_i and ep_i as the starting and ending positions of the factor $T[i]$ inside the reference string R, i.e., $T[i] = R[sp_i..ep_i]$. We say that any factor $T[i]$ *covers a position* p if $sp_i \leqslant p \leqslant ep_i$. Also, factor $T[i]$ is to the *left of* factor $T[j]$ if either: (1) $sp_i < sp_j$; or (2) $sp_i = sp_j$ and $ep_i < ep_j$. Let $G[1..s]$ be an array of indices such that $G[i] = j$ if $T[j]$ is the i-th leftmost factor. To be able to convert between indices, we define $I_s[j] = sp_{G[i]}$ and $I_e[j] = ep_{G[i]}$. Note that $I_s[1]$ is the starting position of the leftmost factor and that the values of $I_s[1..s]$ are non-decreasing.

Next, for every $p \in \{1, 2, \ldots, n\}$, define $D[p] = \max_{j=1..s}\{I_e[j] - p + 1\colon\ I_s[j] \leqslant p\}$. Intuitively, $D[p]$ measures the distance from position p to the rightmost ending position of all factors that cover p. We have the following observation. Let $\{p_i\}$ be the set of positions of the occurrences of pattern P in R. For any such p_j, if $D[p_j] > \ell$ (where ℓ is the pattern length), the occurrence p_j of P is covered by at least one factor. (In other words, position p_j is a true positive occurrence of pattern P.) Therefore, if the $\{p_i\}$ is sorted by the values of $D[p_i]$, the true positive and false positive occurrences can be separated easily.

However, since the occurrences of P in R are already sorted by the order in the suffix array SA_R, we need some additional conceptual structures to remember the $D[p_i]$-order. Let $D'[1..n]$ be an array such that $D'[p] = D[SA_R[p]]$. (For an example, see Fig. 7(a).) $D'[p]$ tells us the length of the longest interval whose starting position equals $SA_R[p]$. Hence, D and D' can be used to filter all false positive occurrences according to the next lemma:

Lemma 4.2. *For any index p and length a, there exists a factor $T[j]$ that covers all positions from $SA_R[p]$ to $(SA_R[p] + \ell - 1)$ in R if and only if $D'[p] \geqslant \ell$.*

Proof. (Necessary condition). If $T[j]$ covers $SA_R[p]$, then $D[SA_R[p]] \geqslant |T[j]|$. We also have that $T[j]$ covers $(SA_R[p] + a - 1)$; therefore, $|T[j]| \geqslant a$. That means $D[SA_R[p]] \geqslant a$. By the definition of D', we have $D'[p] \geqslant a$.

(Sufficient condition). Follows directly from the definitions of D and D'. □

Now, we describe the new data structure $\mathcal{I}(T)$. It consists of:

- The array $G[1..s]$, using $s \log n$ bits;
- A successor data structure (see Section 3) for I_s, using $s \log n + o(n)$ bits;
- A range maximum data structure (see Section 3) for I_e, using $2s + o(s)$ bits; and
- A range maximum data structure for D', using $2n + o(n)$ bits.

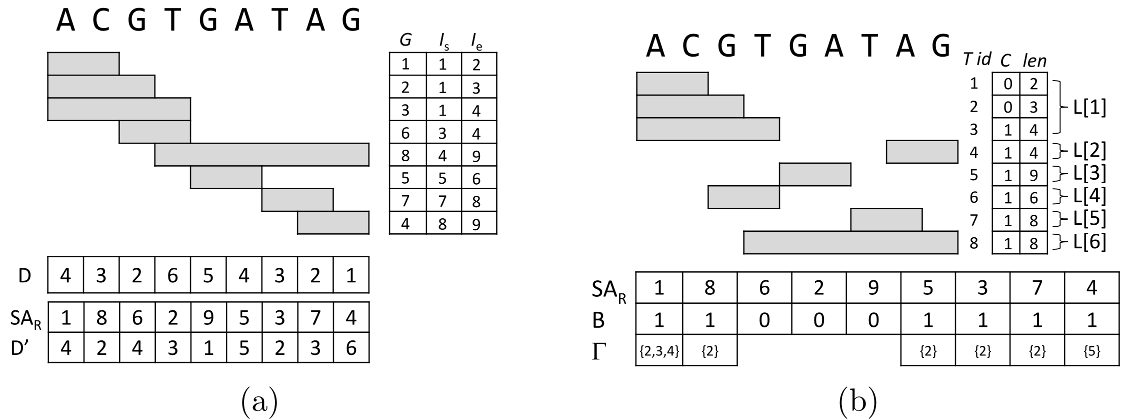

Fig. 7. (a) The factors (displayed as grey bars) from the example in Fig. 2 listed in left-to-right order, and the arrays G, I_s, I_e, D, and D' that define the data structure $\mathcal{I}(T)$ in Section 4. (b) The same factors ordered lexicographically from top to bottom, and the arrays B, C, and Γ that define the data structure $\mathcal{X}(\overline{T})$ in Section 5.

Algorithm Search_Pattern(st, ed)

Input: The data structure $\mathcal{I}(T)$, the FM-index of R and the suffix range $st..ed$ of the pattern P in SA_R.
Output: Every factor $T[j]$ in which P occurs.

1: Compute $q = \max_\text{index}_{D'}(st, ed)$
2: **if** $D'[q] \geqslant \ell$ **then**
3: Report all factors that cover $SA_R[q]..(SA_R[q] + \ell - 1)$ using Lemma 4.3
4: Search_Pattern$(st, q-1)$
5: Search_Pattern$(q+1, ed)$
6: **end if**

Fig. 8. Algorithm for computing all occurrences of P in $T[1..s]$.

Note that we do not explicitly store the arrays $D[1..n]$, $D'[1..n]$, $I_s[1..s]$, and $I_e[1..s]$. Lemma 4.3 shows how to recover the values of $D[p]$ and $D'[p]$ for any position $p \in \{1, 2, \ldots, n\}$ from the data structure $\mathcal{I}(T)$. Also, $I_s[i]$ and $I_e[i]$ can be computed in $O(1)$ time given $G[i]$ and T.

Lemma 4.3. *Given the data structures $\mathcal{I}(T)$ and the FM-index of R, for any positions p and q in R, we can*:

(i) *Compute $D[p]$ in $O(1)$ time*;
(ii) *Compute $D'[p]$ in $O(\log_\sigma^\epsilon n + \log\sigma)$ time*; *and*
(iii) *Report all factors that cover positions $p..q$ in $O(1 + occ)$ time.*

Proof. For (i), using the successor data structure for I_s, we can identify the maximum y such that $I_s[y] \leqslant p$ in $O(1)$ time. Using the range maximum data structure for I_e, we can identify an index v such that $I_e[v] = \max_{j \leqslant y} I_e[j]$ in $O(1)$ time. Then, $D[p] = I_e[v] - p + 1$. For (ii), $SA_R[p]$ can be computed in $O(\log^\epsilon n + \log\sigma)$ time by Lemma 3.1 in Section 3, so $D'[p] = D[SA_R[p]]$ can be computed in the same time.

For (iii), it is obvious that all the factors that cover both p and q need to start at a position less than or equal to p. Among them, the factors ending at q or to the right of q are those that need to be reported. Formally, the set of answers is $\{T[G[i]]\colon I_s[i] \leqslant p \text{ and } q \leqslant I_e[i]\}$.

This is the 2-sided range query problem. We first find the maximum index y such that $I_s[y] \leqslant p$. Since I_s is non-decreasing, the problem becomes reporting every value i such that $i \leqslant y$ and $q \leqslant I_e[i]$. This problem can be handled by the maximum data structure (Section 3.1) for I_e. $\square$

Based on $\mathcal{I}(T)$ and the suffix range for the query pattern P, Algorithm Search_Pattern in Fig. 8 finds all occurrences of P in factors from $T[1..s]$. Basically, it checks the occurrences $\{p_i\}$ of P in R based on the order of $D[p_i]$ from bigger to smaller, and stops when $D[p_i] < \ell$.

In Fig. 8, the value p_i is implicitly represented by $SA_R[q]$, i.e., $p_i = SA_R[q]$. Let $st..ed$ be the suffix range of P in SA_R. In line 1, the algorithm finds an index q from the range $st \leqslant q \leqslant ed$, such that $D[SA_R[q]]$ has the biggest value. The condition $D'[q] \geqslant |P|$, in line 2, guarantees that $SA_R[q]$ and $SA_R[q] + \ell - 1$ are covered by at least one factor (where ℓ is the length of the pattern). Since $st \leqslant q \leqslant ed$, it holds that $R[SA_R[q]..(SA_R[q] + \ell - 1)]$ is an occurrence of P in R. Then, the line 3 of

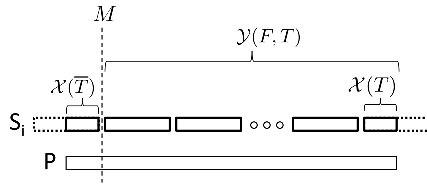

Fig. 9. Data structures used in case 2.

the algorithm reports every $T[j]$ that contains P by using Lemma 4.3. Finally, the algorithm recursively finds smaller values from its sub-ranges in line 4 and line 5.

5. The data structure $\mathcal{X}(T)$ and $\mathcal{X}(\overline{T})$ for case 2

We now turn our attention to case 2 in Section 2 (see Fig. 4(b)). This section gives the details of two symmetry data structures $\mathcal{X}(T)$ and $\mathcal{X}(\overline{T})$. For any given pattern P, $\mathcal{X}(T)$ ($\mathcal{X}(\overline{T})$) locates every occurrence of a suffix (prefix) of P that equals a prefix (suffix) of a factor of $\mathcal{S}$. (See Fig. 9.) Data structure $\mathcal{X}(\overline{T})$ is used to find the left part of the pattern in our searching algorithm outlined in Section 2.2. Data structure $\mathcal{X}(T)$ is used as a component in the data structure $\mathcal{Y}(F,T)$ to find the right part in Section 6. To simplify the presentation, we only describe $\mathcal{X}(T)$ below.

Our solution considers every non-empty suffix of P as a separate query pattern for $\mathcal{X}(T)$. For each suffix Q, we assume that Q is specified by the corresponding suffix range $st_Q..ed_Q$ in the suffix array SA_R for the reference string R, along with the length of Q. Since $T[1..s]$ stores all distinct factors S_{ij} in lexicographically sorted order, all occurrences of Q in $\mathcal{S}$ can be represented as a range $p..q$ in T such that Q is a prefix of every element in $T[p],\dots,T[q]$. The theorem and corollary below summarize the data structures $\mathcal{X}(T)$ and $\mathcal{X}(\overline{T})$.

Theorem 5.1. *The data structure $\mathcal{X}(T)$ uses $O(s\log n)+o(n)$ bits. For any suffix range $st..ed$ in SA_R of a query pattern P, it can report the maximal range $p..q$ such that P is a prefix of all $T[j]$, where $p\leqslant j\leqslant q$, in $O(\log\log n)$ time.*

Corollary 5.2. *The data structure $\mathcal{X}(\overline{T})$ uses $O(s\log n)+o(n)$ bits. For any suffix range $st..ed$ in $SA_{\overline{R}}$ of a query pattern P, it can report the maximal range $p..q$ such that $\overline{P}$ is a prefix of all $\overline{T}[j]$, where $p\leqslant j\leqslant q$, in $O(\log\log n)$ time.*

A simple solution for this problem is to build a trie of all the factors of $\mathcal{S}$. However, such a data structure requires too much space. In this section, we observe a mapping between the lexicographically sorted order of the factors (stored in the array $T[1..s]$) and the suffix array of reference sequence in Lemma 5.3. Based on this mapping, to find if one pattern is a prefix of any factor, we search for the pattern in the reference suffix array SA_R, and then calculate the mapping using Lemma 5.4 in $O(\log\log n)$ time to extract all factors.

To start, we need some efficient way to check if the query pattern P is a prefix of any specified factor $T[j]$. Since the factor $T[j]$ is a substring of R, let $st_j..ed_j$ denote the corresponding suffix range of $T[j]$ in SA_R. The next lemma says how their suffix ranges are related:

Lemma 5.3. *Suppose $st_P..ed_P$ is the suffix range of P in SA_R. P is a prefix of $T[j]$ if and only if either:* (1) $st_P<st_j\leqslant ed_P$; *or* (2) $st_P=st_j$ *and* $|T[j]|\geqslant|P|$.

Proof. We use the following property of the suffix array: Given a suffix array SA_R, consider two strings x and y such that $|x|<|y|$. Let st_x and ed_x be the suffix range of x in SA_R. Let st_y and ed_y be the suffix range of y. If x is prefix of y, then $st_x\leqslant st_y\leqslant ed_y\leqslant ed_x$. Otherwise, (st_x,ed_x) and (st_y,ed_y) are disjoint.

($\rightarrow$) By the property of the suffix array, if P is prefix of $T[j]$, then the suffix range of $T[j]$ is inside the suffix range of P in SA_R. That is $st_P\leqslant st_j\leqslant ed_j\leqslant ed_P$. In addition, since P is prefix of $T[j]$, we have $|P|\leqslant|T[j]|$. That means condition (1) or (2) is correct.

($\leftarrow$) If condition (1) is true, i.e., $st_P<st_j\leqslant ed_P$, then $ed_j\leqslant ed_P$. (Otherwise it will violate the property of the suffix array.) Since P is equals the share prefixes of all $R[SA_R[st_P]..n]\dots R[SA_R[ed_P]..n]$ and $T[j]$ is the share prefix of $R[SA_R[st_j]..n]\dots R[SA_R[ed_j]..n]$. Since the range of $T[j]$ is strictly inside the range of $T[j]$, the length of P is strictly less than the length of $T[j]$. Therefore, P is a proper prefix of $T[j]$.

If condition (2) is true, we have $st_P=st_j$. Thus, either P is prefix of $T[j]$ or $T[j]$ is prefix of P. However, we also have $|T[j]|\geqslant P$; therefore, P is a prefix of $T[j]$. □

For every $i=1,\dots,n$, define $\Gamma(i)=\{|T[j]|{:}\ st_j=i$ and $st_j..ed_j$ is the suffix range of $T[j]$ in $SA_R\}$. In other words, $\Gamma(i)$ is the set of lengths of factors whose suffix ranges start at i in SA_R. We use $\Gamma(i)$ to map a suffix range in SA_R to a range of factors in T according to:

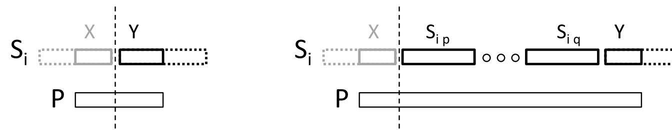

Fig. 10. Two sub-cases.

Lemma 5.4. *Suppose $st_P..ed_P$ is the suffix range of P in SA_R. Then, $p..q$ is the range in $T[1..s]$ such that P is a prefix of all $T[j]$ where $p \leqslant j \leqslant q$, where $p = 1 + \sum_{i=1}^{st_P-1} |\Gamma(i)| + |\{x \in \Gamma(st_P): x < |P|\}|$ and $q = \sum_{i=1}^{ed_P} |\Gamma(i)|$.*

Proof. By the definition of $\Gamma(i)$, for every $T[j]$ such that $1 + \sum_{i=1}^{st_P-1} |\Gamma(i)| \leqslant j \leqslant \sum_{i=1}^{ed_P} |\Gamma(i)|$, we have $st_P \leqslant st_j \leqslant ed_P$. If $st_P < st_j \leqslant ed_P$, condition (1) in Lemma 5.3 holds. Otherwise, $st_P = st_j$. However, $p = 1 + \sum_{i=1}^{st_P-1} |\Gamma(i)| + |\{x \in \Gamma(st_P): x < |P|\}|$. All the $T[j]$'s with length less than $|P|$ are not included; therefore, condition (2) in Lemma 5.3 holds. □

Now, we present the data-structure $\mathcal{X}(T)$ based on Lemma 5.4. First, let $B[1..n]$ be a bit vector such that $B[i] = 1$ if $\Gamma(i)$ is non-empty, and $B[i] = 0$ otherwise. Next, suppose $\Gamma(i)$ is the r-th non-empty set, and let $L[r]$ be a y-fast trie [40] for $\Gamma(i)$ (see Section 3). Let $C[1..s]$ be a bit vector such that $C[\sum_{i=1}^{r} |\Gamma(i)|] = 1$, and 0 otherwise. See Fig. 7(b). The data structure $\mathcal{X}(T)$ consists of three parts: (i) The *rank* data structure for the bit vector $B[1..n]$ ($s \log n + o(n)$ bits); (ii) The *select* data structure for the bit vector $C[1..s]$ ($s \log n + o(n)$ bits); and (iii) The y-fast trie data structure $L[r]$ for $\Gamma(i)$ if $\Gamma(i)$ is the r-th non-empty set ($O(s \log n)$ bits). In total, $\mathcal{X}(T)$ requires $O(s \log n) + o(n)$ bits.

Note that, for any ℓ, we have

$$\sum_{i=1}^{\ell} |\Gamma(i)| = \text{select}_C\big(\text{rank}_B(\ell)\big)$$

$$\big|\{x \in \Gamma(\ell): x < c\}\big| = \text{successor_index}\big(L\big[rank_B(\ell)\big], c\big)$$

Using $\mathcal{X}(T)$, they can be computed in $O(\log \log n)$ time. Hence, the values of p and q in Lemma 5.4 can be computed in $O(\log \log n)$ time. Theorem 5.1 follows.

6. The data structure $\mathcal{Y}(F, T)$ for case 2

This section outlines our solution for finding the occurrences for the right part of the pattern. For each suffix $P[i..\ell]$ for $1 \leqslant i \leqslant \ell$ of the pattern P, we need to check if $P[i..\ell]$ is the prefix of some factor suffix in $\mathcal{S}$ (see Fig. 4 in Section 2.2).

Solving this problem is not too complicated. Since any factor suffix has a unique RLZ factorization; given one pattern P', we can factorize the pattern using the reference, i.e., $RLZ(P'|R)$; then, match all the factorizations generated from the pattern with those sequences of $\mathcal{S}$. However, in this problem, all the suffixes $P[i..\ell]$ needs to be matched against $\mathcal{S}$. If we treat each suffix as an independent query pattern, it would take $O(\ell^2)$ time to answer all the queries. In this section, we present our approach to reuse the factorization and matching information between the suffixes to speed up the whole process. Briefly, each suffix of P is represented by a suffix range in the factor suffix array F. We factorize the suffixes of the pattern and match them with the database sequences in one run from right to left using dynamic programming.

To be precise, we build a data structure $\mathcal{Y}(F, T)$ which for any pattern P of length ℓ can compute the range of $P[i..\ell]$ in F for all $1 \leqslant i \leqslant \ell$, i.e., the range $st..ed$ in F where $P[i..\ell]$ is a prefix of $F[st], \ldots, F[ed]$. Let $Q[i]$ denote the range for each i. The following theorem summarizes the main result:

Theorem 6.1. *The data structure $\mathcal{Y}(F, T)$ uses $O(n) + (2 + 1/\epsilon)nH_k(R) + o(n \log \sigma) + O(m \log n)$ bits. It can find all suffix ranges of F that match some suffix of a query pattern P of length ℓ in $O(\ell(\log \sigma / \log \log n + \log \log n))$ time.*

First, there are two sub-cases for our pattern in this section (see Fig. 10): (1) the whole suffix $P[i..\ell]$ is a prefix of some factors; and (2) suffix $P[i..\ell]$ contains at least one factor inside and a tail which is a prefix of some factors. Solving the first sub-case is straightforward (since we can use data structure $\mathcal{X}(T)$ in Section 5). The second sub-case can be simplified based on the observation that the matched factors are unique. The properties of the two sub-cases are summarized in the following lemma:

Lemma 6.2. *For any $F[i']$, define the* head of $F[i']$ *to be the first factor of $F[i']$. For any $1 \leqslant i \leqslant \ell$, if $P[i..\ell]$ is prefix of $F[st], \ldots, F[ed]$, then P and $st..ed$ satisfy either one of the following properties:*

- (1) *$P[i..\ell]$ is the prefix of the heads of all factor suffixes $F[st], \ldots, F[ed]$.*
- (2) *The heads of all $F[st], \ldots, F[ed]$ are prefix of $P[i..\ell]$. In fact, these heads are the same; and equal $P[i..j]$ where j is the biggest index such that $P[i..j]$ is a factor of $\mathcal{S}$.*

```
1: Let r_R and r_R̄ be suffix ranges of the empty string ε in SA_R and SA_R̄.
2: j = 1
3: for i = 1 to |P| do
4:    while j ⩽ |P| and the last forward search succeeded do
5:       r_R, r_R̄ = forward_search(r_R, r_R̄, P[j])
6:       j = j + 1
7:    end while
8:    if r_R is a factor according to X(T) then let A[i] = the factor found by X(T)
9:    else let A[i] = nil
10:   r_R, r_R̄ = delete_back(r_R, r_R̄)
11: end for
```

Fig. 11. Algorithm to fill in the array $A[1..|P|]$.

Proof. Denote $P[i..\ell]$ by P_i for short. Assume P_i is the prefix of two factor suffix $F[x]$ and $F[y]$. Let $X = T[x']$ and $Y = T[y']$ are the head of $F[x]$ and $F[y]$ respectively. Because x and y are symmetrical, without loss of generality, we just consider two cases $|X| = |Y|$ and $|X| < |Y|$. Assume $|X| < |Y|$, we have the following sub-cases: (a) X is prefix of Y and Y is prefix of P_i. (b) X is prefix of P_i and P_i is prefix of Y. (c) P_i is prefix of both X and Y.

We will prove that sub-cases (a) and (b) cannot be true. Therefore, only sub-case (c) or $|X| = |Y|$ happens which leads to only cases (1) and (2) of the lemma.

In both sub-cases (a) and (b), factor X is a prefix of Y. Let c be the character in position $(|X| + 1)$ of $F[x]$. Due to the maximal property of the encoding in Section 2.1, $X \cdot c$ does not occurs in the reference R. But, Y must have some occurrence in R; therefore, the character at position $(|X| + 1)$ of $F[x]$ is different from the character at the same position of $F[y]$. However, P_i matches the position $(|X| + 1)$ of both $F[x]$ and $F[y]$ in these sub-cases, and therefore, it is a contradiction. □

Lemma 6.2 gives an important property of maximally factored suffixes. Namely, if a pattern matches the prefixes of two factor suffixes, the list of factors in the two prefixes are identical, except for the last factor whose prefix matches a suffix of the pattern. In other words, the factorization of $P[i..\ell]$ only depends on the factorization of one other suffix $P[i + a..\ell]$ (for some value a that can be computed). From this observation, we obtain the following.

Let $\mathbb{S}$ be the concatenation of the factorizations of all strings in $\mathcal{S}$, and let $\mathcal{B}$ be a general FM-index of $\mathbb{S}$ (Section 3) that supports $backward_search_{\mathbb{S}}(T[i], (st, ed))$. The array $Q[i]$ can be computed as follows. Define $A[i] = P[i..j]$, where j is the largest index such that $P[i..j]$ is a factor of $\mathcal{S}$, if one exists, and *nil* otherwise. Let $Y[i]$ be the range $st..ed$ in F such that $P[i..\ell]$ is the prefix of all the heads of factor suffixes $F[st]..F[ed]$, if one exists, and *nil* otherwise.

Informally, each entry $Y[i]$ store the result of the sub-case (1). Each entry $A[i]$ of array A stores the trace of a possible factorization for suffix $P[i..\ell]$. Then, array $Q[1..\ell]$ can be computed by dynamic programming based on the following equation:

$$Q[i] = \begin{cases} Y[i] & \text{if } Y[i] \neq \text{nil} \\ \text{backward_search}_{\mathbb{S}}(A[i], Q[i + |A[i]|]) & \text{if } Y[i] = \text{nil} \,\&\, A[i] \neq \text{nil} \\ \text{nil} & \text{otherwise} \end{cases} \quad (1)$$

By Eq. (1), $Q[1..\ell]$ can be computed in three steps:

(a) compute $A[i]$ for $i = 1$ to ℓ;
(b) compute $Y[i]$ for $i = \ell$ to 1; and
(c) compute $Q[i]$ for $i = \ell$ to 1.

Next, we present the data structure $\mathcal{Y}(F, T)$ and discuss steps (a)–(c). The data structure $\mathcal{Y}(F, T)$ consists of:

- The bi-directional BWT (see Section 3.3).
- The data structure $\mathcal{X}(T)$ (see Section 5).
- The *select* data structure for a bit-vector $V[1..m]$, defined by $V[i] = 1$ if the head of $F[i]$ differs from the head of $F[i + 1]$, and $V[i] = 0$ otherwise.
- The general FM-index $\mathcal{B}$ of $\mathbb{S}$.

First, we discuss step (a). Fig. 11 gives the algorithm to compute $A[1..\ell]$. Lemma 6.3 presents the correctness of the time complexity of the algorithm.

Lemma 6.3. *We can compute all $A[1..\ell]$ in $O(\ell(\log\sigma / \log\log n + \log\log n))$ time.*

Proof. We apply the bi-directional FM-index (see Section 3.3) to compute $A[1..\ell]$, as shown in Fig. 11:

The inner loop (lines 4–7) of the algorithm extends the search sequence to the maximal length. The outer loop (lines 3–11) assigns a value to $A[i]$ and deletes the first character to move to the next position. To check any factor in

Suf. id	Seg. suffix	V	BWT
$F[1]$	1 7	1	6
$F[2]$	2	1	8
$F[3]$	3	1	6
$F[4]$	4 5	1	\$
$F[5]$	5 6 1 7	1	\$
$F[6]$	5	0	4
$F[7]$	6 1 7	1	5
$F[8]$	6 3	0	\$
$F[9]$	7	1	1
$F[10]$	8 2	1	\$

(a)

		$F[1]$	$F[2]$	$F[3]$	$F[4]$	$F[5]$	$F[6]$	$F[7]$	$F[8]$	$F[9]$	$F[10]$
\$	\$				1	1			1		1
$\overline{T}[1]$	$T[5]$							1			
$\overline{T}[2]$	$T[7]$										
$\overline{T}[3]$	$T[1]$									1	
$\overline{T}[4]$	$T[4]$						1				
$\overline{T}[5]$	$T[8]$		1								
$\overline{T}[6]$	$T[2]$										
$\overline{T}[7]$	$T[6]$	1		1							
$\overline{T}[8]$	$T[3]$										

(b)

Fig. 12. (a) The array $F[1..m]$ consists of the factor suffixes $S_{ip}S_{i(p+1)}\dots S_{ic_i}$, encoded as indices of $T[1..s]$. Also shown in the table is a bit vector V and BWT-values, defined in Section 6. (b) For each factor suffix $F[j]$, column j in M indicates which of the factors that precede $F[j]$ in $\mathcal{S}$. To search for the pattern $P = \text{AGTA}$, we need to do two 2D range queries in M: one with $st = 1$, $ed = 2$, $st' = 7$, $ed' = 8$ since A is a suffix of $T[5]$ and $T[7]$ (i.e., a prefix in $\overline{T}[1..2]$) and GTA is a prefix in $F[7..8]$, and another one with $st = 4$, $ed = 4$, $st' = 9$, $ed' = 9$ since AG is a suffix of $T[4]$ (i.e., a prefix in $\overline{T}[4]$) and TA is a prefix in $F[9]$.

$\mathcal{X}(T)$ takes $O(\log\log n)$ time. The alphabet is of constant size, so the time for every forward_search and delete_back operation is $O(\log\sigma/\log\log n)$. Thus, each $A[i]$ is obtained in $O(\log\sigma/\log\log n + \log\log n)$ time. □

In step (b), we compute $Y[1..\ell]$ in two phases. The first phase computes another array $Y'[1..\ell]$, defined as follows: $Y'[i]$ is the range $st'..ed'$ in T such that $P[i..\ell]$ is the prefix of $T[st'], \dots, T[ed']$. By using the $\mathcal{X}(T)$ data structure from Section 5, we can obtain $Y'[1..\ell]$. Then, given $Y'[1..\ell]$, the second phase computes $Y[1..\ell]$ with the *select* data structure for V as follows: $Y[i] = (\text{select}_V(st-1)+1, \text{select}_V(ed))$, where $(st, ed) = Y'[i]$. Finally, in step (c), we apply Eq. (1) to compute $Q[1..\ell]$. The total running time is therefore $O(\ell(\log\sigma/\log\log n + \log\log n))$.

The data structure $\mathcal{X}(T)$ uses $O(s\log n) = O(m\log n)$ bits. The bi-directional BWT uses $(2 + 1/\epsilon)nH_k(R) + O(n)$ bits. The general FM-index $\mathcal{B}$ requires $O(m\log s) = O(m\log n)$ bits. The *select* data structure on bit-vector V is implemented using $O(m)$ bits. Thus, Theorem 6.1 follows.

7. Decoding the occurrence locations

Recall that given strings $\mathcal{S} = \{S_1, S_2, \dots S_t\}$, we decompose each S_i into factors. The substring from the start of a factor to the end of the string is called factor suffix. One factor may occur at multiple locations of the set of strings $\mathcal{S}$, but every factor suffix has a unique location in $\mathcal{S}$. All the distinct factors are represented in the array $T[1..s]$. The sorted order of the factor suffixes is represented in the array $F[1..m]$.

The result of case 1 of our algorithm is a set of factors such that P is a substring of them. Since each factor in this set can have multiple locations in $\mathcal{S}$, the first problem reports, for an index p of T, all the locations in $\mathcal{S}$ that factor $T[p]$ occurs at.

The result of case 2 is a set of factor suffixes represented in F such that a suffix of P is the prefix of these factor suffixes. The second problem reports, for an index p of F, the unique location in $\mathcal{S}$ that the factor suffix $F[p]$ occurs at. We design a pipeline with 3 phases to resolve cases 1 and 2.

- Phase (I): Given an index p of T, return a set of indices $\{p'\}$ such that $T[p]$ equals the first factor of each $F[p']$.
- Phase (II) computes relative locations in $\mathcal{S}$ for a factor suffix in F:
  Given an index p of F, return i, j such that $F[p]$ starts at S_{ij} in $\mathcal{S}$.
- Phase (III) converts the relative locations in $\mathcal{S}$ to the exact location in $\mathcal{S}$:
  Given i, j, return $1 + \sum_{q=1}^{j-1} |S_{iq}|$, i.e., the starting location of S_{ij} in the input string S_i.

To obtain the results for case 1, we apply all 3 phases. For case 2, we only apply phases (II) and (III).

Phase (I) can be done using the $\mathcal{Y}(F,T)$ data structure in $O(1+occ)$ time. Phase (II) can be done by decoding the general FM-index with $\mathcal{Y}(F,T)$ in $O(1+occ\cdot\log m/\log s)$ time.

Phase (III) is described next. The idea is to compute the position of S_{ij} in the string that is the concatenation of $S_1,\dots,S_t$ and then convert it to the position in S_i. Let $L[1..s]$ be an array storing the lengths of all factors in the order of occurrences in the concatenated string, that is, the length of factor S_{ij} is stored in entry $L[\sum_{i'=1}^{i-1} c_{i'}+j]$. Let $C[0..s]$ be a bit vector where $C[0]$ is set to 1, and $C[\sum_{i'=1}^{i} c_{i'}]$ are set to 1 for all $i=1,\dots,N$ where $c_{i'}$ is the number of factors in $S_{i'}$. (Thus, C encodes the indices in L of heads of factors.)

To implement phase (III), we store: the prefix sum data structure for L and the *select* data structure for C. The location of S_{ij} in S_i is obtained as follows. First, compute $s=\mathrm{select}_C(i)$. Then, the value of $1+\sum_{q=1}^{j-1}|S_{iq}|$ is given by $1+\mathrm{prefix_sum}_L(s+j-1)-\mathrm{prefix_sum}_L(s)$.

Lemma 7.1. *Phase (III) runs in $O(occ\cdot\log\log n)$ time and uses $O(m\log n)$ bits.*

Proof. The array L has m elements and the sum of all of them is at most mn. Based on [11], the space for the prefix sum data structure of L is $O(m\log(mn/m))+O(m)=O(m\log n)$ bits. Because the length of C is at most m, the *select* data structure for C uses at most $O(m)$ bits. Therefore, the total size of this data structure is $O(m\log n)$ bits.

The $\mathrm{prefix_sum}_L$ operation in L takes $O(\log\log n)$ time, and the select_C operation in C takes $O(1)$ time. □

Acknowledgements

J.J., K.S., and W.K.S. were supported in part by The Hakubi Project at Kyoto University, KAKENHI 23240002, and the MOE's AcRF Tier 2 funding R-252-000-444-112, respectively.

References

[1] The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing, Nature 467 (7319) (2010) 1061–1073.
[2] A. Apostolico, S. Lonardi, Compression of biological sequences by greedy off-line textual substitution, in: DCC, 2000, pp. 143–152.
[3] D. Arroyuelo, G. Navarro, K. Sadakane, Reducing the space requirement of LZ-index, in: CPM, in: LNCS, vol. 4009, 2006, pp. 318–329.
[4] P. Bille, G.M. Landau, R. Raman, K. Sadakane, S.R. Satti, O. Weimann, Random access to grammar-compressed strings, in: SODA, 2011, pp. 373–389.
[5] P. Bose, M. He, A. Maheshwari, P. Morin, Succinct orthogonal range search structures on a grid with applications to text indexing, in: WADS, in: LNCS, vol. 5664, 2009, pp. 98–109.
[6] M.D. Cao, T.I. Dix, L. Allison, C. Mears, A simple statistical algorithm for biological sequence compression, in: DCC, 2007, pp. 43–52.
[7] T.M. Chan, K.G. Larsen, M. Pătraşcu, Orthogonal range searching on the RAM, revisited, in: SoCG, 2011, pp. 1–10.
[8] X. Chen, S. Kwong, M. Li, A compression algorithm for DNA sequences and its applications in genome comparison, in: RECOMB, 2000, p. 107.
[9] S. Christley, Y. Lu, C. Li, X. Xie, Human genomes as email attachments, Bioinformatics 25 (2) (2009) 274–275.
[10] F. Claude, G. Navarro, Self-indexed text compression using straight-line programs, in: MFCS, in: LNCS, vol. 5734, 2009, pp. 235–246.
[11] O. Delpratt, N. Rahman, R. Raman, Compressed prefix sums, in: SOFSEM, 2007.
[12] P. Ferragina, G. Manzini, Compression boosting in optimal linear time using the Burrows–Wheeler Transform, in: SODA, 2004, pp. 655–663.
[13] P. Ferragina, G. Manzini, Indexing compressed text, Journal of the ACM 52 (4) (2005) 552–581.
[14] J. Fischer, Wee LCP, Information Processing Letters 110 (2010) 317–320.
[15] J. Fischer, Combined data structure for previous- and next-smaller-values, Theoretical Computer Science 412 (2011) 2451–2456.
[16] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: ESCAPE, in: LNCS, vol. 4614, 2007, pp. 459–470.
[17] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, S.J. Puglisi, A faster grammar-based self-index, in: LATA, in: LNCS, vol. 7183, Springer, 2012, pp. 240–251.
[18] A. Golynski, J.I. Munro, S.S. Rao, Rank/select operations on large alphabets: a tool for text indexing, in: SODA, 2006, pp. 368–373.
[19] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, in: SODA, SIAM, 2003, pp. 841–850.
[20] S. Grumbach, F. Tahi, Compression of DNA sequences, in: DCC, 1993, pp. 340–350.
[21] S. Huang, T.W. Lam, W.-K. Sung, S.-L. Tam, S.-M. Yiu, Indexing similar DNA sequences, in: AAIM, in: LNCS, vol. 6124, 2010, pp. 180–190.
[22] S. Kreft, G. Navarro, LZ77-like compression with fast random access, in: DCC, 2010, pp. 239–248.
[23] S. Kreft, G. Navarro, Self-indexing based on LZ77, in: CPM, in: LNCS, vol. 6661, 2011, pp. 41–54.
[24] S. Kuruppu, B. Beresford-Smith, T. Conway, J. Zobel, Repetition-based compression of large DNA datasets, 2009, Poster at RECOMB.
[25] S. Kuruppu, S.J. Puglisi, J. Zobel, Relative Lempel–Ziv compression of genomes for large-scale storage and retrieval, in: SPIRE, in: LNCS, vol. 6393, 2010, pp. 201–206.
[26] S. Kuruppu, S.J. Puglisi, J. Zobel, Reference sequence construction for relative compression of genomes, in: SPIRE, in: LNCS, vol. 7024, 2011, pp. 420–425.
[27] T.W. Lam, R. Li, A. Tam, S. Wong, E. Wu, S.M. Yiu, High throughput short read alignment via bi-directional BWT, in: BIBM, IEEE, 2009, pp. 31–36.
[28] N.J. Larsson, A. Moffat, Offline dictionary-based compression, in: DCC, 1999, pp. 296–305.
[29] V. Mäkinen, G. Navarro, Implicit compression boosting with applications to self-indexing, in: SPIRE, in: LNCS, vol. 4726, 2007, pp. 229–241.
[30] V. Mäkinen, G. Navarro, J. Sirén, N. Välimäki, Storage and retrieval of highly repetitive sequence collections, Journal of Computational Biology 17 (3) (2010) 281–308.
[31] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: SODA, 2002, pp. 657–666.
[32] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Computing Surveys 39 (1) (2007).
[33] M. Pătraşcu, Succincter, in: FOCS, 2008, pp. 305–313.
[34] E. Rivals, J.-P. Delahaye, M. Dauchet, O. Delgrange, A guaranteed compression scheme for repetitive DNA sequences, in: DCC, 1996, p. 453.
[35] L.M.S. Russo, A.L. Oliveira, A compressed self-index using a Ziv–Lempel dictionary, in: SPIRE, in: LNCS, vol. 4209, 2006, pp. 163–180.
[36] W. Rytter, Application of Lempel–Ziv factorization to the approximation of grammar-based compression, Theoretical Computer Science 302 (2003) 211–222.

[37] K. Sadakane, Compressed suffix trees with full functionality, Theory of Computing Systems 41 (2007) 589–607.
[38] K. Schneeberger, J. Hagmann, S. Ossowski, N. Warthmann, S. Gesing, O. Kohlbacher, D. Weigel, Simultaneous alignment of short reads against multiple genomes, Genome Biology 10 (2009) 1–12.
[39] J. Sirén, N. Välimäki, V. Mäkinen, G. Navarro, Run-length compressed indexes are superior for highly repetitive sequence collections, in: SPIRE, in: LNCS, vol. 5280, 2008, pp. 164–175.
[40] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(N)$, Information Processing Letters 17 (2) (1983) 81–84.
[41] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343.