Theoretical Computer Science 710 (2018) 148–157

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

FM-index of alignment with gaps

Joong Chae Na [a], Hyunjoon Kim [b], Seunghwan Min [b], Heejin Park [c], Thierry Lecroq [d,e], Martine Léonard [d], Laurent Mouchard [d,f], Kunsoo Park [b,*]

[a] Department of Computer Science and Engineering, Sejong University, Seoul 05006, South Korea
[b] School of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea
[c] Department of Computer Science and Engineering, Hanyang University, Seoul 04763, South Korea
[d] Normandie University, UNIROUEN, UNIHAVRE, INSA Rouen, LITIS, 76000 Rouen, France
[e] Centre for Combinatorics on Words & Applications, School of Engineering & Information Technology, Murdoch University, Murdoch WA 6150, Australia
[f] Laboratoire d'Informatique de l'Ecole Polytechnique (LIX), CNRS UMR 7161, France

ARTICLE INFO

Article history:
Received 14 June 2016
Received in revised form 13 January 2017
Accepted 14 February 2017
Available online 2 March 2017

Keywords:
Indexes for similar strings
FM-indexes
Suffix arrays
Alignments
Backward search

ABSTRACT

Recently a compressed index for similar strings, called the *FM-index of alignment* (FMA), has been proposed with the functionalities of pattern search and random access. The FMA is quite efficient in space requirement and pattern search time, but it is applicable only for an alignment of strings without gaps. In this paper we propose the *FM-index of alignment with gaps*, a realistic index for similar strings, which allows gaps in their alignment. For this, we design a new version of the suffix array of alignment by using an alignment transformation and a new definition of the alignment-suffix. The new suffix array of alignment enables us to support the LF-mapping and backward search, the key functionalities of the FM-index, regardless of gap existence in the alignment. We experimentally compared our index with RLCSA due to Mäkinen et al. and related indexes GCSA due to Sirén et al. and GCSA2 due to Sirén on genome sequences from the 1000 Genomes Project. The index size of our index is smaller than those of other indexes.

1. Introduction

The *collection indexing* problem is defined as follows [24]: Given a collection of highly similar strings, build a compressed index for the collection of strings, and when a pattern is given, find all occurrences of the pattern in the given strings. Since the index is compressed, we also need a separate operation which extracts a specified substring of one of the given strings. Many indexes for the collection indexing problem have been developed such as RLCSA [23,24,32], LZ-scheme based indexes [6,8,16], compressed suffix trees [1,29], and so on [15,28]. To exploit the similarity of the given strings, most of them use classical compression schemes such as run-length encoding and Lempel–Ziv compressions [17,33]. Recently, Na et al. [25–27] took a new approach for the collection indexing problem by using an alignment of similar strings, and they proposed indexes of alignment called *suffix tree of alignment* [26], *suffix array of alignment* (SAA) [27], and *FM-index of*

* Corresponding author.
E-mail addresses: jcna@sejong.ac.kr (J.C. Na), hjkim@theory.snu.ac.kr (H. Kim), shmin@theory.snu.ac.kr (S. Min), hjpark@hanyang.ac.kr (H. Park), Thierry.Lecroq@univ-rouen.fr (T. Lecroq), Martine.Leonard@univ-rouen.fr (M. Léonard), Laurent.Mouchard@univ-rouen.fr (L. Mouchard), kpark@theory.snu.ac.kr (K. Park).

http://dx.doi.org/10.1016/j.tcs.2017.02.020
0304-3975/

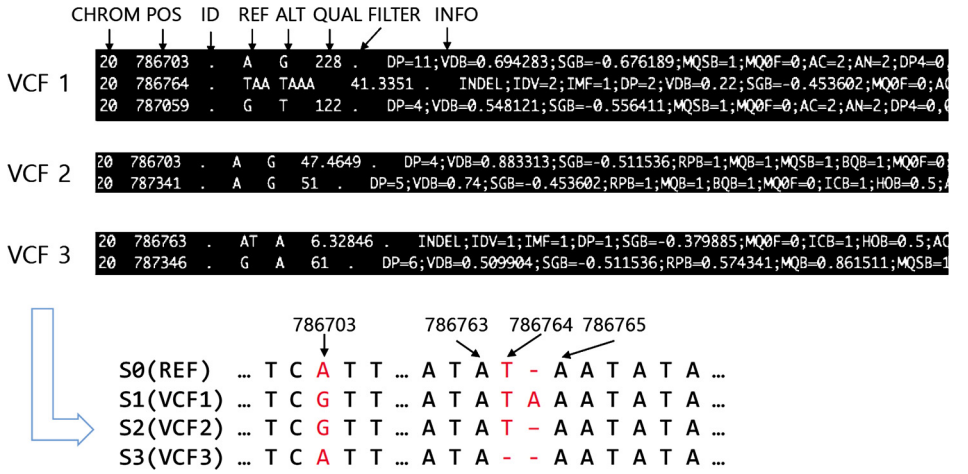


Fig. 1. Example of a VCF file.

alignment (FMA) [25]. The FMA, a compressed version of the SAA, is the most efficient among the three indexes but it is applicable only for an alignment of similar strings *without gaps*.

However, real-world data include gaps in alignments. Fig. 1 shows Variant Call Format (VCF) files created by SAMtools (Sequence Alignment/Map tools) for sequences from the 1000 Genomes Project [5]. A VCF file contains alignment information between an individual sequence and its reference sequence. Note that not only substitutions but also indels (insertions and deletions) are contained in an alignment. For example, the first line of the 'VCF 3' file in Fig. 1 says that AT at position 786763 in the reference sequence is aligned with A in the individual sequence. Thus, the FMA [25] allowing only substitutions in an alignment is an unrealistic index.

In this paper we propose a new FM-index of alignment, a realistic compressed index for similar strings, allowing indels as well as substitutions in an alignment. (We call our index the FMA with gaps and the previous version the FMA without gaps.) For this, we design a new version of the SAA by using an alignment transformation and a new definition of the suffix of an alignment (called the *alignment-suffix*). In our index, an alignment is divided into two kinds of regions, common regions and non-common regions, and gaps in a non-common region are put together into one gap in the transformed alignment. The alignment-suffix is defined for the transformed alignment but its definition is different from those defined in [25–27]. Due to the alignment transformation and the new definition of the alignment-suffix, our index supports the LF-mapping and backward search, the key functionalities of the FM-index [10–12], regardless of gap existence in the alignment.

For constructing our index, we must find common regions and non-common regions for the given strings but we do not need to find a multiple alignment for the given strings since the knowledge about positions where substitutions and indels occur are of no use in our transformed alignment. Finding common and non-common regions is much easier and simpler than finding a multiple alignment. For instance, common regions and non-common regions between an individual sequence and its reference sequence can be directly obtained from a VCF file. In the example of Fig. 1, position 786703 is a non-common region and positions 786704..786763 are a common region. Hence, based on the reference sequence, common regions and non-common regions of genome sequences can be easily created. We implemented the FMA with gaps and did experiments on 101 genome sequences from the 1000 Genomes Project. We compared our FMA with RLCSA due to Mäkinen et al. [24]. The index size of our FMA is less than one third of that of RLCSA.

The FM-index [12] is defined on a string, but it is such a powerful tool that it can be applied to many generalizations of a string: a tree [9], a graph [3,31,30], an alignment, etc. to produce compressed and searchable indexes. GCSA [31] is an FM-index on a graph, and ours is an FM-index on an alignment (i.e., they solve different problems), though there are some common ingredients in the two indexes (because they are FM-indexes). For comparison, we included GCSA and GCSA2 in our experiments, which show that the index size of FMA is smaller than those of GCSA and GCSA2. The FM-index also has been used in various ways for read alignment [14,18–22] and for finding set-maximal matches [7].

This paper is organized as follows. We first describe our FMA and search algorithm for an alignment with gaps in Section 2 and give experimental results in Section 3. In Section 4, we conclude with remarks.

2. FM-index of alignment with gaps

2.1. Alignments with gaps

Consider a multiple alignment in Fig. 2 (a) of four similar strings: $S^1 =$ $\texttt{\$cctcaaaccc\#}$, $S^2 =$ $\texttt{\$cctccaaacaa\#}$, $S^3 =$ $\texttt{\$ccttataac\_\#}$, and $S^4 =$ $\texttt{\$cct\_aacc\#}$. These strings are the same except the underlined characters and one string can be transformed into another strings by replacing, inserting or deleting underlined substrings. Formally, we are given an

pos.	1	2	3	4	5	6	7	8	9	0	1	2	3
$S^1 =$	$	c	c	t	c	-	a	-	a	a	c	c	#
$S^2 =$	$	c	c	t	c	c	a	-	a	a	c	a	#
$S^3 =$	$	c	c	t	t	-	a	t	a	a	c	-	#
$S^4 =$	$	c	c	t	-	-	-	-	a	a	c	c	#
	$\tilde{\alpha}_1^\diamond$		$\tilde{\alpha}_1^+$		Δ_1				$\tilde{\alpha}_2^\diamond$		$\tilde{\alpha}_2^+$	Δ_2	α_3

(a)

pos.	1	2	3	4	5	6	7	8	9	0	1	2
$S^1 =$	$	c	-	c	t	c	a	a	a	c	c	#
$S^2 =$	$	c	c	t	c	c	a	a	a	c	a	#
$S^3 =$	$	c	c	t	t	a	t	a	-	a	c	#
$S^4 =$	$	c	-	-	-	c	t	a	a	c	c	#
	$\tilde{\alpha}_1^\diamond$		$\tilde{\alpha}_1^+\Delta_1$					$\tilde{\alpha}_2^\diamond$	$\tilde{\alpha}_2^+\Delta_2$			α_3

(b)

Fig. 2. An example of (a) an original alignment and (b) its transformed alignment.

alignment Υ of m similar strings $S^j = \alpha_1\Delta_1^j \ldots \alpha_r\Delta_r^j\alpha_{r+1}$ $(1 \le j \le m)$ over an alphabet Σ, where α_i $(1 \le i \le r+1)$ is a common substring in all strings and Δ_i^j $(1 \le i \le r)$ is a non-common substring in string S^j. In the example above, $\alpha_1 =$ `$cct`, $\alpha_2 =$ `aac`, $\alpha_3 =$ `#`. Without loss of generality, we assume α_1 starts with `$` and α_{r+1} ends with `#` where `$` and `#` are special symbols occurring nowhere else in S^j, and each α_i is not empty.

For each common substring α_i, we define $\tilde{\alpha}_i^+$ as follows.[1]

Definition 1. The string $\tilde{\alpha}_i^+$ $(1 \le i \le r)$ is the shortest suffix of α_i occurring only once in each string S^j $(1 \le j \le m)$ and $\tilde{\alpha}_{r+1}^+$ is an empty string.

Consider $\alpha_1 =$ `$cct` in Fig. 2. Since the suffix `t` of length 1 occurs more than once in S^3 but the suffix `ct` of length 2 occurs only once in each string, $\tilde{\alpha}_1^+$ is `ct`. Similarly, $\tilde{\alpha}_2^+$ is `ac`, which is the shortest suffix of α_2 occurring only once in each string. Without loss of generality, for $2 \le i \le r+1$, $\tilde{\alpha}_i^+$ is assumed to be shorter than α_i. (If $\tilde{\alpha}_i^+$ is equal to α_i, we merge α_i with its adjacent non-common substrings Δ_{i-1}^j and Δ_i^j, and regard $\Delta_{i-1}^j\alpha_i\Delta_i^j$ as one non-common substring).

For indexing similar strings whose alignment includes gaps, we first transform the given alignment Υ into its right-justified form $\widetilde{\Upsilon}$ so that the characters in each $\tilde{\alpha}_i^+\Delta_i^j$ $(1 \le i \le r,\ 1 \le j \le m)$ are right-justified. See Fig. 2 for an example, where a gap is represented by a series of hyphens '–' (note that '–' is not a character). Hereafter, to indicate positions of characters in S^j, we use the positions in the transformed alignment $\widetilde{\Upsilon}$ and denote by $\widetilde{S}^j[i]$ the character of S^j at the ith position in $\widetilde{\Upsilon}$. If $\widetilde{S}^j[i]$ is '–', we say $\widetilde{S}^j[i]$ is empty. The positions in S^j and $\widetilde{S}^j$ can be easily converted into each other by storing gap information. Moreover, we denote the suffix of $\widetilde{S}^j$ starting at position q by suffix (j, q), e.g., the suffix $(3, 8)$ is `aac#` in Fig. 2.

An alignment of similar strings can be compactly represented by combining each common substring α_i in all strings as in [25–27]. However, the representation is not suitable for the transformed alignment $\widetilde{\Upsilon}$ because the characters in $\tilde{\alpha}_i^+$ are not aligned in $\widetilde{\Upsilon}$. Thus, we introduce another representation. Let $\tilde{\alpha}_i^\diamond$ $(1 \le i \le r+1)$ be the prefix of α_i such that $\alpha_i = \tilde{\alpha}_i^\diamond\tilde{\alpha}_i^+$. Then, we represent the transformed alignment $\widetilde{\Upsilon}$ by combining $\tilde{\alpha}_i^\diamond$ (rather than α_i): $\widetilde{\Upsilon} = \tilde{\alpha}_1^\diamond(\tilde{\alpha}_1^+\Delta_1^1/\cdots/\tilde{\alpha}_1^+\Delta_1^m)\cdots\tilde{\alpha}_r^\diamond(\tilde{\alpha}_r^+\Delta_r^1/\cdots/\tilde{\alpha}_r^+\Delta_r^m)\tilde{\alpha}_{r+1}^\diamond$. The alignment in Fig. 2 is represented as $\widetilde{\Upsilon} =$ `$c(ctca/cctca/ctcat/ct)a(acc/aca/ac/acc)#`. We denote $(\tilde{\alpha}_i^+\Delta_i^1/\cdots/\tilde{\alpha}_i^+\Delta_i^m)$ by $\tilde{\alpha}_i^+\Delta_i$ and call it a *ps-region* (partially-shared region). Also, we call $\tilde{\alpha}_i^\diamond$ a *cs-region* (completely-shared region).

2.2. Suffix array and FM-index of alignment with gaps

In our index, one or more suffixes starting at an identical position q are compactly represented by one *alignment-suffix* (for short *a-suffix*) defined as follows. We have two cases according to whether the starting position q is in a cs-region or a ps-region.

- The case when q is in a cs-region $\tilde{\alpha}_i^\diamond$ $(1 \le i \le r+1)$. Let α_i' be the suffix of $\tilde{\alpha}_i^\diamond$ starting at q. All the suffixes starting at q are represented by the a-suffix $\alpha_i'(\tilde{\alpha}_i^+\Delta_i^1/\cdots/\tilde{\alpha}_i^+\Delta_i^m)\cdots$. In the previous example, the suffixes starting at position 8 are represented by the a-suffix `a(acc/aca/ac/acc)#`.
- The case when q is in a ps-region $\tilde{\alpha}_i^+\Delta_i$ $(1 \le i \le r)$. Let δ_i^j $(1 \le j \le m)$ be the suffix of $\tilde{\alpha}_i^+\Delta_i^j$ starting at q. Then, the set of the suffixes starting at q is partitioned so that the suffixes of S^{j_1} and S^{j_2} are in the same subset if and only if $\delta_i^{j_1} = \delta_i^{j_2}$. For each subset $\{\delta_i^{j_1}\cdots\tilde{\alpha}_{r+1}^\diamond, \ldots, \delta_i^{j_k}\cdots\tilde{\alpha}_{r+1}^\diamond\}$, all the suffixes in the subset are represented by the a-suffix $(\delta_i^{j_1}/\cdots/\delta_i^{j_k})\cdots\tilde{\alpha}_{r+1}^\diamond$. For example, the set of the suffixes starting at position 9 is partitioned into two subsets $\{\widetilde{S}^1[9..12], \widetilde{S}^4[9..12]\}$ and $\{\widetilde{S}^2[9..12]\}$, and they are represented by the a-suffixes `(acc/acc)#` and `aca#`, respectively. Note that no suffix of $\widetilde{S}^3$ starts at position 9.

The suffixes represented by an a-suffix appear consecutively in the generalized suffix array of the given strings since $\tilde{\alpha}_i^+$ occurs only once in each given string. Note that a suffix of $\tilde{\alpha}_i^+$ may occur more than once in a string, and thus $\tilde{\alpha}_i^+$ does not belong to a cs-region but to a ps-region.

[1] Note that the definition of $\tilde{\alpha}_i^+$ is different from that of $\tilde{\alpha}_i^*$ in [25–27]. The $\tilde{\alpha}_i^+$ is longer than $\tilde{\alpha}_i^*$ by one.

idx	SAA		*F*	a-suffixes (cyclic shifts)	*L*	$occ(\sigma, i)$ & B_σ					
	strs	*pos*				a		c		t	
1	0	1	$	$c(ctca/ctcca/cttat/ct)a(acc /aca/ac/acc)#	#	0		0		0	
2	0	12	#	#$c(ctca/ctcca/cttat /ct)a(acc/aca/ac/acc)	a,c	1		1		0	
3	2	11	a	a#$cctccaaac	c	1		2		0	
4	1,2	7	a	(a/a)a(acc/aca)#$c(ctc/ctcc)	c	1		3		0	
5	0	8	a	a(acc/aca/ac/acc)#$c(ctca/ctcca /cttat/ct)	a,t	2		3		1	
6	3	10	a	ac#$cctatata	a	3	1	3		1	
7	2	9	a	aca#$cctccaa	⟨a⟩	3	1	3		1	
8	1,4	9	a	(acc/acc)#$ c(ctca/ct)a	⟨a⟩	3	1	3		1	
9	3	6	a	ataac#$ cctt	t	3		3		2	
10	1,3,4	11	c	(c/c/c)#$ c(ctca/cttat/ct)a(ac/a/ac)	a,c	4		4		2	
11	2	10	c	ca#$cctccaaa	a	5		4		2	
12	1,2	6	c	(ca/ca)a(acc/aca)#$c(ct/ctc)	c,t	5		5		3	
13	1,4	10	c	(cc/cc)#$ c(ctca/ct)a(a/a)	a	6		5		3	
14	2	5	c	ccaaaca#$cct	t	6		5		4	
15	0	2	c	c(ctca/ctcca/cttat/ct)a(acc /aca/ac/acc)#$	$	6		5		4	
16	4	6	c	ctaacc#$c	c	6		6	1	4	
17	1	4	c	ctcaaacc#$c	⟨c⟩	6		6	1	4	
18	2	3	c	ctccaaaca#$c	⟨c⟩	6		6	1	4	
19	3	3	c	cttataac#$c	⟨c⟩	6		6	1	4	
20	3,4	7	t	(t/t)a(ac/acc)#$c(ctta/c)	a,c	7		7		4	
21	3	5	t	tataac#$cct	t	7		7		5	
22	1	5	t	tcaaacc#$cc	c	7		8		5	
23	2	4	t	tccaaaca#$cc	c	7		9		5	
24	3	4	t	ttataac#$cc	c	7		10		5	

Fig. 3. The SAA and FMA for $\widetilde{\Upsilon}$ = $c(ctca/ctcca/cttat/ct)a(acc/aca/ac/acc)#. (Bit 0 is omitted in B_σ.)

The *suffix array of alignment (SAA)* is a lexicographically sorted array of all the a-suffixes of the transformed alignment $\widetilde{\Upsilon}$. See Fig. 3 for an example, where the string number 0 indicates the string numbers $1, \ldots, m$. We denote by $SAA[i]$ the ith entry of the SAA. Let us consider a-suffixes in the SAA as cyclic shifts (rotated alignments) as in the Burrows–Wheeler transform [4]. Then, the array $F[i]$ (resp. $L[i]$) is the set of the first (resp. last) characters of the suffixes represented by the a-suffix in $SAA[i]$. By definition of the a-suffixes, the first characters of the suffixes represented by an a-suffix are of the same value and thus $F[i]$ has one element. However, $L[i]$ may have more than one element (at most $|\Sigma|$ elements) as shown in Fig. 3. For example, $F[13] = \{\widetilde{S}^1[10], \widetilde{S}^4[10]\} = \{\mathtt{c}\}$ and $L[10] = \{\widetilde{S}^1[10], \widetilde{S}^3[10], \widetilde{S}^4[10]\} = \{\mathtt{a}, \mathtt{c}\}$. Since gaps are not considered as characters in $\widetilde{\Upsilon}$, when letting q be the position of the characters in $F[i]$, the positions of the characters in $L[i]$ may be less than $q-1$. (On the other hand, the position of the characters in $L[i]$ is always $q-1$ for an alignment without gaps when $q > 1$.) In Fig. 3, $F[17] = \widetilde{S}^1[4]$ and $L[17] = \widetilde{S}^1[2]$ because $\widetilde{S}^1[3]$ is empty.

We define the LF-mapping for the arrays L and F. Let $\mathcal{L}$ be the set of pairs of a character σ and an entry index i such that $\sigma \in L[i]$. In the example of Fig. 3, $\mathcal{L} = \{(\#, 1), (\mathtt{a}, 2), (\mathtt{c}, 2), (\mathtt{c}, 3), (\mathtt{c}, 4), (\mathtt{a}, 5), (\mathtt{t}, 5), \ldots\}$. For a pair $(\sigma, i) \in \mathcal{L}$, the *LF-mapping* $LF(\sigma, i)$ is defined as the index of $F[k]$ containing the characters corresponding to σ in $L[i]$. For example, see $L[10] = \{\widetilde{S}^1[10], \widetilde{S}^3[10], \widetilde{S}^4[10]\} = \{\mathtt{a}, \mathtt{c}\}$. Since $\mathtt{a}$ in $L[10]$ corresponds to $\widetilde{S}^3[10]$ and it is contained in $F[6]$, $LF(\mathtt{a}, 10) = 6$. Similarly, $LF(\mathtt{c}, 10) = 13$ since $\widetilde{S}^1[10]$ and $\widetilde{S}^4[10]$ (i.e., $\mathtt{c}$ in $L[10]$) are contained in $F[13]$. Note that $LF(\mathtt{c}, 10)$ is well defined since the characters in $L[10]$ whose values are $\mathtt{c}$ are all contained in an *identical* entry $F[13]$. This is always true in the transformed alignment $\widetilde{\Upsilon}$ even though gaps exist in $\widetilde{\Upsilon}$, as shown in the following lemma. (It is not true in the untransformed alignment Υ.)

Lemma 1. *For a pair $(\sigma, i) \in \mathcal{L}$, the characters in $L[i]$ whose values are σ are all contained in an identical entry of F.*

Proof. Let q be the starting position of the suffixes in $SAA[i]$, and $\widetilde{S}^{j_1}[q_1]$ and $\widetilde{S}^{j_2}[q_2]$ ($j_1 \neq j_2$) be two characters in $L[i]$ whose values are σ. Without loss of generality, we assume $q > 1$. Then, q_1 and q_2 are less than q. We have three cases according to whether $\widetilde{S}^{j_1}[q-1]$ and $\widetilde{S}^{j_2}[q-1]$ are empty.

- First, when none of $\widetilde{S}^{j_1}[q-1]$ and $\widetilde{S}^{j_2}[q-1]$ are empty (i.e., $q_1 = q_2 = q-1$), $\widetilde{S}^{j_1}[q_1]$ and $\widetilde{S}^{j_2}[q_2]$ are contained in an identical entry of F by definition of the a-suffix, which can be shown as in [25].
- Second, when both of $\widetilde{S}^{j_1}[q-1]$ and $\widetilde{S}^{j_2}[q-1]$ are empty, both $\widetilde{S}^{j_1}[q_1]$ and $\widetilde{S}^{j_2}[q_2]$ are the last character in a cs-region $\tilde{\alpha}_i^{\diamond}$ since the characters in ps-region $\tilde{\alpha}_i^{+}\Delta_i$ are right-justified in $\widetilde{\Upsilon}$. Thus, $q_1 = q_2$ and by definition of the a-suffix, the suffixes (j_1, q_1) and (j_2, q_2) are contained in an identical entry of the SAA. Hence, $\widetilde{S}^{j_1}[q_1]$ and $\widetilde{S}^{j_2}[q_2]$ are contained in an identical entry of F.
- The third case is when only one of $\widetilde{S}^{j_1}[q-1]$ and $\widetilde{S}^{j_2}[q-1]$ is empty. We show by contradiction that this case cannot happen. Without loss of generality, assume $\widetilde{S}^{j_1}[q-1]$ is empty and $\widetilde{S}^{j_2}[q-1]$ is not empty. Since $\widetilde{S}^{j_1}[q-1]$ is empty, $\widetilde{S}^{j_1}[q_1]$ is the last character in a cs-region $\tilde{\alpha}_k^{\diamond}$ and $\widetilde{S}^{j_1}[q]$ is the first character in ps-region $\tilde{\alpha}_k^{+}\Delta_k$. It means that the suffix (j_1, q) is prefixed by $\tilde{\alpha}_k^{+}$. Since both suffixes (j_1, q) and (j_2, q) are in $SAA[i]$, by definition of the a-suffix, the

suffix (j_2, q) is also prefixed by $\tilde{\alpha}_k^+$. Since $\tilde{\alpha}_k^+$ occurs only once in each string, $\widetilde{S}^{j_2}[q_2]$ is the last character in $\tilde{\alpha}_k^\diamond$ (i.e., $q_1 = q_2$) and $\widetilde{S}^{j_2}[q_2 + 1..q - 1]$ is empty. It contradicts with the assumption that $\widetilde{S}^{j_2}[q - 1]$ is not empty.

Therefore, the characters in $L[i]$ whose values are σ are all contained in an identical entry of F. □

For a character $\sigma \in \Sigma$, a pair $(\sigma, i) \in \mathcal{L}$ will be called an $\mathcal{L}_\sigma$-pair. For two $\mathcal{L}_\sigma$-pairs (σ, i) and (σ, i'), we say that (σ, i) is smaller than (σ, i') if and only if $i < i'$.

The LF-mapping $LF(\sigma, i)$ is not a one-to-one correspondence. Multiple pairs can be mapped to the same entry in F. See $L[6] = \{\widetilde{S}^3[8]\} = \{\mathtt{a}\}$, $L[7] = \{\widetilde{S}^2[8]\} = \{\mathtt{a}\}$, and $L[8] = \{\widetilde{S}^1[8], S^4[8]\} = \{\mathtt{a}\}$. Since all of them are a in $F[5]$, $LF(\mathtt{a}, 6) = LF(\mathtt{a}, 7) = LF(\mathtt{a}, 8) = 5$. Thus, we classify pairs $(\sigma, i) \in \mathcal{L}$ into two types: A pair $(\sigma, i) \in \mathcal{L}$ is an *(m:1)-type* (many-to-one mapping-type) pair if there exists another pair $(\sigma, i') \in \mathcal{L}$ such that $LF(\sigma, i) = LF(\sigma, i')$; otherwise, (σ, i) is a *(1:1)-type* (one-to-one mapping-type) pair. The following lemma shows that for a (m:1)-type pair (σ, i), the last characters of all the suffixes in $SAA[i]$ are mapped to an identical entry in F. (This lemma is necessary for our search algorithm to work correctly and it is also satisfied for the FMA without gaps [25]. However, it is not satisfied when defining our index using $\tilde{\alpha}^*$ in [25] rather than $\tilde{\alpha}^+$.)

Lemma 2. *If a pair $(\sigma, i) \in \mathcal{L}$ is of (m:1)-type, no pair (σ', i) such that $\sigma' \neq \sigma$ exists in $\mathcal{L}$, i.e., $L[i]$ has only one character σ.*

Proof. Let $k = LF(\sigma, i)$ and q_k be the starting position of the suffixes in $SAA[k]$. By definition of the a-suffix, the pair (σ, i) is of (m:1)-type only if q_k is the last position in a cs-region $\tilde{\alpha}_j^\diamond$ and the last character in $\tilde{\alpha}_j^\diamond$ is σ. Hence, all the suffixes in $SAA[i]$ are prefixed by $\tilde{\alpha}_j^+$. Since $\tilde{\alpha}_j^+$ occurs only once in each string, the preceding character of $\tilde{\alpha}_j^+$ is the last character in $\tilde{\alpha}_j^\diamond$, i.e., σ. Therefore, $L[i]$ has only one character σ. □

To handle (m:1)-type pairs in $\mathcal{L}$ efficiently, we define bit-vectors B_σ's as follows: $B_\sigma[i] = 1$ if and only if (σ, i) is in $\mathcal{L}$ and it is of (m:1)-type (see Fig. 3).

The LF-mapping can be easily computed using the array C and the function *occ* defined as follows [25].

- For $\sigma \in \Sigma$, $C[\sigma]$ is the total number of entries in F containing characters alphabetically smaller than σ. $C[|\Sigma| + 1]$ is the size of F.
- For a character $\sigma \in \Sigma$ and an entry index i in the SAA, $occ(\sigma, i)$ is the number of $\mathcal{L}_\sigma$-pairs (σ, i') such that $i' \leq i$, i.e., the number of entries in $L[1..i]$ containing the character σ. If more than one pair $(\sigma, i') \in \mathcal{L}$ are mapped to an identical entry in F, we count only the smallest $\mathcal{L}_\sigma$-pair among them. For example, consider $occ(\mathtt{a}, i)$ for $i = 6, 7, 8$. Since a's in $L[6..8]$ are all contained in $F[5]$, we only count $(\mathtt{a}, 6)$ and thus $occ(\mathtt{a}, i)$'s are the same for $i = 6, 7, 8$. In Fig. 3, uncounted characters in L are indicated by $\langle\ \rangle$.

Then, $LF(\sigma, i) = C[\sigma] + occ(\sigma, i)$. See $L[10]$ in Fig. 3, which has two $\mathcal{L}_\sigma$-pairs $(\mathtt{a}, 10)$ and $(\mathtt{c}, 10)$. We have $LF(\mathtt{a}, 10) = C[\mathtt{a}] + occ(\mathtt{a}, 10) = 2 + 4 = 6$ and $LF(\mathtt{c}, 10) = C[\mathtt{c}] + occ(\mathtt{c}, 10) = 9 + 4 = 13$.

2.3. Pattern search

Pattern search is to find all occurrences of a given pattern $P[1..p]$ in the given strings $S^1, \ldots, S^m$. Our pattern search algorithm proceeds backward using the LF-mapping with the array C and the function *occ*. It consists of at most p steps from Step p to Step 1. In Step $\ell = p, \ldots, 1$, the algorithm finds the *closed* range $(\mathsf{First}_\ell, \mathsf{Last}_\ell)$ in the SAA defined as follows:

i) First_p (resp. Last_p) is the smallest (resp. largest) index i such that $F[i] = \{P[p]\}$.
ii) For $\ell = p - 1, \ldots, 1$, First_ℓ (resp. Last_ℓ) is the LF-mapping value of the smallest (resp. largest) $\mathcal{L}_\sigma$-pair in the range $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$, where $\sigma = P[\ell]$. If there exists no $\mathcal{L}_\sigma$-pair in $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$, then we set $\mathsf{First}_\ell = \mathsf{Last}_\ell + 1$.

Then, all the suffixes prefixed by $P[\ell..p]$ are in $SAA[\mathsf{First}_\ell..\mathsf{Last}_\ell]$ and the size of the range decreases monotonically when ℓ decreases.

While the size of the range $(\mathsf{First}_\ell, \mathsf{Last}_\ell)$ is greater than one (i.e., $\mathsf{First}_\ell < \mathsf{Last}_\ell$), all the suffixes in $SAA[\mathsf{First}_\ell..\mathsf{Last}_\ell]$ are prefixed by $P[\ell..p]$. When $\mathsf{First}_\ell = \mathsf{Last}_\ell$, however, some suffixes in $SAA[\mathsf{First}_\ell]$ may not be prefixed by $P[\ell..p]$. For example, when assuming that $P = \mathtt{aaacc}$, we have $(\mathsf{First}_2, \mathsf{Last}_2) = (5, 5)$, and the suffixes $(1, 8)$ and $(4, 8)$ in $SAA[5]$ are prefixed by $\mathtt{aacc}$ but the other suffixes $(2, 8)$ and $(3, 8)$ are not. Also, we have $(\mathsf{First}_1, \mathsf{Last}_1) = (4, 4)$, and the suffix $(1, 7)$ in $SAA[4]$ is prefixed by $\mathtt{aaacc}$ but the suffix $(2, 7)$ is not. Thus, in addition to the range $(\mathsf{First}_\ell, \mathsf{Last}_\ell)$, we maintain the set Z_ℓ defined as follows:

- When $\mathsf{First}_\ell = \mathsf{Last}_\ell$, Z_ℓ is the set of the string numbers of the suffixes prefixed by $P[\ell..p]$.

Algorithm 1 BackwardSearch($P[1..p]$) ▷ using the FM-index of alignment.

```
1: Z ← {1, ..., m};                                              ▷ Set of all string numbers
2: σ ← P[p], First ← C[σ]+1, Last ← C[σ+1], ℓ ← p−1;
3: while (First ≤ Last) and Z ≠ ∅ and (ℓ ≥ 1) do
4:     σ ← P[ℓ], First' ← First, Last' ← Last;                    ▷ Previous range
5:     First ← C[σ] + occ(σ, First−1) + 1, Last ← C[σ] + occ(σ, Last);
6:     if First ≥ Last then
7:         Z_m ← {j | (j,q) ∈ SAA[i] such that First' ≤ i ≤ Last' and B_σ[i] = 1};
8:         if Z_m ≠ ∅ then
9:             First ← Last, Z ← Z ∩ Z_m;
10:        else
11:            Z_c ← {j | (j,q) ∈ SAA[First..Last]},              ▷ If First > Last, Z_c = ∅
12:            Z ← Z ∩ Z_c;
13:    ℓ ← ℓ−1;
14: for all (j,q) ∈ SAA[First..Last] do                          ▷ If First > Last, no occurrence
15:     if j ∈ Z then print "(j,q)";                             ▷ Reporting an occurrence
```

For simplicity, we define Z_ℓ to be $\{1,\ldots,m\}$ when $\mathsf{First}_\ell < \mathsf{Last}_\ell$. Then, regardless of the size of the range $(\mathsf{First}_\ell, \mathsf{Last}_\ell)$, a suffix (j,q) is prefixed by $P[\ell..p]$ if and only if $(j,q) \in SAA[\mathsf{First}_\ell..\mathsf{Last}_\ell]$ and $j \in \mathsf{Z}_\ell$.

Algorithm 1 shows the search algorithm using our index, which is the same as the code in [25]. (Since the definition of the a-suffix is different from that in [25], however, we need a correctness proof which will be given later.) The algorithm maintains the following loop invariant for a range $(\mathsf{First}, \mathsf{Last})$ and a string number set Z:

> At the end of Step $\ell = p,\ldots,1$, the range $(\mathsf{First}, \mathsf{Last}) = (\mathsf{First}_\ell, \mathsf{Last}_\ell)$ and $\mathsf{Z} = \mathsf{Z}_\ell$.

Initially (in Step p), we set $(\mathsf{First}, \mathsf{Last}) = (\mathsf{First}_p, \mathsf{Last}_p)$ and $\mathsf{Z} = \mathsf{Z}_p$ (lines 1–2). Each iteration of the while loop (lines 3–13) represents each Step $\ell = p-1,\ldots,1$. In Step $\ell = p-1,\ldots,1$, we first compute range $(\mathsf{First}, \mathsf{Last})$ using the LF-mapping of the previous range $(\mathsf{First}', \mathsf{Last}') = (\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$ and $\sigma = P[\ell]$ (lines 4–5). If the size of the range $(\mathsf{First}, \mathsf{Last})$ is more than one, then $(\mathsf{First}, \mathsf{Last}) = (\mathsf{First}_\ell, \mathsf{Last}_\ell)$ and $\mathsf{Z} = \mathsf{Z}_\ell = \{1,\ldots,m\}$. Thus, we continue to the next step (by skipping lines 6–12 and going to line 13). Otherwise (i.e., the size of $(\mathsf{First}, \mathsf{Last})$ is one or less), we compute Z_ℓ as follows (lines 6–12). Let Z_m be the set of the string numbers in $SAA[i]$'s such that $\mathsf{First}_{\ell+1} \le i \le \mathsf{Last}_{\ell+1}$ and $B_\sigma[i] = 1$, and let Z_c be the set of the string numbers in $SAA[\mathsf{Last}]$. Then, $\mathsf{Z}_\ell = \mathsf{Z}_{\ell+1} \cap Z_\mathrm{m}$ if $Z_\mathrm{m} \neq \emptyset$, and $\mathsf{Z}_\ell = \mathsf{Z}_{\ell+1} \cap Z_\mathrm{c}$, otherwise. (As in [25], lines 10–12 for Z_c can be removed by using a loose definition and a lazy update for Z_ℓ.) For example, assume $P = \mathtt{aaacc}$. In Step 2, given $(\mathsf{First}_3, \mathsf{Last}_3) = (8,8)$ and $\mathsf{Z}_3 = \{1,2,3,4\}$, we have $Z_\mathrm{m} = \{1,4\}$ and thus $\mathsf{Z}_2 = \{1,4\}$. In Step 1, given $(\mathsf{First}_2, \mathsf{Last}_2) = (5,5)$ and $\mathsf{Z}_2 = \{1,4\}$, we have $Z_\mathrm{m} = \emptyset$ and $Z_\mathrm{c} = \{1,2\}$ ($\mathsf{Last}_1 = 4$), and thus $\mathsf{Z}_1 = \{1,4\} \cap \{1,2\} = \{1\}$. After the while loop terminates, the occurrences of P are reported using the range $(\mathsf{First}, \mathsf{Last})$ and Z (lines 14–15). Since the SAA stores positions in the transformed alignment $\widetilde{\Upsilon}$, we need to convert them to the original positions in the given strings S^j, which can be easily done by using gap information.

Now we show the invariant is satisfied at the end of each step (an iteration of the while loop) by induction. Trivially, the invariant is true at the end of Step p, which is the induction basis. At the beginning of Step $\ell = p-1,\ldots,1$, by inductive hypothesis, $(\mathsf{First}, \mathsf{Last}) = (\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$. After executing line 5, $\mathsf{First} = C[\sigma] + occ(\sigma, \mathsf{First}_{\ell+1} - 1) + 1$ and $\mathsf{Last} = C[\sigma] + occ(\sigma, \mathsf{Last}_{\ell+1})$, where $\sigma = P[\ell]$. Then, the following lemmas show Algorithm 1 computes correctly $(\mathsf{First}_\ell, \mathsf{Last}_\ell)$ and Z_ℓ at the end of Step ℓ.

Lemma 3. *If* $\mathsf{First} < \mathsf{Last}$, *then* $(\mathsf{First}_\ell, \mathsf{Last}_\ell) = (\mathsf{First}, \mathsf{Last})$ *and* $\mathsf{Z}_\ell = \mathsf{Z}_{\ell+1}$.

Proof. By definition of the LF-mapping, the suffixes in $SAA[\mathsf{First}..\mathsf{Last}]$ are prefixed by $P[\ell..p]$. We show that no suffix outside $SAA[\mathsf{First}..\mathsf{Last}]$ is prefixed by $P[\ell..p]$. Suppose that a suffix prefixed by $P[\ell..p]$ is contained in an $SAA[i]$ outside $SAA[\mathsf{First}..\mathsf{Last}]$. Then, all suffixes in $SAA[i]$ are prefixed by $P[\ell..p]$. (If two suffixes in two distinct entries of the SAA are prefixed by $P[\ell..p]$, then all the suffixes in the two entries are prefixed by $P[\ell..p]$, which can be easily shown using the definition of the a-suffix.) Let (σ, k) be the smallest $\mathcal{L}_\sigma$-pair such that $LF(\sigma, k) = i$. Since the suffixes in $SAA[k]$ are prefixed by $P[\ell+1..p]$, k is included in the previous range $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$ by definition and thus its LF-mapping value i is also included in $(\mathsf{First}, \mathsf{Last})$ (note that the pair (σ, k) is always counted in the function occ). It contradicts with the assumption that i is outside the range $(\mathsf{First}, \mathsf{Last})$. Therefore, we get $(\mathsf{First}_\ell, \mathsf{Last}_\ell) = (\mathsf{First}, \mathsf{Last})$. Furthermore, since $\mathsf{First}_\ell \neq \mathsf{Last}_\ell$, $\mathsf{Z}_\ell = \mathsf{Z}_{\ell+1} = \{1,\ldots,m\}$ by definition. □

Lemma 4. *If* $\mathsf{First} \ge \mathsf{Last}$ *and* $Z_\mathrm{m} \neq \emptyset$, *then* $(\mathsf{First}_\ell, \mathsf{Last}_\ell) = (\mathsf{Last}, \mathsf{Last})$ *and* $\mathsf{Z}_\ell = \mathsf{Z}_{\ell+1} \cap Z_\mathrm{m}$.

Proof. Since $Z_\mathrm{m} \neq \emptyset$, there exist $\mathcal{L}_\sigma$-pairs of (m:1)-type in $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$. Furthermore, all of the $\mathcal{L}_\sigma$-pairs are mapped to one entry $F[\mathsf{Last}]$ of F since $\mathsf{First} \ge \mathsf{Last}$. Therefore, $(\mathsf{First}_\ell, \mathsf{Last}_\ell) = (\mathsf{Last}, \mathsf{Last})$.

Next, let us consider Z_ℓ. In this case, $\mathcal{L}_\sigma$-pairs in $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$ are all of (m:1)-type. Moreover, for every $\mathcal{L}_\sigma$-pair (σ, i) in $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$, the last characters of the suffixes in $SAA[i]$ are all σ by Lemma 2. Thus, Z_m is the set of

the string numbers of the suffixes whose last characters are σ in $SAA[\mathsf{First}_{\ell+1}..\mathsf{Last}_{\ell+1}]$. By definition, $\mathsf{Z}_{\ell+1}$ is the set of the string numbers of the suffixes prefixed by $P[\ell+1..p]$. Thus, a suffix (j,q) in $SAA[\mathsf{Last}]$ is prefixed by $\sigma P[\ell+1..p]$ $(= P[\ell..P])$ if and only if $j \in Z_{\mathrm{m}}$ and $j \in \mathsf{Z}_{\ell+1}$. Therefore, we get $\mathsf{Z}_\ell = Z_{\mathrm{m}} \cap \mathsf{Z}_{\ell+1}$. □

Lemma 5. *If* $\mathsf{First} \geq \mathsf{Last}$ *and* $Z_{\mathrm{m}} = \emptyset$, *then* $(\mathsf{First}_\ell, \mathsf{Last}_\ell) = (\mathsf{First}, \mathsf{Last})$ *and* $\mathsf{Z}_\ell = \mathsf{Z}_{\ell+1} \cap Z_{\mathrm{c}}$.

Proof. Since $Z_{\mathrm{m}} = \emptyset$, there is no $\mathcal{L}_\sigma$-pair of (m:1)-type in $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$. If $\mathsf{First} = \mathsf{Last}$, there is one $\mathcal{L}_\sigma$-pair of (1:1)-type in $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$. If $\mathsf{First} > \mathsf{Last}$, there is no $\mathcal{L}_\sigma$-pair of (1:1)-type in $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$. In both cases, $(\mathsf{First}_\ell, \mathsf{Last}_\ell) = (\mathsf{First}, \mathsf{Last})$.

Next, let us consider Z_ℓ when $\mathsf{First} = \mathsf{Last}$. Let (σ, i) be the only one $\mathcal{L}_\sigma$-pair in $(\mathsf{First}_{\ell+1}, \mathsf{Last}_{\ell+1})$. Since (σ, i) is of (1:1)-type, the set of the string numbers in $SAA[\mathsf{Last}]$ (i.e., Z_{c}) is the same as the set of the string numbers of the suffixes whose last characters are σ in $SAA[\mathsf{First}_{\ell+1}..\mathsf{Last}_{\ell+1}]$. Thus, a suffix (j,q) in $SAA[\mathsf{Last}]$ is prefixed by $\sigma P[\ell+1..p]$ $(= P[\ell..P])$ if and only if $j \in Z_{\mathrm{c}}$ and $j \in \mathsf{Z}_{\ell+1}$. Therefore, we get $\mathsf{Z}_\ell = Z_{\mathrm{c}} \cap \mathsf{Z}_{\ell+1}$. □

Therefore, we can get the following theorem.

Theorem 1. *Algorithm 1 finds correctly all the occurrences of a pattern* P.

2.4. Data structures

Our index consists of the function *occ*, the array C, the bit-vectors B_σ, and a sampled SAA. Furthermore, we store gap information for mutual conversion between positions in an original string S^j and positions in its transformed string $\widetilde{S}^j$. As described in [27], we first build the suffix array of alignment, from which we construct the FM index of alignment (i.e., *occ*, C, B_σ, and a sampled SAA).

We store the SAA using two kinds of sampling as in [25], the *regular-position sampling* and the *irregular-position sampling*. For the regular-position sampling, we sample $SAA[i]$ storing every d-th position in the transformed alignment $\widetilde{\Upsilon}$ where d is a given parameter. Then, we get an $SAA[i]$ in a sampled SAA by repeating the LF-mapping from $SAA[i]$ until a sampled entry $SAA[k]$ is encountered. In order to guarantee that the string numbers in $SAA[i]$ are the same as the ones in $SAA[k]$, we also need the following irregular sampling: an $SAA[i]$ is sampled when $L[i]$ has multiple characters or, for any $\sigma \in \Sigma$, the pair (σ, i) is of (m:1)-type. Note that such an $SAA[i]$ has different string numbers from the string number in $SAA[i']$ where $i' = LF(\sigma, i)$ for a character $\sigma \in L[i]$.

For supporting extraction (retrieval) operations, we also need a sampled inverse SAA. In the FMA without gaps [25], the regular-position sampling is enough for the inverse SAA. Due to gaps, however, we need also an irregular sampling for the inverse SAA. Suppose that a gap in a transformed string $\widetilde{S}^j$ includes a regular sampling position q. Then, we cannot sample the position q in $\widetilde{S}^j$. Let q' be the leftmost position such that $q' > q$ and no gap in $\widetilde{S}^j$ includes q'. Then, the position q', instead of q, is sampled in $\widetilde{S}^j$. For example, assuming position 4 is a regular sampling position in Fig. 2, instead of position 4, position 6 is sampled in $\widetilde{S}^4$ since $\widetilde{S}^4[3..5]$ is a gap.

3. Experiments

We compared our FM-index of alignment (FMA) with RLCSA [24], GCSA [31], and GCSA2 [30]. We used SDSL (Succinct Data Structure Library [13]) to implement the FMA and used the implementations of the other indexes distributed by the authors.[2] All experiments were conducted on a computer with Intel Xeon X5672 CPU and 32GB RAM, running the Linux debian 3.16.0-4-amd64 operating system.

The experimental data set is a reference sequence and 100 individual sequences, which are downloaded from the 1000 Genomes Project website. The reference genome is chromosome 20 of hs37d5 of length about 63 million bases and each individual sequence consists of a pair of BAM and BAI files, where a BAM file contains reads (short segments of length 90–125) of each individual and a BAI file contains the alignment of the reads. Each pair of BAM and BAI files is fed to SAMtools (Sequence Alignment/Map tools) to obtain a VCF file which stores genetic mutations such as substitutions, insertions and deletions relative to the reference genome. The individual sequences are created from the reference and the VCF files.

First, we compared the sizes of the four indexes for 31 and 101 sequences (Table 1). The FMA, RLCSA, and GCSA were created with sampling rates $d = 32$, 128, and 512, and GCSA2 was created using order-128, which means GCSA2 is a 128-mer index storing substrings of length 128 (we could not find out a parameter for sampling in the implementation of GCSA2). The table shows that the index size of FMA is smaller than those of other indexes in every case. Note that the FMA is the smallest even when only the "core" and "sampling" sizes are considered. Another merit of the FMA is that its size is little influenced by the sampling rates or the number of sequences, while the sizes of RLCSA and GCSA are substantially influenced by them.

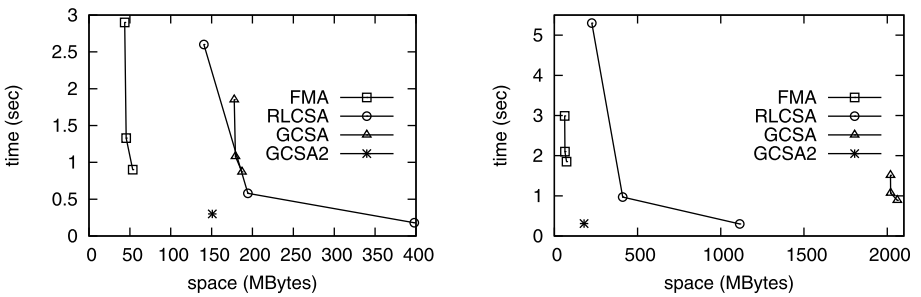
[2] GCSA2 version 0.7 was used in our experiments and an implementation of a variation [2] of RLCSA is also available.

Table 1
The index sizes (in MBytes) where "sampling" means the space for sampling, "gap" means the space for storing gap information, and "core" means the space except for sampling and gap. For GCSA, "backbone" is the space for storing the path in a graph corresponding to the reference sequence. For GCSA2, "counter" and "lcp" are the spaces for supporting counting queries and for storing lcp information, respectively.

Number of sequences		31			101		
Sampling rate		32	128	512	32	128	512
FMA	total	53.5	45.5	43.6	72.8	63.8	61.6
	core	36.8	36.8	36.8	44.1	44.1	44.1
	gap	1.6	1.6	1.6	5.7	5.7	5.7
	sampling	15.1	7.2	5.2	22.9	14.0	11.7
RLCSA	total	397.9	194.3	140.6	1114.0	410.8	225.4
	core	121.6	121.6	121.6	159.9	159.9	159.9
	sampling	276.3	72.7	19.0	954.1	250.9	65.5
GCSA	total	187.1	179.2	177.7	2060.4	2020.6	2019.3
	core	67.8	67.8	67.8	576.9	576.9	576.9
	sampling	56.0	48.2	46.7	765.7	725.9	724.6
	backbone	63.2	63.2	63.2	717.8	717.8	717.8
GCSA2	total	150.8			178.7		
	core	47.4			53.2		
	sampling	39.2			52.3		
	counter	10.5			13.4		
	lcp	53.7			59.8		

Table 2
Pattern search (location) time (in secs) for 5000 queries.

Query length	Number of sequences	31			101		
	Sampling rate	32	128	512	32	128	512
10	FMA	85.0	267.6	777.2	172.5	318.8	577.6
	RLCSA	72.5	318.6	1623.5	190.0	830.0	4887.0
	GCSA	85.9	190.5	508.5	349.4	452.2	651.5
	GCSA2	15.38			16.32		
30	FMA	1.25	3.57	9.46	2.53	4.27	7.59
	RLCSA	0.68	3.28	18.32	1.86	8.48	54.25
	GCSA	0.77	2.06	6.11	0.92	2.14	4.63
	GCSA2	0.26			0.28		
100	FMA	0.90	1.33	2.90	1.85	2.10	2.99
	RLCSA	0.18	0.58	2.60	0.30	0.97	5.30
	GCSA	0.87	1.08	1.85	0.89	1.07	1.51
	GCSA2	0.30			0.31		



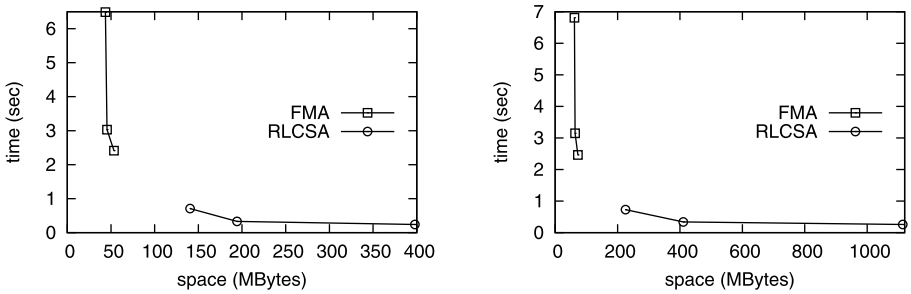
(a) 31 sequences (b) 101 sequences

Fig. 4. Total index sizes and pattern search (location) times for 5000 queries of length 100. Each index was tested with sampling rates $d = 32$, 128, and 512.

Second, we compared the running time of pattern search (location) reporting all occurrences. We performed the pattern search with patterns of lengths 10, 30, and 100 on the indexes with sampling rates $d = 32$, 128, and 512 (Table 2 and Fig. 4). GCSA2 shows the best performance in most cases but GCSA and GCSA2 report nodes of a graph but not occurrences in given sequences (note that they are indexes on a graph with different functionalities). In comparison with RLCSA, FMA is faster than RLCSA when many sequences are indexed, sampling is sparse, and/or queries are short. We also compared the

Table 3
Extraction time (in secs) for 5000 queries.

Query length	Number of sequences	31			101		
	Sampling rate	32	128	512	32	128	512
10	FMA	0.51	1.34	4.79	0.50	1.37	5.05
	RLCSA	0.06	0.15	0.71	0.06	0.16	0.64
30	FMA	1.10	1.71	5.28	1.13	1.78	5.45
	RLCSA	0.14	0.19	0.79	0.14	0.20	0.59
100	FMA	2.41	3.03	6.49	2.46	3.15	6.81
	RLCSA	0.24	0.33	0.71	0.26	0.34	0.73



(a) 31 sequences (b) 101 sequences

Fig. 5. Total index sizes and extraction times for 5000 queries of length 100. Each index was tested with sampling rates $d = 32$, 128, and 512.

extraction time (Table 3 and Fig. 5). We compared only FMA and RLCSA because we could not find out how to perform an extraction query in the implementations of GCSA and GCSA2. In all cases, RLCSA is faster than FMA.

4. Concluding remarks

We have proposed the FM-index of alignment with gaps, a realistic index for similar strings, which allows gaps in their alignment. For this, we have designed a new version of suffix array of alignment by using an alignment transformation and a new definition of the alignment-suffix. The new SAA enabled us to support the LF-mapping and backward search regardless of gap existence in alignments. Experimental results showed that our index is more space-efficient than RLCSA while its extraction time is slower than that of RLCSA. It remains as future work to do extensive experiments and analysis on various real-world data.

Acknowledgements

Joong Chae Na was supported by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under National program for Excellence in Software program (the SW oriented college support program) (R7718-16-1005) supervised by the IITP (Institute for Information & communications Technology Promotion), and by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2014R1A1A1004901). Heejin Park was supported by the research fund of Signal Intelligence Research Center supervised by Defense Acquisition Program Administration and Agency for Defense Development of Korea. Thierry Lecroq, Martine Léonard and Laurent Mouchard were supported by the French Ministry of Foreign Affairs Grant 27828RG (INDIGEN, PHC STAR 2012). Kunsoo Park was supported by the Bio & Medical Technology Development Program of the NRF funded by the Korean government, MSIP (NRF-2014M3C9A3063541).

References

[1] A. Abeliuk, G. Navarro, Compressed suffix trees for repetitive texts, in: String Processing and Information Retrieval – 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21–25, 2012, Proceedings, 2012, pp. 30–41.
[2] D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, M. Raffinot, Composite repetition-aware data structures, in: Combinatorial Pattern Matching – 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29–July 1, 2015, Proceedings, 2015, pp. 26–39.
[3] A. Bowe, T. Onodera, K. Sadakane, T. Shibuya, Succinct de Bruijn graphs, in: Algorithms in Bioinformatics – 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10–12, 2012, Proceedings, 2012, pp. 225–235.
[4] M. Burrows, D.J. Wheeler, A block-sorting lossless data compression algorithm, Technical Report 124 Digital Equipment Corporation, Paolo Alto, California, 1994.
[5] The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing, Nature 467 (7319) (2010) 1061–1073.
[6] H.H. Do, J. Jansson, K. Sadakane, W.-K. Sung, Fast relative Lempel–Ziv self-index for similar sequences, Theoret. Comput. Sci. 532 (2014) 14–30.

[7] R. Durbin, Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT), Bioinformatics 30 (9) (2014) 1266–1272.
[8] H. Ferrada, T. Gagie, T. Hirvola, S.J. Puglisi, Hybrid indexes for repetitive datasets, Philos. Trans. R. Soc. Lond. Ser. A, Math. Phys. Sci. 372 (2016) (April 2014).
[9] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and indexing labeled trees, with applications, J. ACM 57 (1) (2009).
[10] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: 41st Annual Symposium on Foundations of Computer Science, FOCS 2000, Redondo Beach, California, USA, 2000, pp. 390–398.
[11] P. Ferragina, G. Manzini, An experimental study of an opportunistic index, in: Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, Washington, DC, USA, 2001, pp. 269–278.
[12] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (4) (2005) 552–581.
[13] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: plug and play with succinct data structures, in: Experimental Algorithms – 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29–July 1, 2014, Proceedings, 2014, pp. 326–337.
[14] L. Huang, V. Popic, S. Batzoglou, Short read alignment with populations of genomes, Bioinformatics 29 (13) (2013) 361–370.
[15] S. Huang, T.W. Lam, W.K. Sung, S.L. Tam, S.M. Yiu, Indexing similar DNA sequences, in: Algorithmic Aspects in Information and Management, 6th International Conference, AAIM 2010, Weihai, China, July 19–21, 2010, Proceedings, 2010, pp. 180–190.
[16] S. Kreft, G. Navarro, On compressing and indexing repetitive sequences, Theoret. Comput. Sci. 483 (2013) 115–133.
[17] S. Kuruppu, S.J. Puglisi, J. Zobel, Relative Lempel–Ziv compression of genomes for large-scale storage and retrieval, in: String Processing and Information Retrieval – 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11–13, 2010, Proceedings, 2010, pp. 201–206.
[18] B. Langmead, S.L. Salzberg, Fast gapped-read alignment with Bowtie 2, Nat. Methods 9 (4) (2012) 357–359.
[19] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows–Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[20] B. Liu, H. Guo, M. Brudno, Y. Wang, deBGA: read alignment with de Bruijn graph-based seed and extension, Bioinformatics (2016) btw371.
[21] C.M. Liu, T. Wong, E. Wu, R. Luo, S. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, T.W. Lam, SOAP3: ultra-fast GPU-based parallel alignment tool for short reads, Bioinformatics 28 (6) (2012) 878–879.
[22] S. Maciuca, C.O. Elias, G. McVean, Z. Iqbal, A natural encoding of genetic variation in a Burrows–Wheeler transform to enable mapping and genome inference, in: Algorithms in Bioinformatics – 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22–24, 2016, Proceedings, 2016, pp. 222–233.
[23] V. Mäkinen, G. Navarro, J. Sirén, N. Välimäki, Storage and retrieval of individual genomes, in: Research in Computational Molecular Biology, 13th Annual International Conference, RECOMB 2009, Tucson, AZ, USA, May 18–21, 2009, Proceedings, 2009, pp. 121–137.
[24] V. Mäkinen, G. Navarro, J. Sirén, N. Välimäki, Storage and retrieval of highly repetitive sequence collections, J. Comput. Biol. 17 (3) (2010) 281–308.
[25] J.C. Na, H. Kim, H. Park, T. Lecroq, L. Mouchard, M. Léonard, K. Park, FM-index of alignment: a compressed index for similar strings, Theoret. Comput. Sci. 638 (2016) 159–170.
[26] J.C. Na, H. Park, M. Crochemore, J. Holub, C.S. Iliopoulos, L. Mouchard, K. Park, Suffix tree of alignment: an efficient index for similar data, in: Combinatorial Algorithms – 24th International Workshop, IWOCA 2013, Rouen, France, July 10–12, 2013, Revised Selected Papers, 2013, pp. 337–348.
[27] J.C. Na, H. Park, S. Lee, M. Hong, T. Lecroq, L. Mouchard, K. Park, Suffix array of alignment: a practical index for similar data, in: String Processing and Information Retrieval – 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7–9, 2013, Proceedings, 2013, pp. 243–254.
[28] G. Navarro, Indexing highly repetitive collections, in: Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19–21, 2012, Revised Selected Papers, 2012, pp. 274–279.
[29] G. Navarro, A. Ordóñez, Faster compressed suffix trees for repetitive collections, ACM J. Exp. Algorithmics 8 (2016) 1.
[30] J. Sirén, Indexing variation graphs, in: Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, January 17–18, 2017.
[31] J. Sirén, N. Välimäki, V. Mäkinen, Indexing graphs for path queries with applications in genome research, IEEE/ACM Trans. Comput. Biol. Bioinformatics 11 (2) (March 2014) 375–388.
[32] J. Sirén, N. Välimäki, V. Mäkinen, G. Navarro, Run-length compressed indexes are superior for highly repetitive sequence collections, in: String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10–12, 2008, Proceedings, 2008, pp. 164–175.
[33] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–343.