Fast Relative Lempel-Ziv Self-index for Similar Sequences

Huy Hoang Do[1], Jesper Jansson[2], Kunihiko Sadakane[3], and Wing-Kin Sung[1]

[1] National University of Singapore, COM 1, 13 Computing Drive, Singapore 117417
{hoang,ksung}@comp.nus.edu.sg
[2] Ochanomizu University, 2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan
Jesper.Jansson@ocha.ac.jp
[3] National Institute of Informatics, 2-1-2 Hitotsubashi, Tokyo 101-8430, Japan
sada@nii.ac.jp

Abstract. Recent advances in biotechnology and web technology are 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 as follows: Given a (large) set S of strings, 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 that supports fast pattern searching.

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. 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 continually 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 similar texts may allow people to search archived versions of Wikipedia documents quickly.

To compress a single string S of length n, methods that are guaranteed to achieve the empirical k-order entropy $nH_k(S)$ are often used. However, this entropy measurement may not be a good bound for repetitive texts whose repeats are longer than k. For example, the entropy for storing the text SS (where $|S| > k$) is greater than $2nH_k(S)$. On the other hand, one can easily encode the text in $nH_k(S)+O(1)$ space. Thus, there are methods which achieve the empirical k-order entropy, yet perform poorly for repetitive texts [24]. As a consequence,

J. Snoeyink, P. Lu, K. Su, and L. Wang (Eds.): FAW-AAIM 2012, LNCS 7285, pp. 291–302, 2012.

compression methods have been designed for specific types of repetitive texts in biology. Christley *et al.* [5] compressed DNA sequences with respect to a reference sequence, and BioCompress [12] and XM [3] are other repetitive compressors designed specifically for DNA. Alternative approaches include methods based on grammar compression and LZ77 compression [26] for general repetitive texts. These methods can store repetitive texts compactly, but do not allow random access to the compressed text directly. Some previous work has addressed this issue. Kreft and Navarro [14] provided the first efficient random access operations for the LZ77 method. Bille *et al.* [2] 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.

However, one important operation on large text databases is *indexing*, in which the occurrences of an arbitrary pattern inside the stored text need to be located quickly. Some specialized data structures for indexing repetitive texts have appeared recently. In a pioneering paper of Mäkinen *et al.* [19], 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.* [13] 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 [15] 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 the maximal depth of the phrases $h \leq m$. In another line of research, Claude and Navarro [6] 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 in their grammar, and the resulting query time is quadratic ($O((\ell^2 + h(\ell + occ)) \log r)$). Some results for LZ78 compression and FM-index were given in [8]; on the negative side, these methods require $O(NH_k)$ bits in the worst case, and they may not be good enough to index a repetitive text in practice [24] or in theory [23]. In summary, existing indexes for a set of similar strings either require: (1) a lot of space, (2) that the indexed text has some special structure, or (3) quadratic query time.

New Results: Our main contribution is a data structure that stores a set $\mathcal{S}$ of strings and a reference string R in asymptotically almost optimal space, while providing almost linear-time pattern searching queries, as follows:

Theorem 1. *Given a reference string R of length n over an alphabet Σ of size σ and a set of strings $\mathcal{S} = \{S_1, \ldots, S_t\}$ over Σ, let m be the smallest possible number of factors to represent $\mathcal{S}$ with respect to R. All exact occurrences of any query pattern P of length ℓ can be reported within either of the following space and time complexities:*

(a) $2nH_k(R) + 5.55n + O(m \log n)$ *bits and* $O(\ell(\log \sigma + \frac{\log n}{\log \log n}) + occ \cdot (\log n + \frac{\log m}{\log n}))$ *query time; or, alternatively,*

(b) $2nH_k(R) + 5.55n + O(m \log n \log \log n)$ *bits and* $O(\ell(\log \sigma + \log \log n) + occ \cdot (\log n + \frac{\log m}{\log n}))$ *query time,*

where occ is the number of occurrences of P.

In this paper, we assume that the reference R is given. In case no such R is available, we can apply the method of Kuruppu *et al.* [17] to find a suitable one.

We compress each sequence in $\mathcal{S}$ using a new variant of the relative Lempel-Ziv (RLZ) compression scheme from [16]. RLZ 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. 1 for an example. Experiments on large scale genomic data in [16] have shown that this method yields good compression ratios for repetitive texts even when parts of the sequence are rearranged.

Our pattern searching algorithm follows a "standard" strategy for strings decomposed into factors. It 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. 2.) For case 2, the pattern is usually divided into two parts: left and right. The left part ends at one end of a factor, and the right part begins at the start of another factor. Each part is searched independently and then joined together by an appropriate 2D range query data structure.

Although using the same basic strategy, the currently existing methods require quadratic time for pattern searching due to the fact that they need to re-search the left part and right part for each possible boundary between two factors. In our approach, we deploy multiple tricks to search all the possible left parts and right parts in only one run, and combine the results effectively. Notably, for the left part search (Section 4), we observe a mapping between the suffix array of the factors and the reference sequence, and then simulate the search in factors using the data structure for the reference. To implement the right part search (Sections 5-6), we use dynamic programming and backward search to utilize the results of previous searches. Note that these techniques are only valid because of the properties provided by the RLZ compression scheme. According to Theorem 1 above, the total space used by our data structure which supports fast pattern searching queries is only $O(nH_k(R) + n + m \log n)$ bits, where m is the minimal number of encoding factors. This is very close to the minimal space required by any RLZ variant, which is $\Omega(nH_k(R) + m \log n)$ bits.

We remark that recently, Gagie *et al.* [10] 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, their method always uses more space than the method in

our Theorem 1 (a) above but is faster, and is incomparable to the method in Theorem 1 (b).) Also note that in their method, the reference must be equal to one of the sequences in $\mathcal{S}$ since 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 structure. Section 3 describes some auxiliary data structures from the literature used in our construction. Section 3 also presents a new data structure for answering a restricted type of 2D range queries. Sections 4 – 6 describe further technical details of our main data structure. Due to space limitations, the focus will be on describing the construction of our new data structure; correctness proofs and additional intermediate explanations will be available in the full version of the paper.

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 [16], defined next: 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 $LZ(S|R)$, 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. $LZ(S|R)$ can be computed in linear time [16]. 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 1. *$LZ(S|R)$ represents S using the smallest possible number of factors.*

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)

Fig. 1. (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.

We will need some more definitions. Let m be the minimum number of factors required to represent all of $\mathcal{S}$ with respect to R. Denote the Lempel-Ziv factorization of each S_i relative to R by $S_i = S_{i1}S_{i2}\dots S_{ic_i}$ for $i = 1, 2, \dots, t$. 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. 1 (b)). Define $m = \sum_{i=1}^{t} c_i$. Note that $s \leq \min\{n^2, m\}$. 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. 1 (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 $S_{ip}S_{i(p+1)}\dots S_{ic_i}$, and is called a *factor suffix* from here on. For any string x, $\overline{x}$ denotes its reverse. Let $\overline{T}[1..s]$ be an array of all reversed distinct factors $\overline{S}_{ij}$ sorted lexicographically.

2.2 Pattern Searching

To find the occurrences of a query pattern P in $\mathcal{S}$, we follow the basic strategy briefly mentioned in Section 1. Suppose P is a query pattern. Each occurrence of P in $S_1, \dots, S_t$ belongs to one of the following two main cases; see Fig. 2:

- **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, as explained next, to report all occurrences of P in $\mathcal{S}$ according to case 1 and case 2 separately. Summing all their complexities together yields Theorem 1 above. 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] We use the data structure $\mathcal{I}(T)$ defined in Section 4 to find all occurrences of P in $O(|P| + occ_1 \log n)$ time (Theorem 2). The data structure is of size $2n + o(n) + O(s \log n)$ bits. See Section 4 for details.

Case 2: [P is not a substring of a single factor] As illustrated in Fig. 2, in this case, every occurrence of P can be divided into two parts: the *left part* (the suffix of a factor), and the *right part* (starting with a factor). We use three additional

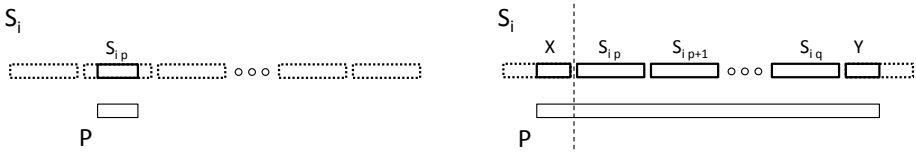

Fig. 2. When P occurs in string S_i, there are two possibilities, referred to as case 1 and case 2. In case 1 (shown on the left), P is contained inside a single factor S_{ip}. In case 2 (shown on the right), P stretches across two or more factors $S_{i(p-1)}, S_{ip}, \dots, S_{i(q+1)}$.

data structures: (i) $\mathcal{X}(\overline{T})$ to find the left parts; (ii) $\mathcal{Y}(F,T)$ to find the right parts by dynamic programming; and (iii) $\mathcal{M}$ to report the correct combinations of the left parts and right parts. The technical details of $\mathcal{X}(\overline{T})$, $\mathcal{Y}(F,T)$, and $\mathcal{M}$ are given in Sections 5, 6, and 3, respectively. Their usage is summarized as follows:

(i) $\mathcal{X}(\overline{T})$ 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 $O(|P|\log\log n)$ 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 $\overline{T}[st_j],\ldots,\overline{T}[ed_j]$.

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

(iii) Encode all combinations of X and $S_{ip}\ldots S_{iq}Y$ that are adjacent in some $S_i\in\mathcal{S}$ as follows: Define M to be a binary $(s\times m)$-matrix where $M[x,y]=1$ iff $\overline{T}[x]$ is the preceding factor of the suffix $F[y]$, i.e., $F[y]=S_{ip}S_{i(p+1)}\ldots S_{ic_i}$ 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. 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. 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, and if $\mathcal{M}$ is of size $O(m\log s)$ bits, the query takes $O(\log s/\log\log s+occ\cdot\log^{\epsilon}s)$ time.

As a final step, we decode all occurrences of case 1 and 2 to find their actual locations in $\mathcal{S}$. A simple array of $m\log n$ bits is used to store sampled occurrences and an extra $O(\log m/\log n)$ time for reporting each occurrence is required. (Due to space constraints, the details are deferred to the full version of this paper.)

3 Some Useful Auxiliary Data Structures

***Rank* and *Select* and Integer Data Structures:** 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 $\{k\in i..j : A[k]\geq v\}$. 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 $A[i]\geq v$. The data structure for this operation is called the *y-fast trie* [25]. The complexities of some existing data structures supporting the above operations are listed in the next table.

Operation	Extra space	Time	Reference	Remark
$\text{rank}_B(i)$, $\text{select}_B(i)$	$\log \binom{n}{k} + o(n)$	$O(1)$	[22]	
$\text{max\_index}_A(i,j)$	$2n + o(n)$	$O(1)$	[9]	
$\text{range\_query}_A(i,j,v)$	$O(n \log m)$	$O(1 + occ)$	[20], p. 660	
$\text{successor\_index}_A(v)$	$O(n \log m)$	$O(\log \log m)$	[25]	A is sorted

The Suffix Array and BWT 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 all suffixes of R sorted lexicographically. Any substring x of R can be represented by a pair of indices (st, ed), called a *suffix range* or SA_R*-range*. For any given string P specified by its suffix range (st, ed) in SA_R, a *BWT (Burrows-Wheeler transform) 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 $\text{backward\_search}_R(c, (st, ed))$, where c is any character, returns the suffix range in SA_R of the string cP.

Given any string R of length n over an alphabet of size σ, [7,18] showed how to construct a BWT index of R that uses $nH_k(R) + o(n)$ bits and supports backward_search_R in $O(\log \sigma)$ time and Ψ_R in $O(1)$ time. Using an additional $n + o(n)$ bits, lookup_R can be supported in $O(\log n)$ time.

A *general BWT index* is a BWT index extended to alphabets of unbounded size. The next lemma is our simple extension of the normal BWT to the general BWT case, obtained by applying the result from [11] and some additional arrays:

Lemma 2. *Given any string S of length m over an alphabet of size s, there exists a general BWT 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.*

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. Using existing results by Chan *et al.* [4] and Nekrich [21], we can improve the time complexity for 2D range queries for the special case when each column of M contains exactly one 1. We obtain:

Lemma 3. *Let M be a given binary matrix of size $s \times m$, where $s \leq m$ 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 $O(\log s / \log \log s + occ \cdot \log^{\epsilon} s)$ query time,*

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

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 2. *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 $O(occ_1 \log n)$ 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 an interval cover problem. 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 $sp_i \leq p \leq ep_i$. 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 $D[p] = \max_{j=1..s}\{I_e[j] - p + 1 : I_s[j] \leq p\}$. Intuitively, $D[p]$ measures the distance from position p to the rightmost ending position of all factors that cover p. Let $D'[1..n]$ be an array such that $D'[p] = D[SA_R[p]]$. (For an example, see Fig. 3 (a).) $D'[p]$ tells us the length of the longest interval whose starting position equals $SA_R[p]$. Hence, we can check if a substring of R is covered by at least one factor according to the next lemma:

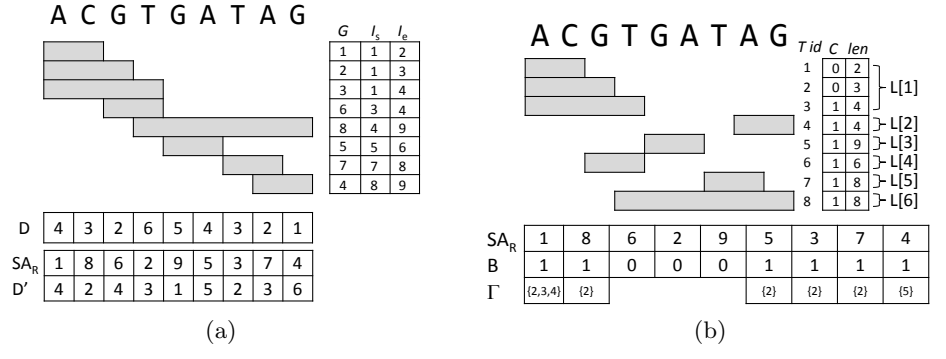

Fig. 3. (a) The factors (displayed as grey bars) from the example in Fig. 1 listed in left-to-right order, and the arrays G, I_s, I_e, D, and D' that define the data structure $\mathcal{I}(T)$ in Section 4. (b) The same factors ordered lexicographically from top to bottom, and the arrays B, C, and Γ that define the data structure $\mathcal{X}(\overline{T})$ in Section 5.

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

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

Fig. 4. Algorithm for computing all occurrences of P in $T[1..s]$

Lemma 4. *For any index p and length ℓ, there exists a factor $T[j]$ that covers positions $SA_R[p]..(SA_R[p] + \ell - 1)$ in R if and only if $D'[p] \geq \ell$.*

Now, we describe the new data structure $\mathcal{I}(T)$. It consists of: (i) The array $G[1..s]$, using $s \log n$ bits; (ii) A successor data structure (see Section 3) for I_s, using $s \log n + o(n)$ bits; (iii) A range maximum data structure (see Section 3) for I_e, using $2s + o(s)$ bits; and (iv) A range maximum data structure for D', using $2n + o(n)$ bits. 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 5 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 information about the factors.

Lemma 5. *Given two positions p and q in R, we can: (i) Compute $D[p]$ in $O(1)$ time and $D'[p]$ in $O(\log n)$ time; and (ii) Report all factors that cover positions $p..q$ in $O(1 + occ)$ time.*

Based on $\mathcal{I}(T)$ and the suffix range for the query pattern P, Algorithm Search_Pattern in Fig. 4 computes all occurrences of P in factors from $T[1..s]$. Let $st..ed$ be the suffix range of P in SA_R. The algorithm recursively finds every index q such that $st \leq q \leq ed$ (lines 4 and 5) and $D'[q] \geq |P|$ (lines 1 and 2). By Lemma 4, this condition guarantees that $SA_R[q]$ and $SA_R[q] + |P| - 1$ are covered by at least one factor. Since $st \leq q \leq ed$, it holds that $R[SA_R[q]..(SA_R[q] + |P| - 1)]$ is an occurrence of P in R. Then, the algorithm reports every $T[j]$ that contains P by using Lemma 5 on line 3.

5 The Data Structure $\mathcal{X}(\overline{T})$ for Case 2

We now turn our attention to case 2 in Section 2 (see Fig. 2 (b)). This section gives the details of the data structure $\mathcal{X}(\overline{T})$ which supports the following query: for any given pattern P, locate every occurrence of a prefix of P that equals a suffix X of a factor of $\mathcal{S}$.

First, note that each of the $|P| - 1$ non-empty proper prefixes of P may be considered separately as a query pattern for $\mathcal{X}(\overline{T})$. Therefore, we only consider how to locate the occurrences of the entire P as suffixes of factors. Secondly, we assume that P is specified by the corresponding suffix range $st_P..ed_P$ in the

suffix array SA_R for the reference string R, along with the length of P. Thirdly, recall that the array $T[1..s]$ stores the s distinct factors of the form $S_{ij} \in \mathcal{S}$ sorted lexicographically, and that $\overline{T}[1..s]$ stores all reversed distinct factors $\overline{S}_{ij}$ sorted lexicographically. Thus, $\mathcal{X}(\overline{T})$ will output the maximal range $p..q$ in $\overline{T}$ such that $\overline{P}$ is a prefix of every element in $\overline{T}[p], \ldots, \overline{T}[q]$. In Section 6, we will also need the symmetric data structure $\mathcal{X}(T)$ which, for any given query pattern P, outputs the maximal range $p..q$ in T such that P is a prefix of every element in $T[p], \ldots, T[q]$. To simplify the presentation, we only describe $\mathcal{X}(T)$ below.

Theorem 3. *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 $p \leq j \leq q$, in $O(\log \log n)$ time.*

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. For every $i = 1, \ldots, n$, 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:

Lemma 6. *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 $p \leq j \leq q$, where $p = 1 + \sum_{i=1}^{st_P - 1} |\Gamma(i)| + |\{x \in \Gamma(st_P) : x < |P|\}|$ and $q = \sum_{i=1}^{ed_P} |\Gamma(i)|$.*

Now, we define $\mathcal{X}(T)$ based on Lemma 6. 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 [25] for $\Gamma(i)$ (see Section 3). Let $C[1..s]$ be a bit vector such that $C\left[\sum_{i=1}^{r} |\Gamma(i)|\right] = 1$, and 0 otherwise. See Fig. 3 (b). We define $\mathcal{X}(T)$ to consist 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 $\sum_{i=1}^{\ell} |\Gamma(i)| = \text{select}_C(\text{rank}_B(\ell))$ and $|\{x \in \Gamma(\ell) : x < c\}| = \text{successor\_index}(L[rank_B(\ell)], c)$. Using $\mathcal{X}(T)$, they can be computed in $O(\log \log n)$ time. Hence, the values of p and q in Lemma 6 can be computed in $O(\log \log n)$ time. Theorem 3 follows.

6 The Data Structure $\mathcal{Y}(F, T)$ for Case 2

Our next task is: Given any pattern P, compute the range of $P[i..|P|]$ in F for $1 \leq i \leq |P|$, i.e., the range $st..ed$ in F such that $P[i..|P|]$ is a prefix of $F[st], \ldots, F[ed]$. Let $Q[i]$ denote the range for each i. This section introduces a data structure $\mathcal{Y}(F, T)$ which allows us to compute these ranges efficiently:

Theorem 4. *The data structure $\mathcal{Y}(F, T)$ uses $2.55n + 2nH_k(R) + O(m \log n)$ bits. It can find all suffix ranges of F that match some suffix of a query pattern P in $O(|P|(\log \sigma + \log \log n))$ time.*

For any $F[i]$, define the *head of* $F[i]$ to be the first factor of $F[i]$. Let $\mathbb{S}$ be the concatenation of the factor representations of all strings in $\mathcal{S}$, and let $\mathcal{B}$ be a general BWT index of $\mathbb{S}$ (see Section 3) supporting $backward\_search_{\mathbb{S}}(T[i], (st, ed))$.

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 $P[i..|P|]$ is the prefix of all the heads of factor suffixes $F[st]..F[ed]$, if one exists, and nil otherwise. Then:

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

By Equation (1), $Q[1..|P|]$ can be computed in three steps: (a) Compute $A[i]$ for $i = 1$ to $|P|$; (b) Compute $Y[i]$ for $i = |P|$ to 1; and (c) Compute $Q[i]$ for $i = |P|$ 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 BWT of R and the BWT of $\overline{R}$. Used to compute $A[1..|P|]$.
- The data structure $\mathcal{X}(T)$ (see Section 5). Used to compute $A[1..|P|]$, $Y[1..|P|]$.
- 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. Used to compute $Y[1..|P|]$.
- The general BWT index $\mathcal{B}$ of $\mathbb{S}$. Used to compute $Q[1..|P|]$.

In step (a), we compute $A[1..|P|]$ in $O(|P|(\log \sigma + \log \log n))$ time by using $\mathcal{X}(T)$ along with a bi-directional BWT index. In step (b), we compute $Y[1..|P|]$ in two phases. The first phase computes another array $Y'[1..|P|]$, defined as follows: $Y'[i]$ is the range $st'..ed'$ in T such that $P[i..|P|]$ is the prefix of $T[st'], \ldots, T[ed']$. By using the $\mathcal{X}(T)$ data structure from Section 5, we can obtain $Y'[1..|P|]$. Then, given $Y'[1..|P|]$, the second phase computes $Y[1..|P|]$ 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 Equation (1) to compute $Q[1..|P|]$.

Acknowledgments. JJ, KS, and WKS were supported in part by the Special Coordination Funds for Promoting Science and Technology (Japan), 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), 1061–1073 (2010)
2. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: SODA, pp. 373–389 (2011)
3. Cao, M.D., Dix, T.I., Allison, L., Mears, C.: A simple statistical algorithm for biological sequence compression. In: DCC, pp. 43–52 (2007)
4. Chan, T.M., Larsen, K.G., Pătraşcu, M.: Orthogonal range searching on the RAM, revisited. In: SoCG, pp. 1–10 (2011)
5. Christley, S., Lu, Y., Li, C., Xie, X.: Human genomes as email attachments. Bioinformatics 25(2), 274–275 (2009)

6. Claude, F., Navarro, G.: Self-indexed Text Compression Using Straight-Line Programs. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 235–246. Springer, Heidelberg (2009)
7. Ferragina, P., Manzini, G.: Compression boosting in optimal linear time using the Burrows-Wheeler Transform. In: SODA, pp. 655–663 (2004)
8. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM 52(4), 552–581 (2005)
9. Fischer, J., Heun, V.: A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 459–470. Springer, Heidelberg (2007)
10. Gagie, T., Gawrychowski, P., Kärkkäinen, J., Nekrich, Y., Puglisi, S.J.: A Faster Grammar-Based Self-index. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 240–251. Springer, Heidelberg (2012)
11. Golynski, A., Munro, J.I., Rao, S.S.: Rank/select operations on large alphabets: a tool for text indexing. In: SODA, pp. 368–373 (2006)
12. Grumbach, S., Tahi, F.: Compression of DNA sequences. In: DCC, pp. 340–350 (1993)
13. Huang, S., Lam, T.W., Sung, W.K., Tam, S.L., Yiu, S.M.: Indexing Similar DNA Sequences. In: Chen, B. (ed.) AAIM 2010. LNCS, vol. 6124, pp. 180–190. Springer, Heidelberg (2010)
14. Kreft, S., Navarro, G.: LZ77-like compression with fast random access. In: DCC, pp. 239–248 (2010)
15. Kreft, S., Navarro, G.: Self-indexing Based on LZ77. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 41–54. Springer, Heidelberg (2011)
16. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 201–206. Springer, Heidelberg (2010)
17. Kuruppu, S., Puglisi, S.J., Zobel, J.: Reference Sequence Construction for Relative Compression of Genomes. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 420–425. Springer, Heidelberg (2011)
18. Mäkinen, V., Navarro, G.: Implicit Compression Boosting with Applications to Self-indexing. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 229–241. Springer, Heidelberg (2007)
19. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of highly repetitive sequence collections. J. of Computational Biology 17(3), 281–308 (2010)
20. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In: SODA, pp. 657–666 (2002)
21. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Computational Geometry 42(4), 342–351 (2009)
22. Pătraşcu, M.: Succincter. In: FOCS, pp. 305–313 (2008)
23. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science 302, 211–222 (2003)
24. Sirén, J., Välimäki, N., Mäkinen, V., Navarro, G.: Run-Length Compressed Indexes Are Superior for Highly Repetitive Sequence Collections. In: Amir, A., Turpin, A., Moffat, A. (eds.) SPIRE 2008. LNCS, vol. 5280, pp. 164–175. Springer, Heidelberg (2008)
25. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. Information Processing Letters 17(2), 81–84 (1983)
26. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)