Efficient Direct Search on Compressed Genomic Data

Xiaochun Yang #1, Bin Wang #2, Chen Li *3, Jiaying Wang #4, Xiaohui Xie *5

# *College of Information Science and Engineering, Northeastern University, Liaoning 110819 China*
1,2 {yangxc,binwang}@mail.neu.edu.cn, 4 wangjiaying@research.neu.edu.cn

* *Department of Computer Science, University of California, Irvine, CA 92697*
3,5 {chenli,xhx}@ics.uci.edu

***Abstract*—The explosive growth in the amount of data produced by next-generation sequencing poses significant computational challenges on how to store, transmit and query these data, efficiently and accurately. A unique characteristic of the genomic sequence data is that many of them can be highly similar to each other, which has motivated the idea of compressing sequence data by storing only their differences to a reference sequence, thereby drastically cutting the storage cost. However, an unresolved question in this area is whether it is possible to perform search directly on the compressed data, and if so, how. Here we show that directly querying compressed genomic sequence data is possible and can be done efficiently. We describe a set of novel index structures and algorithms for this purpose, and present several optimization techniques to reduce the space requirement and query response time. We demonstrate the advantage of our method and compare it against existing ones through a thorough experimental study on real genomic data.**

I. Introduction

Motivation: Recent advances in next-generation sequencing technologies have made DNA sequencing orders of magnitude faster and cheaper than ever before. Massive amounts of sequence data are being generated daily. For instance, Complete Genomics, one of the next-generation sequencing companies, sequenced 700 human genomes in the third quarter of 2011 alone. A number of commercial and academic entities are now actively competing for the "Archon X prize," which sets the goal of sequencing 100 human genomes within 30 days or less at a cost of no more than $1,000 per genome. Many scientists believe that this goal can be achieved within a couple of years. The ushering of personal genomics era brings significant computational challenges on how to handle the large scale genomic data in a daily life.

Although each haploid human genome is quite large (consisting of 3 billion letters of A, C, G, and T), it has been recognized that it is much more efficient to store the differences between each genome and a reference genome, instead of the genome itself, due to the high sequence similarity between them [1]. In particular, a computer program called "DNAzip" has been demonstrated to be able to compress an entire haploid genome of 3 billion characters to a merely 4 MB [2] by utilizing this idea.

Problem: In this paper we study a problem that arises naturally in this context: How to answer queries on sequences stored in this compressed representation? Formally, consider a collection of sequences $\mathcal{S} = \{S_1, \ldots, S_n\}$, in which each S_i is very long and stored in a compressed format as follows. Let B be a reference sequence (a.k.a. base sequence). For a sequence S_i, consider a transformation $\Delta_i : B \rightarrow S_i$, which consists of a set of *edit operations* that can transform B to S_i. An edit operation can be an insertion, a deletion, or a substitution, possibly of multiple consecutive characters. The characters not specified in the edit operations in Δ_i are mapped to S_i using an identity mapping. The sequences in $\mathcal{S}$ are highly similar to each other in terms of both length and content. *How to efficiently answer queries on the sequences in $\mathcal{S}$?* In this paper, we mainly focus on pattern-search queries, each of which specifies a pattern with a length from tens to thousands, and ask for all substrings in $\mathcal{S}$ that match the pattern exactly or approximately.

As an example, consider a hospital setting, where a doctor wants to find out which genomes of his/her patients contain a particular disease allele. A naive solution is to first uncompress the compressed genomes on the fly, and then do a search on the reconstructed genomes. However, this approach has significant disadvantages because uncompressing the genomes both takes time and requires additional storage space, neither of which would be desirable in a typical office setting. Moreover, an online algorithm based on each decompressed data sequence will certainly have a lower performance than an offline algorithm that can take advantage of index structures of the sequences. Our goal here is to develop methods that can do query search directly on compressed sequences.

This paper takes on the challenge of designing efficient search algorithms on compressed genomic sequences represented as a reference sequence and transformations from the reference sequence to the original genomic sequences. We begin by presenting an algorithm for exact pattern search on multiple compressed sequences, utilizing a q-gram based index on the reference sequence and a binary tree storing the differences between each sequence and the reference one (Section III). We then continue to address the following questions:

1) Do all matching grams in the reference sequence need to be checked? We give an analysis in Section IV to show that only a small subset of grams in the reference sequence need to be considered to find the answers.

We study how to improve query performance by storing additional information in the inverted index. We further describe an algorithm for selecting a minimal number of grams to efficiently compute answers.

2) The space of gram-based index can be large, especially when the length of the reference sequence is long. Can we reduce the index size without compromising the correctness? In Section V, we propose two techniques that can significantly reduce the index size while still guaranteeing both correctness and high efficiency.
3) How to efficiently support approximate pattern search with an edit-distance threshold τ by extending the techniques for exact queries? In SectionVII we develop a technique that partitions a pattern P into $\tau + 1$ disjoint pieces and computes candidate substrings of each data sequence based on the exact matches of each piece. In order to avoid verifying many candidates, we propose a novel filtering method to reduce the number of candidates to verify.

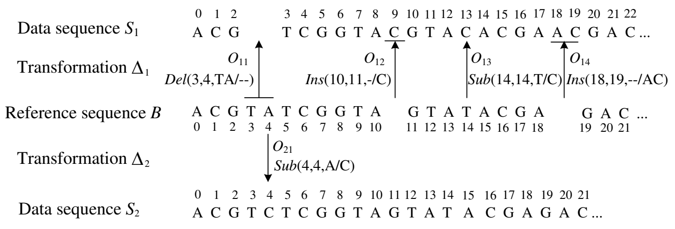
We end this paper by conducting an experimental study to evaluate these techniques and compare them with existing solutions. The results on real sequences demonstrate the space and time efficiency of the proposed techniques. Our approach outperforms existing methods in terms of both time and space.

II. Preliminaries

A. Sequences in a Compressed Representation

Let $\Sigma = \{\texttt{A,C,G,T}\}$ be the alphabet of characters in genomic sequences. For a sequence S of the characters in Σ, we use $|S|$ to denote its length, $S[t]$ to denote its t-th character (starting from 0), and $S[t_1, t_2]$ to denote the substring from its t_1-th character to its t_2-th character. A q-gram starting at position t in S is a substring $S[t, t+q-1]$, denoted by $g_t(S)$.

Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a collection of genomic sequences. As shown in Fig. 1, each S_i is represented in a compressed format using a reference sequence B and transformation Δ_i. The substrings in B that remain unchanged under Δ_i are called the *preserved substrings of B with respect to S_i*, or simply *preserved substrings* if the sequence S_i is clear in the context. The inserted or substituted substrings under Δ_i are called *incremental substrings of B with respect to S_i*, or simply *incremental substrings* if S_i is clear in the context. The length difference between B and a sequence S_i due to an operation O_{ij} is denoted by $l(O_{ij})$.



Data sequence S_1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
A C G T C G G T A C G T A C A C G A A C G A C ...
Transformation Δ_1
O_{11} Del(3,4,TA/--)
O_{12} Ins(10,11,-/C)
O_{13} Sub(14,14,T/C)
O_{14} Ins(18,19,--/AC)
Reference sequence B
A C G T A T C G G T A G T A T A C G A G A C ...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
Transformation Δ_2
O_{21} Sub(4,4,A/C)
Data sequence S_2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
A C G T C T C G G T A G T A T A C G A G A C ...

Fig. 1. Genomic sequences represented as differences to a reference.

Fig. 1 shows two example transformations Δ_1 and Δ_2, consisting of four and one edit operations, respectively. The edit operation O_{11} deletes the two characters TA starting at the third position of the reference sequence B. Since O_{11} reduces the length of sequence S_1 by 2, we have $l(O_{11}) = -2$. The edit operation O_{12} inserts a new character C after the 10-th position of B, so $l(O_{12}) = 1$. The edit operation O_{13} replaces the 14-th character T with a character C. This substitution operation does not change the length of S_1, so $l(O_{12}) = 0$. The substring $B[0, 2]$ is preserved with respect to S_1 and the character C resulting from the operation O_{12} is an incremental substring.

B. Pattern Search Queries

We focus on the following common types of queries on genomic sequences.

Exact Pattern Search. Given a pattern P, find all substrings of the sequences in $\mathcal{S}$ that match P exactly. Each answer is represented as a start position in a sequence in $\mathcal{S}$. For example, consider the two sequences in Fig. 1. For the query pattern CGTA, position 9 in sequence S_1 is an answer since the corresponding substring $S_1[9, 12]$ matches the pattern exactly.

Approximate Pattern Search. Given a pattern P, find all substrings of the sequences in $\mathcal{S}$ that match P approximately. There are different metrics to measure the similarity between two sequences. We will focus on the widely-adopted edit distance measure.

Edit Distance: The edit distance between two sequences α and β, denoted by $ed(\alpha, \beta)$, is the minimum number of single-character insertions, deletions, and substitutions that are needed to transform α to β. When using edit distance, we want to find all the substrings in the data set whose edit distances to the pattern are less than or equal to a given threshold τ. In Fig. 1, suppose a pattern P = CAGTA and the edit-distance threshold $\tau = 1$. The substrings $S_1[4, 8]$, $S_1[9, 12]$, $S_2[6, 10]$, $S_2[9, 13]$, and $S_2[10, 13]$ are the answers to P.

III. Exact Pattern Search

In this section, we develop an efficient algorithm to answer exact pattern search queries. In Section III-A, we classify possible answers into two categories. In III-B, we give an algorithm for finding answers in the first category. We leave the discussion of finding answers in the second category in Section VI.

A. Classification of Answers

Consider a sequence S_i in a compressed format, represented by a transformation Δ_i from a reference sequence B to S_i. Our goal here is to find the substrings of S_i that match a pattern P (generally $|P| \ll |B|$) exactly.

A substring $S_i[a, b]$ that matches P can be grouped in one of two categories. Category (1): There exists a q-gram (where q is a positive integer) in a preserved substring in B that matches a substring of $S_i[a, b]$. We call this q-gram a *seed*. Category (2): Such a seed does not exist. In this section we study how to find such $S_i[a, b]$ substrings that are in the first category. In Section VI we will study how to find such $S_i[a, b]$ substrings in the second category by searching in those incremental substrings.

B. Algorithm Description

BASIC Algorithm: This algorithm computes answers in category (1) by searching in preserved substrings in the reference sequence B (see Algorithm 1). It first decomposes a pattern P into a set of q-grams, and finds every occurrence of $g_j(P)$ ($0 \leq j \leq |P|-q+1$) in B. The algorithm ignores the matching gram $B[t, t+q-1]$ of $g_j(P)$ if Δ_i modifies some characters in it (line 4). It then checks $P[0, j-1]$ on B for all sequences in $\mathcal{S}$ in parallel (lines 6 – 13). It checks the left part from position t in B by examining characters in preserved substring of B and $P[0, j-1]$ until it meets the first transformation Δ_i at position π'_b (line 7). It then calculates $B[t]$'s corresponding position in S_i under Δ_i, denoted by $\Delta_i(t)$, and calls a function "VERIFY" to check if the substring starting at $\Delta_i(t) - j$ of length $|P|$ matches the pattern P, where $\Delta_i(t)$ denotes $B[t]$'s corresponding position in S_i under Δ_i (lines 10 – 13). A similar process is done for $P[j+1, |P|-1]$ on B (lines 14 – 22). After the above checks, the algorithm inserts every pair $(S_i, \Delta_i(t)-j)$ corresponding to a remaining transformation Δ_i into the results since the transformation Δ_i does not modify its corresponding preserved substring (lines 23 – 24).

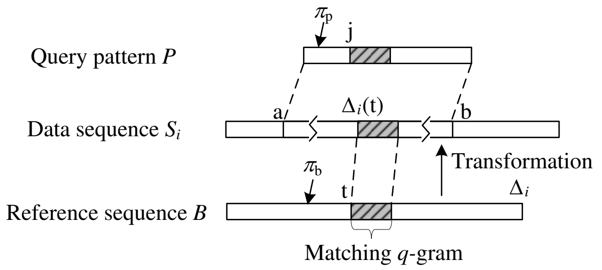


Fig. 2. A substring of S_i matching the pattern P has a matching q-gram in the reference sequence B.

The main idea of the function "VERIFY(P, j_l, j_r, B, t_l, t_r, Δ_i)" is as follows. We want to check if the substring $S_i[a, b]$ matches P exactly (see Fig. 2). Since the original sequence S_i is not physically stored, we need to construct the substring $S_i[a, b]$ on the fly using B and Δ_i, and compare its characters with those in P. The function returns FALSE as soon as it finds a mismatch between a character in B and the corresponding character in P.

The function first verifies substring $P[0, j_l]$. It starts by setting $\pi_p = j_l$ and $\pi_b = t_l$ (line 1). For each position π_b, it checks whether there is an insertion located between $B[\pi_b]$ and $B[\pi_b + 1]$ (line 3). If so, it compares the inserted substring I with the corresponding substring in P (lines 4 – 7). It then checks Δ_i's operation on $B[\pi_b]$ (line 10). We consider three possible cases of this operation, namely identity mapping (lines 11 – 13), substitution (lines 14 – 16), and deletion (lines 17 – 18). We do the corresponding character comparison and adjust the two cursors accordingly. If the verification on $P[0, j_l]$ passes, we then proceed to verify $P[j_r, |P|-1]$ starting from $P[j_r]$ similarly (line 21). The function returns TRUE if all verification steps have passed (line 22).

Algorithm 1: BASIC – Exact pattern search.

```
   Input: A pattern $P$, a reference sequence $B$, a set of sequences $\mathcal{S}$ =
          $\{S_1, \ldots, S_n\}$ stored as transformations $\mathbf{\Delta}$ from $B$ to $\mathcal{S}$;
   Output: Starting positions of substrings of each $S_i$ in $\mathcal{S}$ matching $P$;
1  Result set $R = \emptyset$, $R' = \emptyset$;
2  Decompose $P$ to a set of $q$-grams;    // Category (1)
3  foreach q-gram $B[t, t+q-1]$ matching a q-gram $P[j, j+q-1]$ do
4      Find those transformations $\mathbf{\Delta}' \in \mathbf{\Delta}$ that do not modify
       $B[t, t+q-1]$;
5      $\pi_p = j-1$; $\pi_b = t-1$;    // set cursors
6      while $\mathbf{\Delta}' \neq \emptyset$ or $\pi_p \geq 0$ do
7          Find the lowest position $\pi'_b$ such that $B[\pi'_b, \pi_b]$ matches
           $P[\pi'_p, \pi_p]$ and $B[\pi'_b, \pi_b]$ is not modified by any $\Delta_i \in \mathbf{\Delta}'$;
8          if $\pi'_p \neq 0$ then
9              $\pi'_p = \pi_p + \pi'_b - \pi_b - 1$;    $\pi'_b = \pi'_b - 1$;
10             foreach $\Delta_i \in \mathbf{\Delta}'$ that modifies $B[\pi'_b]$ do
11                 if VERIFY($P$, $\pi'_p$, $j+q$, $B$, $\pi'_b$, $t+q$, $\Delta_i$) then
12                     Insert the pair $(S_i, \Delta_i(t)-j)$ into $R$;
13                 $\mathbf{\Delta}' = \mathbf{\Delta}' - \Delta_i$;
14     $\pi_p = j+q$; $\pi_b = t+q$;
15     while $\mathbf{\Delta}' \neq \emptyset$ or $\pi_p \leq |P|-1$ do
16         Find the greatest position $\pi'_b$ such that $B[\pi_b, \pi'_b]$ matches
           $P[\pi_p, \pi'_p]$ and $B[\pi_b, \pi'_b]$ is not modified by any $\Delta_i \in \mathbf{\Delta}'$;
17         if $\pi'_p \neq |P|-1$ then
18             $\pi'_p = \pi_p + \pi'_b - \pi_b + 1$;    $\pi_b = \pi'_b + 1$;
19             foreach $\Delta_i \in \mathbf{\Delta}'$ that modifies $B[\pi'_b]$ do
20                 if VERIFY($P$, 0, $\pi'_p$, $B$, $t-j$, $\pi'_b$, $\Delta_i$) then
21                     Insert the pair $(S_i, \Delta_i(t)-j)$ into $R$;
22                 $\mathbf{\Delta}' = \mathbf{\Delta}' - \Delta_i$;
23     foreach $\Delta_i \in \mathbf{\Delta}'$ do
24         Insert the pair $(S_i, \Delta_i(t)-j)$ into $R$;
   // Category (2)
25 $R'$ = search incremental substrings (see Section VI);
26 return $R \cup R'$;
```

Inverted index of grams: We build an inverted index of q-grams for the reference sequence B. For each q-gram in B, we generate an inverted list of grams represented as their starting positions in B. For instance, in the running example, the list of the gram CG includes positions "1", "6", and "16" of the three matching grams in B.

Index for transformations: We use a binary tree, called *transformation tree*, to store all transformations. We use the end positions of edit operations as the keys to build the tree. Each leaf node stores the following information for edit operations with the same end position: (1) their start and end positions in B, (2) the types of each operation, (3) changes made by each operation, and (4) the corresponding position in S_i for each key (end position in B).

The performance of the algorithm BASIC depends on the number of positions on matching gram lists that need to be verified. Let the number of sequences in $\mathcal{S}$ be n, $|B| = l$, $|P| = m$, and the number of edit operations in all transformations $\mathbf{\Delta}$ be n_{op}. The time complexity of verification for each position in matching gram lists is $C_V = O(mn + \frac{m}{l} \cdot n_{op} \cdot \log n_{op})$ and the number of positions on matching gram lists is N_B. Then the time complexity of the algorithm BASIC is the summation of time for scanning matching inverted lists, searching in the transformation tree, and verifications, i.e., $C_{\text{BASIC}} = O(N_B + N_B \cdot \log n_{op} + C_V \cdot N_B)$.

Function VERIFY $(P, j_l, j_r, B, t_l, t_r, \Delta_i)$

```
Input: A pattern $P$, a position $j_l$ in $P$, a position $j_r$ in $P$, a reference
       sequence $B$, a position $t_l$ in $B$, a position $t_r$ in $B$, a
       transformation $\Delta_i$ from $B$ to $S_i$;
Output: TRUE if substring $S_i[\Delta_i(t) - j, \Delta_i(t) - j + |P| - 1]$ matches
       $P$, and FALSE otherwise;
// verify substring $P[0, j_l]$
$\pi_p = j_l$; $\pi_b = t_l$;    // set cursors
while $\pi_p \geq 0$ and $\pi_b \geq 0$ do
    if $\Delta_i$ inserts a substring $I$ immediately after $B[\pi_b]$ then
        $\alpha = |I| - 1$;
        while $\pi_p \geq 0$ and $\alpha \geq 0$ do
            if $P[\pi_p] \neq I[\alpha]$ then return FALSE;
            $\pi_p = \pi_p - 1$; $\alpha = \alpha - 1$;
        if $\pi_p == -1$ then
            continue;// no letters left in $P$
    switch $\Delta_i$'s operation on character $B[\pi_b]$ do
        case Identity mapping:
            if $P[\pi_p] \neq B[\pi_b]$ then return FALSE;
            $\pi_p = \pi_p - 1$; $\pi_b = \pi_b - 1$;
        case Substitution:
            if $P[\pi_p] \neq$ the new character then return FALSE;
            $\pi_p = \pi_p - 1$; $\pi_b = \pi_b - 1$;
        case Deletion:
            $\pi_b = \pi_b - 1$;
if $\pi_p \neq -1$ then
    return FALSE;// no letters left in $B$
verify $P[j_r, |P| - 1]$ similarly;
return TRUE;
```

IV. Minimizing Matching Grams to Verify

In this section, we study how to optimize the BASIC algorithm by reducing the number of grams to be verified while still obtaining all the answers.

Often an answer substring in a sequence S_i contains multiple q-grams that are preserved in B. Each of these grams is a matching gram and needs to be verified in the algorithm, although they lead to the same answer. Consider the running example in Fig. 1 and a query $P =$ACGT. The two grams $g_0(P)$ and $g_1(P)$ of the query appear in B at positions 0 and 1, respectively. As a consequence, the algorithm needs to verify these two different positions although they are led to the same answer $B[0, 5]$ for Δ_1 and the same answer $B[0, 3]$ for Δ_2. Our goal is to avoid this type of *duplicate verification*.

A straightforward way to avoid duplicate verification is to first record all starting positions $\Delta_i(t) - j$ for each matching gram of $g_j(P)$ at position t in B and then eliminate duplicate starting positions on them before verifying. The cost of this approach is $O(N_B + N_B \cdot \log n_{op} + C_V \cdot N')$, where N' is the number of distinct starting positions $\Delta_i(t) - j$. Although $N' \leq N_B$, we have to calculate $\Delta_i(t) - j$ for N_B times, which is obviously undesirable.

The main idea of our solution is to add a small amount of bit information at each position of the inverted list of a gram. This information can effectively help us prune some of the positions in the search. We will show that with this amount of additional information, we can improve search performance significantly.

A. Avoiding Duplicate Verifications

Our optimization is to avoid duplicate verifications that generate the same answer substring from different matching grams.

Definition 1: (*Head Grams*) A gram $B[h, h + q - 1]$ in the reference sequence B is called a *head gram* with respect to the transformation $\Delta_i : B \rightarrow S_i$ if there is an edit operation immediately before $B[h]$, i.e., Δ_i either deletes or substitutes the character $B[h - 1]$, or inserts one or more characters between $B[h - 1]$ and $B[h]$.

Definition 2: (*Tail Grams*) A gram $B[t, t + q - 1]$ in the reference sequence B is called a *tail gram* with respect to the transformation $\Delta_i : B \rightarrow S_i$ if there is an edit operation immediately *after* $B[t + q - 1]$, i.e., Δ_i either deletes or substitutes the character $B[t + q]$, or inserts one or more characters between $B[t + q - 1]$ and $B[t + q]$.

Lemma 1: Let $g_s(P)$ be an arbitrary gram in P. In the algorithm BASIC, we can find the same answer substrings by only considering the following grams:

- matching grams for the gram $g_s(P)$,
- head grams matching a gram $g_l(P)$ $(l > s)$, and
- tail grams matching a gram $g_r(P)$ $(0 \leq r < s)$.

Proof: Consider an answer substring in S_i that matches the query pattern P (see Fig. 3). Let $g_j(P)$ be a query substring matching a gram $B[k, k+q-1]$ in the reference sequence B, such that the corresponding verification can produce this answer substring (see the assumption in Section III-B). There are three cases:

- $j = s$: In this case, $B[k, k + q - 1]$ is a matching gram of $g_s(P)$.
- $j > s$: Consider $B[k, k + q - 1]$'s *leftmost* gram $B[h, h + q - 1]$, such that $k - h \leq j - s$, and there is no edit operation (including insertions) on the characters between $B[h]$ and $B[k]$. If $k - h = j - s$, then $B[h, h + q - 1]$ is a matching gram of $g_s(P)$. Otherwise, $B[h, h + q - 1]$ is a head gram matching $g_{j-(k-h)}(P)$.
- $j < s$: Consider $B[k, k + q - 1]$'s *rightmost* gram $B[t, t + q-1]$, such that $t-k \leq s-j$, and there is no edit operation (including insertions) on the characters between $B[k]$ and $B[t + q - 1]$. If $t - k = s - j$, then $B[t, t + q - 1]$ is a matching gram of $g_s(P)$. Otherwise, $B[t, t + q - 1]$ is a tail gram matching $g_{j+(t-k)}(P)$.

Therefore, each gram $g_m(P)$ $(j-k+h \leq m \leq j)$ in P matches the corresponding gram at position $k - h + m$ in B, and its corresponding verification will produce the same answer. ■

B. Minimizing Positions on Matching Gram lists to Process

For each gram $g_s(P)$ $(s \geq 0)$ of the query pattern, we want to find its matching grams that are head or tail grams. To achieve this goal, we modify the inverted lists by adding $2n$ bits for each position in gram lists. For every two bits, one bit indicates whether a head gram starts at this position, and the other bit indicates whether a tail gram starts at this position. We call this index a B-inverted index, where "B" stands for "Basic." For example, consider the position $\langle 1, \underline{01}00 \rangle$ in the

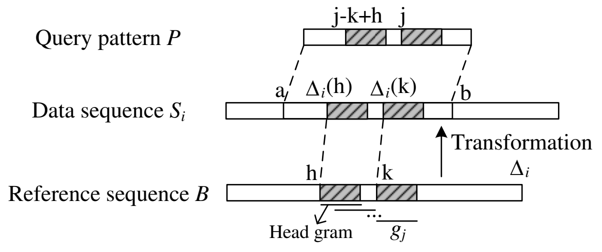


Fig. 3. Intuition of ignoring matching grams.

Algorithm 2: Choose an optimal gram $g_s(P)$.

```
Input: A pattern $P$, inverted lists of $q$-grams in a reference sequence $B$;
Output: The offset $s$ such that $g_s(P)$ is the optimal gram;
1 Decompose $P$ to a set of $q$-grams $g_j(P)$ and retrieve matching lists;
2 $min = |L(g_0(P))|$;   $tcount = |T(g_0(P))|$;   $s = 0$;
3 foreach $j = 1; j < |P| - q + 1; j++$; do
4     $min = min + |H(g_j(P))|$;   $count = tcount + |L(g_j(P))|$;
5     if $min > count$ then
6         $min = count$;   $s = j$;
7     $tcount = tcount + |T(g_j(P))|$;
8 return $s$;
```

list of gram CG in Fig. 4. It means that the gram CG at position 1 is a tail gram but not a head gram in sequence S_1, and is neither head nor tail gram in sequence S_2.

Grams		Lists with head and tail gram flags
AC	$\rightarrow$	$\langle 0, \underline{1010}\rangle$, $\langle 15, \underline{1000}\rangle$, $\langle 20, \underline{0000}\rangle$
AG	$\rightarrow$	$\langle 10, \underline{0000}\rangle$, $\langle 18, \underline{0000}\rangle$
AT	$\rightarrow$	$\langle 4, \underline{0000}\rangle$, $\langle 13, \underline{0000}\rangle$
CG	$\rightarrow$	$\langle 1, \underline{0100}\rangle$, $\langle 6, \underline{0000}\rangle$, $\langle 16, \underline{0000}\rangle$
GG	$\rightarrow$	$\langle 7, \underline{0000}\rangle$
GA	$\rightarrow$	$\langle 17, \underline{0100}\rangle$, $\langle 19, \underline{1000}\rangle$
GT	$\rightarrow$	$\langle 2, \underline{0001}\rangle$, $\langle 8, \underline{0000}\rangle$, $\langle 11, \underline{1000}\rangle$
TA	$\rightarrow$	$\langle 3, \underline{0000}\rangle$, $\langle 9, \underline{0100}\rangle$, $\langle 12, \underline{0100}\rangle$, $\langle 14, \underline{0000}\rangle$
TC	$\rightarrow$	$\langle 5, \underline{1010}\rangle$

Fig. 4. Inverted lists of q-grams with head and tail flags for S_1 and S_2.

Let $L(g_s(P))$ denote the set of positions on the inverted list of gram $g_s(P)$, $T(g_l(P))$ denote the set of positions with tail gram flags in the list of gram $g_l(P)$ $(0 \leq l \leq s-1)$, and $H(g_r(P))$ denote the set of positions with head gram flags in the list of gram $g_r(P)$ $(s+1 \leq r \leq |P|-q)$. Then the number of positions on matching gram lists that need to be verified is:

$$N_{min} = |L(g_s(P))| + \sum_{l=0}^{s-1} |T(g_l(P))| + \sum_{r=s+1}^{|P|-q} |H(g_r(P))|. \quad (1)$$

We want to choose an optimal gram $g_s(P)$ that minimizes the above summation.

MIN_VERIFY algorithm: Based on Lemma 1, we modify the algorithm BASIC to a new algorithm, called MIN_VERIFY, which uses a minimal number of positions on matching gram lists to do pattern search. The algorithm MIN_VERIFY consists of two steps: (i) It chooses an optimal gram $g_s(P)$ with a minimal number of positions to be processed. Algorithm 2 shows that this can be done lineally to $m - q + 1$. (ii) It calls the function VERIFY for a gram starting at each position in matching gram lists that satisfies one of the three conditions in Lemma 1. All other matching grams can be ignored.

For example, let the query P=CGTC. The algorithm MIN_VERIFY chooses gram $g_2(P)$=TC as the optimal gram. We only need to verify position $\langle 1, \underline{0100}\rangle$ on the $g_0(P)$=CG list, position $\langle 2, \underline{0001}\rangle$ on the $g_1(P)$=GT list, and $\langle 5, \underline{1010}\rangle$ on the $g_2(P)$=TC list. The total number of positions to be verified is 3. Compared to the approach that verify all positions of the three matching grams, we have 4 fewer positions to be verified. Moreover, position 1 only requires to be verified for S_1 and position 2 requires to be verified for S_2.

The time complexity of the algorithm MIN_VERIFY is C_{MIN} = $O(N_{min} + N_{min} \cdot \log n_{op} + C_V \cdot N_{min})$, where N_{min} is the number of minimal verified positions in matching lists. The algorithm MIN_VERIFY saves more time on scanning matching lists and traversing the transformation tree. It only scans N_{min} positions since the lists could be ordered according to their head and tail flags during construction of the inverted index. Although $N_{min} \geq N'$, the difference between them is in average $\frac{m}{l} n_{op}$, which is very small. In addition, in Section VIII we show that N_{min} is much smaller than N_B in real data sets.

V. REDUCING INDEX SIZE

Our experimental results show that the inverted index is much larger than the transformation tree, so we focus on how to reduce the size of the inverted index. We analyze the effect of discarding positions on inverted lists of grams in Section V-A. We then propose two lossy compression techniques in Sections V-B and V-C that can guarantee the answer to queries would not be different.

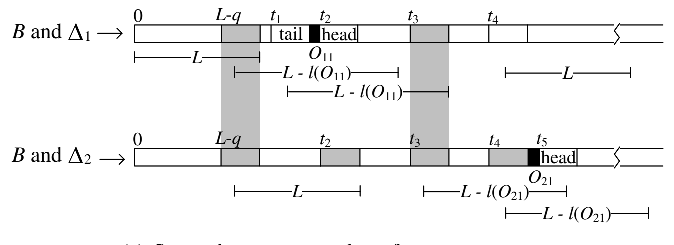
A. Discarding Positions on Inverted Lists of Grams

According to Equation 1, a position t in the list of a matching gram of $g_w(P)$ does not need to be verified if its matching gram $g_t(B) = B[t, t+q-1]$ is neither a head gram nor a tail gram. We call such a gram a *regular gram*. Since most positions associated with regular grams in matching gram lists do not need to be verified, then a natural question is: which of the positions associated with regular grams can be discarded while the remaining positions can still be used to answer queries?

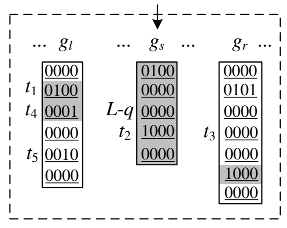
The following analysis shows the potential risk of missing answers due to incautious discard of some positions. Let t be a position on the inverted list of a gram, where the gram $g_t(B) = B[t, t+q-1]$ is a regular gram with respect to S_i. Let $S_i[a, b]$ be an answer substring and $\Delta_i(t) \in [a, b-q+1]$. Suppose the position t appears in the list of gram $g_s(P)$ using B-inverted index, and there is no position t' associated with a matching tail gram in the list of $g_l(P)$ $(l < s)$ to make $\Delta_i(t') - l \in [a, b-q+1]$, nor a position t' associated with a head matching head gram in the list of $g_r(P)$ $(r > s)$ to make $\Delta_i(t') - r \in [a, b-q+1]$. In this case, if we remove the positional gram at t from the inverted list of $g_s(P)$, then we will miss the chance of doing the verification to find this answer!

B. Discarding Partial Full-Regular Grams

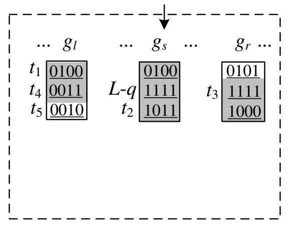
In this section we show that if there is a lower bound on query lengths, some certain positions on the inverted lists can be discarded safely, and with the remaining positions, the answer to queries would not be different. We develop a compression technique that can significantly reduce the inverted-index size.



(a) Set anchor grams on the reference sequence.



(b) Original gram lists.



(c) After lossy compression.

Fig. 5. Setting anchor grams (Grey boxes represent grams to be verified.).

Since each position in an inverted list is associated with a flag with $2n$ bits, we could only choose positional grams to be discarded from those positions who are associated with $2n$ bits 0, i.e., regular grams for all data sequences. We call such grams *full-regular grams*. We want to discard full-regular grams as much as possible without impairing the completeness of the results. For instance, the positional gram CG at position 1 could not be discarded since it is a tail gram associated with the data sequence S_1 (see Fig. 4).

To decide which positions can be discarded safely, we introduce a new kind of grams, called *anchor grams*, so that each candidate substring has at least one matching gram that is not discarded and needs to be verified. Let L be a lower bound of query lengths. We use a sliding window to go through the characters in B (see Fig. 5(a)). Let p be the starting position of the sliding window and its initial value be 0. For each position p, we check the substring $B[p, p+W-1]$ in the window, where W is the window size. If there is no edit operation in this substring, the last gram of the substring is picked as an anchor gram. For example, we set the gram $B[L-q, L-1]$ in Fig. 5(a) as an anchor gram in this window, which means $B[L-q, L-1]$ is an anchor gram for each sequence in $\mathcal{S}$. The benefit behind this is that when we slide the window to the next position $p+1$, this anchor gram is still inside the new window. Therefore, all full-regular grams but $B[L-q, L-1]$ in this window could be discarded safely. Fig. 5(b) shows the original gram lists. After setting this anchor gram $B[L-q, L-1]$, we discard all the other full-regular grams in the window $[0, L-1]$ and change the flag of $B[L-q, L-1]$ from $\underline{0000}$ to $\underline{1111}$ indicating that it is an anchor gram in both S_1 and S_2. Fig. 5(c) shows the compressed gram lists.

If a transformation Δ_i has an operation O_{ij} in the current sliding window, there must be a tail gram in the same window. We change the window size to $L - l(O_{ij})$, where $l(O_{ij})$ is the length difference between B and S_i due to O_{ij}. There are two possible cases as follows.

(1) Case 1: the new window has a head gram immediately after the position of operation O_{ij}. For instance, consider the operation O_{11} in the window starts at position $L-q+1$ in Fig. 5(a). In this case, there is no need of setting an anchor gram inside this window. The reason is that no matter which gram in the query is chosen as the $g_s(P)$ gram, there will be a tail gram $B[t_1, t_1+q-1]$ or a head gram $B[t_2, t_2+q-1]$ inside this window that needs to be verified. Notice that, an operation in a transformation Δ_1 might not appear in another transformation, e.g. Δ_2. As an instance, the size of the window at position $L-q+1$ for Δ_2 is L, because there is no operation inside this window. This window must contain the gram $B[t_1, t_1+q-1]$ otherwise the window $[L-q+1, 2L-q-l(O_{11})]$ would contain neither the tail gram $B[t_1, t_1+q-1]$ nor the head gram $B[t_2, t_2+q-1]$. The flag of position t_2 is changed from $\underline{00}$ to $\underline{11}$ for Δ_2 if it is inside this window as well (see Fig. 5(c)), otherwise, the flag of position t_1 is changed from $\underline{00}$ to $\underline{11}$ for Δ_2.

(2) Case 2: The next head gram is not in this window (e.g. the window $[t_3+1, t_3+L-l(O_{21})]$ in Fig. 5(a)). Then we set the tail gram $B[t_4, t_4+q-1]$ as an anchor gram. The goal is to make sure that this tail gram can always be chosen to do verification. We change the flag of position t_4 from $\underline{0010}$ to $\underline{0011}$. When we move the sliding window to a new position, where it does not contain the previous tail gram, but contains the next head gram, we set this head gram as an anchor gram.

We move the sliding window to the right step by step. At each position we repeat the above process, until we reach the end of the reference sequence B. We use the obtained head grams, tail grams, and anchor grams to build the inverted lists. We call this compressed index with anchor grams an A-inverted index, where "A" stands for "Anchor."

Search using A-inverted index: We now show that using the A-inverted index on the selected grams, we can still compute all answers to a query.

Lemma 2: Consider the algorithm BASIC. Let $g_s(P)$ be the selected gram of the pattern P. We can find the same answers by only considering the following grams:

- Matching grams on the A-inverted list of the gram $g_s(P)$;
- Tail grams and anchor grams matching a gram $g_i(P)$ ($0 \leq i < s$); and
- Head grams and anchor grams matching a gram $g_j(P)$ ($j > s$).

A_VERIFY algorithm: We modify the algorithm MIN_VERIFY slightly by using the A-inverted index. After choosing an optimal gram $g_s(P)$ using Equation 2, the function VERIFY is called for any gram whose starting position in the matching gram lists of A-inverted index satisfies one of the conditions in Lemma 2. We call this modified algorithm on A-inverted index "A_VERIFY."

Using the algorithm A_VERIFY, we need to verify the following number of positions on A-inverted index:

$$|L_A(g_s(P))| + \sum_{l=0}^{s-1} |T(g_l(P))| + \sum_{r=s+1}^{|P|-q} |H(g_r(P))| + \sum_{g_i \neq g_s, i=0}^{|P|-q} |A(g_i(P))|, \quad (2)$$

where "$L_A(g_s(P))$" is the set of positions on the list of gram $g_s(P)$ using A-inverted index, and "$A(g_i(P))$" is the set of positions of anchors on the list of gram $g_i(P)$ ($g_i \neq g_s$).

C. Discarding All Full-Regular Grams

In Section V-B the determining of discarding full-regular gram depends on a lower bound value on query lengths. In this section, we propose another compression technique without any constraints and still we can find all answers efficiently.

Instead of keeping positions of anchor grams in inverted lists, we discard starting positions of all full-regular grams from B-inverted index. We call this compressed index a C-inverted index. Compare with A-inverted index, a C-inverted index saves more space.

According to the analysis in Section V-A, an answer might be lost due to the missing of position t in C-inverted index. Let $S_i[a, b]$ be the missing answer substring. We summarize this scenario into the following three cases.

(i) There exists at least one position t' associated with a matching head gram of $g_l(B)$, where $l < s$ and $\Delta_i(t') - l \in [a, b - q + 1]$;

(ii) There exists at least one position t' associated with a matching tail gram of $g_r(B)$, where $r > s$ and $\Delta_i(t') - l \in [a, b - q + 1]$; or

(iii) There is no position t' in the C-inverted list of gram $g_k(B)$ satisfying $\Delta_i(t') - k \in [a, b - q + 1]$.

Now we show our search algorithm that could find all missing answers efficiently.

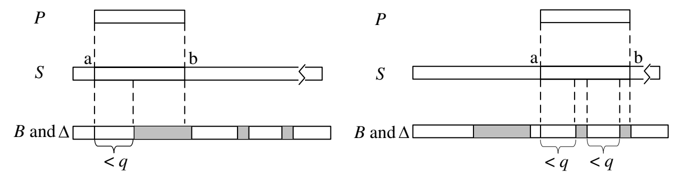
Search Using C-inverted index: For the first case, the discarded position t in the list of gram $g_s(P)$ can be calculated through t' easily. For each position t' in the list of gram $g_l(P)$ ($l < s$) using C-inverted index, we need to verify t' when $B[t' + s - l, t' + s - l + q - 1]$ matches $g_s(P)$ which implies $t = t' + s - l$. We could calculate t for the second case in a similar way.

C_VERIFY algorithm: We modify the algorithm MIN_VERIFY slightly by using the C-inverted index. We first choose an optimal gram $g_s(P)$ using Equation 2. We then call the function VERIFY for a gram starting at each position in matching gram lists of C-inverted index that satisfies one of the three conditions in Lemma 1. Then for each position t' in the list of gram $g_l(P)$ ($l < s$) using C-inverted index, we verify t' if $B[t' + s - l, t' + s - l + q - 1]$ matches $g_s(P)$; for each position t' in the list of gram $g_r(P)$ ($r > s$) using C-inverted index, we verify t' if $B[t' + s - r, t' + s - r + q - 1]$ matches $g_s(P)$. For the third case, we directly do exact pattern match in all preserved substrings in B. This could be done efficiently since the length of each preserved substring is short. We could ignore those preserved substrings with length less than $|P|$ and use the search algorithm in [3] to get all matchings efficiently. We call this modified algorithm on C-inverted index "C_VERIFY."

Using the algorithm C_VERIFY, the upper bound on the number of verified positions on C-inverted index is the summation of positions in all matching lists, i.e. $\sum_{u=0}^{|P|-1}(|L_C(g_u(P))|)$, where $L_C(g_u(P))$ is the set of positions on list of $g_u(P)$ using C-inverted index.

VI. Searching in Incremental Substrings

In Section III we showed that answers to a pattern-search query can be classified into two categories, and gave an algorithm for finding answers in the first category. In this section, we study how to find answers in the second category, i.e., those answers that do not have a matching q-gram in those preserved sequences. As a result, we need to search in those incremental substrings to find answers. As illustrated in Fig. 6(a), there are two possible cases for such an answer $S_i[a, b]$. For the case in Fig. 6(a), if $S_i[a, b]$ matches P, then $S_i[a + q - 1, b - q + 1]$ must match a substring of an incremental substring. For the case in Fig. 6(b), we could not find a substring $S_i[a, b]$ unless we decompress the whole data sequence S_i using B and Δ_i. This approach is computationally expensive.



(a) A matching substring in a long incremental substring.

(b) A matching substring mapped from adjacent edit operations.

Fig. 6. Matching substrings using B and Δ (white areas in B represent preserved substrings and grey areas represent incremental substrings).

In order to avoid the case in Fig. 6(b), we preprocess the data sequence to change short preserved substrings (with a length less than q) to incremental substrings. We first extend the definition of *incremental substring* by allowing identical-character matching in its edit operations. In the preprocessing step, we combine adjacent incremental substrings into a single incremental substring. In addition, for each short preserved substring X with a length less than q, we combine it with its adjacent edit operations to form a new incremental substring, and will no longer treat X as a preserved substring. For example, consider our running example in Fig. 1 and let $q = 4$. Then we combine $B[11, 13]$, O_{12} and O_{13} into one incremental substring CGTAC. We store it into the transformation tree and set "Merge" as its type, set 10 and 15 as its start and end positions.

Lemma 3: After preprocessing the data sequences, for every substring $S_i[a, b]$ matching a pattern P, there is either a matching q-gram of the answer in a preserved substring to match a q-gram in P or a substring of an incremental substring with length $\geq |P| - 2(q - 1)$ to match $P[q - 1, |P| - q]$.

Proof: After the preprocessing step, the length of each remaining preserved substring is greater than or equals to q. Suppose the query P has an answer substring G. If G has a substring of length q that can be mapped into a preserved substring, then the above algorithms can find this answer. Otherwise, since each preserved substring has a length $\geq q$, then G must be matching an incremental substring and its adjacent preserved substrings (at most two). Each of the adjacent preserved substrings can have at most $q-1$ characters matching G, thus G should have at least $|P|-2(q-1)$ characters matching the incremental substring. Such an incremental substring can be found by the above procedure. ■

Algorithm 3 shows that we need to check those incremental substrings with a length $\geq |P| - 2(q-1)$. Let R be such an incremental substring. We need to check whether $P[q-1, |P|-q]$ is a substring of R. If the check step passes, we need to do a verification by comparing the remaining characters in P with the corresponding characters in R and the characters in the adjacent preserved substrings.

Algorithm 3: Search in incremental substrings.

```
Input: A pattern $P$, a reference sequence $B$, a sequence $S_i$ stored as a
       transformation $\Delta_i$ from $B$ to $S_i$;
Output: Starting positions of substrings of $S_i$ matching $P$;
1  Result set $Res = \emptyset$;
2  foreach Incremental substring $R$ with length $\geq |P| - 2(q-1)$ do
3      if $P[q-1, |P|-q]$ is a substring of $R$ then
4          $t$ = starting position of the edit operation w.r.t. $R$;
5          if $R$ is a substring of $P$ at position $x$ then
6              $Res = Res \cup \{\Delta_i(t) + x\}$;
7          if $P[q-1, |P|-1]$ matches $R[0, |P|-q]$ then
8              if VERIFY($P[0, q-2]$, 0, $B$, $t$, $\Delta_i$, $\Delta_i(t)$) then
9                  $Res = Res \cup \{\Delta_i(t)\}$;
10         if $P[0, |P|-q]$ matches $R$ at position $y$ then
11             $t'$ = end position of the edit operation w.r.t. $R$;
12             if VERIFY($P[|P|-q+1, |P|-1]$, $|P|-q+1$, $B$, $t'$, $\Delta_i$,
               $\Delta_i(t')$) then
13                 $Res = Res \cup \{\Delta_i(t) + y\}$;
14 return $Res$;
```

It is interesting to note that the real data sets we tested contain only a few incremental substrings of lengths $\geq |P| - 2(q-1)$ (we will report it in Section VIII-C), which means the checking procedure can be done very efficiently.

VII. Approximate Pattern Search

The approximate pattern search finds substrings whose edit distances to a pattern P are less than or equal to a threshold τ, where $\tau \ll |P|$. We adopt the well-known pigeonhole principle to split P to $\tau + 1$ disjoint pieces $P[0, \pi_1 - 1], P[\pi_1, \pi_2 - 1], \ldots,$ and $P[\pi_\tau, |P|]$, where $\pi_u = \lceil \frac{u \cdot |p|}{\tau+1} \rceil$. For each piece $P[\pi_u, \pi_{u+1} - 1]$, we search its matches in the reference sequence using the exact pattern search methods described above (The parameter $L = \lceil \frac{|P|}{\tau+1} \rceil$ when using A-inverted index).

For each found substring, a straightforward way is to do the corresponding verification of its adjacent characters to check if the corresponding substring matches the pattern P under the given distance threshold. This approach needs to verify each found substring in different data sequences in $\mathcal{S}$, which is costly.

In this section, we propose a novel filtering technique to avoid verifying every found substring in $\mathcal{S}$. The main idea is that for each piece $P[\pi_u, \pi_{u+1} - 1]$, consider one of its grams $g_j(P)$, we find its matching gram $g_t(B)$ in B. From this gram $g_t(B)$, we might find a set of substrings in $S_1, \ldots, S_n$ that can answer the piece $P[\pi_u, \pi_{u+1} - 1]$. This property inspires us to calculate the edit distance between P and a substring of B to determine whether each corresponding substring S'_i of S_i could have an edit distance $ed(P, S'_i) \leq \tau$.

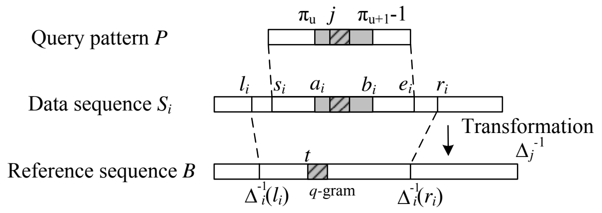


Fig. 7. A substring $S_i[s_i, e_i]$ of S_i is an answer to the pattern P such that $ed(P, S_i[s_i, e_i]) \leq \tau$.

Fig. 7 shows the idea of our filtering approach. Suppose a substring $S'_i = S_i[s_i, e_i]$ of S_i is one answer to the pattern P, i.e. $ed(P, S'_i) \leq \tau$, and its mapping substring is $B' = B[\Delta_i^{-1}(s_i), \Delta_i^{-1}(e_i)]$ in the reference sequence B. We then have:

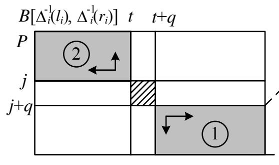
$$\begin{cases} |ed(P, B') - ed(B', S'_i)| \leq ed(P, S'_i) \\ ed(P, S'_i) \leq ed(P, B') + ed(B', S'_i) \\ 0 \leq ed(P, S'_i) \leq \tau \\ ed(B', S'_i) \leq \Delta_i(B', S'_i) \end{cases} \tag{3}$$

According to Equation 3, the lower bound on τ is $ed(P, B') - \Delta_i(B', S'_i)$. Once the given τ is less than this lower bound, we could conclude that $ed(P, S'_i) > \tau$ without uncompressing S'_i or calculating their edit distance.

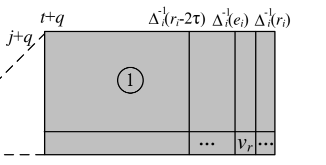
The difficulty in applying this property is that we could not know S'_i in advance as well as B'. If B' is known, it is easy to calculate $ed(P, B')$ and $\Delta_i(B', S'_i)$ due to the transformation Δ_i from B to S_i. Below we show how to estimate B' and get different safe lower bound values on τ for different sequences by only considering the edit distance between P and one substring of B.

We first consider the simple case where the data sequence set $\mathcal{S}$ only contains one sequence S_i. According to the pigeonhole principle, there must be some substring $S_i[a_i, b_i]$ of $S'_i = S_i[s_i, e_i]$ exactly matching a piece $P[\pi_u, \pi_{u+1} - 1]$ of P (see Fig. 7). In order to find S'_i based on the matching substring $S_i[a_i, b_i]$ under the edit distance threshold τ, we need to search P in $S_i[l_i, r_i]$, where $l_i = a_i - \pi_u - \tau$, $r_i = b_i + |P| - \pi_{u+1} + 1 + \tau$. Knowing that $s_i \in [l_i, l_i + 2\tau]$ and $e_i \in [r_i - 2\tau, r_i]$, we could get its corresponding substring $B'' = B[\Delta_i^{-1}(l_i), \Delta_i^{-1}(r_i)]$.

When B'' and $S_i[l_i, r_i]$ share the same gram $g_t(B)$, the substring B'' is divided into three parts: a left part $B_l = B[\Delta_i^{-1}(l_i), t-1]$, a gram $g_t(B)$, and a right part $B_r[t + q, \Delta_i^{-1}(r_i)]$. The query P is also divided into $P[0, j-1]$, $g_j(P)$, and $P[j + q, |P| - 1]$. Fig. 8(a) shows that we only need to calculate edit distance in areas ① and ② to find the lower bound on τ using the classical dynamic programming.



(a) The matrix for P and $B[l_i,r_i]$.



(b) Calculate $ed(P[j{+}q, |P|{-}1], B[t{+}q, y])$ for each $y \in [\Delta_i^{-1}(r_i{-}2\tau), \Delta_i^{-1}(r_i)]$.

Fig. 8. Calculate the lower bound value on τ for substring $S_i[s_i, e_i]$.

The area ① represents the matrix of calculating $ed(P[j+q, |P|-1], B[t+q, y])$ for each $y \in [\Delta_i^{-1}(r_i-2\tau), \Delta_i^{-1}(r_i)]$, among which we choose y with the minimum value $v_r = ed(P[j+q, |P|-1], B[t+q, y])$.

The matrix in the area ② is calculated in a similar way. We first get reversed substrings of $P[0, j-1]$ and $B[x, t-1]$, denoted by $(P[0, j-1])^{-1}$ and $(B[x, t-1])^{-1}$, respectively. After the calculation of $ed(P[0, j-1])^{-1}, (B[x, t-1])^{-1})$, we choose a minimum one, which is denoted by $v_l = ed(P[0, j-1])^{-1}, (B[x, t-1])^{-1})$ ($x \in [\Delta_i^{-1}(l_i), \Delta_i^{-1}(l_i+2\tau)]$). Then the minimum value of $ed(P, B'')$ is $v_l + v_r$. Based on the found substring $B[x, y]$, we can set $\Delta_i(B[x, y], S_i[\Delta_i(x), \Delta_i(y)])$ as the length difference between $B[x, y]$ and $S_i[\Delta_i(x), \Delta_i(y)]$ due to the operations between them, denoted by $l_d(B[x, y], S_i[\Delta_i(x), \Delta_i(y)])$. Since $v_l + v_r$ is less than $ed(P, B')$, the lower bound on τ can be represented as $v_l + v_r - l_d(B[x, y], S_i[\Delta_i(x), \Delta_i(y)])$.

This approach could be extended to multiple sequences by simply defining B'' as $B[\min\{\Delta_1^{-1}(l_1), \ldots, \Delta_n^{-1}(l_n)\}, \max\{\Delta_1^{-1}(r_1), \ldots, \Delta_n^{-1}(r_n)\}]$. We search P in B'' and calculate different lower bound values for different sequence S_i using the above approach.

VIII. Experiments

In this section, we present the experimental results of our techniques on large genomic sequences.

A. Experiment Setup

For experimental testing, we used two publicly available genomic sequences: One is the James Watson's genome (abbreviated as JWB) [1], and the other is the YanHuang genome (abbreviated as YH) provided by BGI (Beijing Genome Institute).[1] Each chromosome in JWB or YH is represented as a reference sequence B and a transformation Δ_i coding the difference between the reference sequence and the original one.[2] We used the same reference genome (HG19 from the UCSC genome browser) for both JWB and YH.

In order to obtain more transformations based on the same reference sequence, we chose one chromosome with large size and relatively high quality (called "chr1.fa") as a reference sequence. We then compute the differences between the other chromosomes and this reference sequence.

[1]Available at http://yh.genomics.org.cn/download.jsp#pd, in the section "YH variants and annotations."

[2]Available at http://www.ics.uci.edu/∼dnazip/

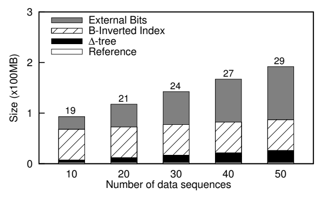
We constructed synthetic query workloads as follows. We chose a reference sequence, and randomly selected its substrings, in which 1% to 5% characters are modified. The query lengths varied from 20 to 2,000. For each query, we constructed a workload including 100 queries with the same query length. We ran each query workload 10 times and computed the average performance numbers. We also used BLAST benchmark queries [3] to construct query workloads. The performance is similar to the results of using a synthetic query workload. For space reasons, we do not report the results of using BLAST benchmark queries.

All the algorithms were implemented using GNU C++. The experiments were run on an HP DC 7800 PC with an Intel 3.16GHz Dual Core CPU and 3.2GB memory with a 500GB disk, running a Ubuntu (Linux) operating system. Index structures were in memory.

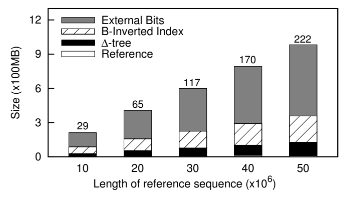
B. Index Size

We first evaluated our techniques for the case of 50 data sequences. We generated reference sequences with different sizes by selecting substrings of B (let B_r be such a reference sequence).

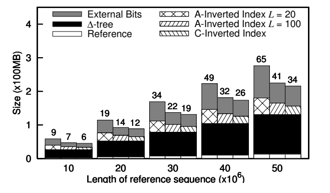
1) Basic Indexes: We measured the sizes of indexes described in Sections III and IV. We separately measured the size of the bits for head and tail grams, as described in Section IV. Figs. 9(a) and 9(b) show the sizes of the reference sequence, transformation tree, inverted lists, and the bits for head and tail grams (denoted by "External Bits").



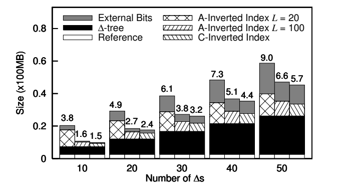
(a) Basic index ($|B_r|$ = 10 million).



(b) Basic index of 50 data sequences.



(c) Compressed indexes of 50 data sequences.



(d) Compressed indexes ($|B_r|$ = 10 million, $|\mathbf{\Delta}|$ = 50).

Fig. 9. Index size on multiple sequences (The value on each bar indicates construction time of the corresponding index in seconds.).

Fig. 9(a) shows the index sizes of a reference sequence with 10 million characters when the number of data sequences increased. The size of the reference sequence was 2.5MB (two bits for each character). When q=10, we used 19 seconds to construct the index for 10 data sequences. The corresponding transformation tree (denoted by Δ-tree in the figure) was only 4.84MB, while the size of the inverted lists was 60.65MB. The

[3]Available at ftp://ftp.ncbi.nlm.nih.gov/blast/demo/benchmark

TABLE I
NUMBER OF VERIFIED GRAMS ON A REFERENCE SEQUENCE OF 10 MILLION CHARACTERS AND 50 TRANSFORMATIONS OF 498, 870 EDIT OPERATIONS. ALL REDUCTION RATIOS ARE WITH RESPECT TO THE ALGORITHM BASIC.

Average query length	q	Inverted index with bits for head/tail grams			Compressed inverted index using anchor grams					
		BASIC	MIN_VERIFY		A_VERIFY (L = 20)		A_VERIFY (L = 100)		C_VERIFY	
		# of verified grams	# of verified grams	Reduction	# of verified grams	Reduction	# of verified grams	Reduction	# of verified grams	Reduction
20	7	73,133,250	2,496,372	96.59%	6,434,479	91.20%	–	–	284,822	99.61%
	9	6,834,500	276,686	95.95%	686,008	89.96%	–	–	26,486	99.61%
	10	2,395,650	116,277	95.15%	256,869	89.28%	–	–	9,677	99.60%
	11	1,106,950	65,133	94.12%	149,802	86.47%	–	–	5,033	99.55%
100	7	486,679,400	2,392,503	99.51%	42,009,689	91.37%	6,301,306	98.69%	640,353	99.87%
	9	62,934,900	330,116	99.48%	6,202,810	90.14%	817,493	98.67%	77,616	99.88%
	10	31,203,850	137,501	99.56%	3,280,748	89.48%	408,114	98.66%	35,301	99.89%
	11	19,432,900	47,822	99.75%	2,521,016	87.03%	251,264	98.65%	19,072	99.90%
2,000	7	11,361,527,800	13,147,183	99.88%	977,490,127	91.40%	142,728,240	98.74%	11,396,783	99.90%
	9	1,838,545,000	2,131,491	99.88%	180,707,216	90.17%	23,390,723	98.73%	1,812,091	99.90%
	10	1,047,048,450	1,152,452	99.89%	109,145,784	89.58%	13,437,476	98.72%	1,019,352	99.90%
	11	705,411,700	780,681	99.89%	90,962,219	87.11%	9,096,410	98.71%	687,331	99.90%

size of the inverted lists did not change along with the increase of the number of sequences for the reason that the size is only dependent on the reference sequence. The slow increase of the transformation tree can be due to the high similarity between the data sequences and the reference sequence. The results for other q values were similar.

Fig. 9(b) shows the index size of 50 data sequences. When the length of reference sequence $|B_r|$ was 10 million and the gram length q was 10, the size of the transformation tree was 23.7 MB, while the size for inverted lists was 60.65MB. The size of the bits for head and tail grams was 125MB. The index took 29 seconds to construct. The index sizes grew linearly when the length of the reference sequence $|B_r|$ increased from 10 million to 50 million.

2) Compressed Indexes: The sizes of the two compressed indexes described in Section V were also measured.

Fig. 9(c) shows the results of 50 data sequences for different sequence lengths. For 50 sequences, when $|B_r| = 10$ million and L=100, the size of A-inverted list was 9.06MB, which was only 14.9% of the size of the original index! The construction time was 7 seconds. As L decreased, both the index size and construction time increased. The size of C-inverted list was less than the size of A-inverted list, which was 7.32MB, 12.1% of the original size, with construction time in 6 seconds.

Fig. 9(d) shows the advantages of our compression technique for different number of data sequences. When there were 10 sequences, each of which containing 10 million characters, the total index size of using anchor grams was only 20.33MB for L=20 and 10.74MB for L=100. The construction time was 3.8 seconds and 1.6 seconds, respectively. C-inverted index required less space and construction time compared with A-inverted index. When storing more sequences as differences from the same reference sequence, the size of each compressed inverted index increased more slowly than the size of the transformation tree.

C. Performance of Exact Pattern Search

We measured the performance of exact pattern search using the proposed techniques.

1) Number of Verified Grams: Table I shows the numbers of verified grams. When q=10 and the query length was 100, the algorithm BASIC needed to verify 31, 203, 850 grams, while algorithm MIN_VERIFY reduced the number by 99.56%! We also evaluated the number of verified grams using compressed inverted indexes. The algorithm A_VERIFY reduced the number of verified grams by 89.48% when L=20, and 98.66% when L=100. The number of verified grams by using the algorithm C_VERIFY was the smallest. We got similar results when using other gram lengths and query lengths.

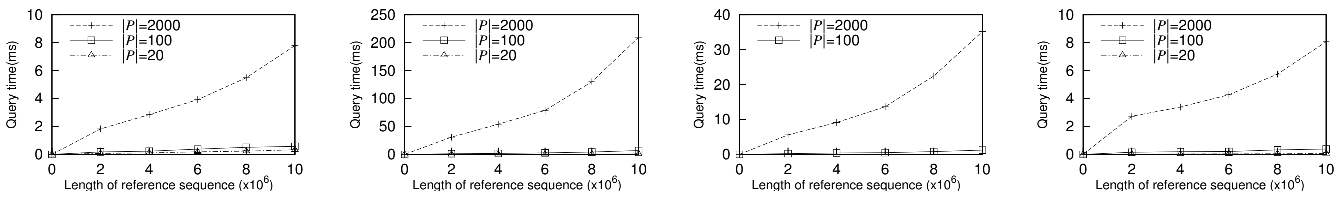
Notice that, for a reference sequence with 10 million characters and 498, 870 edit operations on 50 sequences, the number of incremental substrings that needed to be checked was zero when $|P|$=2000. This number increased to 7 and 191 when $|P|$=200 and $|P|$=20, respectively, which was relatively very small to the real data set. It shows the realistic of Category (1) discussed in Section III-B.

2) Query Time: Fig. 10 shows the performance of exact pattern search on 50 sequences. The performances of MIN_VERIFY and C_VERIFY were very close as Figs. 10(a) and 10(d) shown. When $|B_r| = 10$ million and the lengths of queries were 20, MIN_VERIFY took average 0.32ms while the number was 0.20ms for C_VERIFY. When the query length increased to 2, 000, the average running time became 7.79ms and 8.06ms for MIN_VERIFY and C_VERIFY respectively. Compared with these two algorithms, A_VERIFY spent much longer time as shown in Figs. 10(b) and 10(c).

D. Performance of Approximate Pattern Search

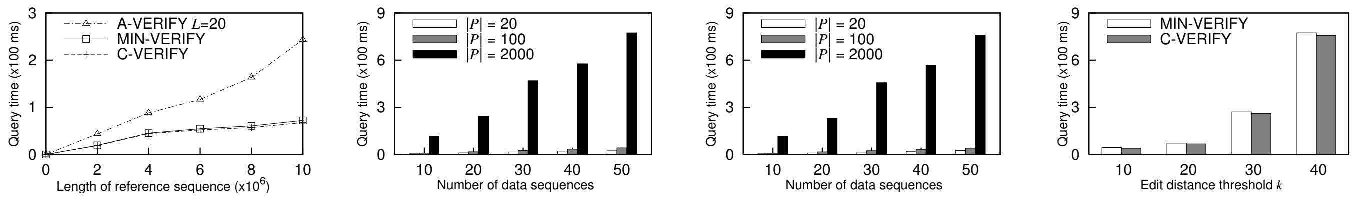
We implemented the approaches in Section VII for conducting approximate pattern search. We set $q = 10$.

Fig. 11(a) shows that the query time of the algorithm MIN_VERIFY on B-inverted index and the algorithm C_VERIFY on C-inverted index were close. The algorithm A_VERIFY on A-inverted index took longer time than the other two approaches. Figs. 11(b) and 11(c) show the performances of MIN_VERIFY and C_VERIFY when increasing the number of data sequences, respectively, and Fig. 11(d) shows the performance when varying the edit distance threshold. All these results reflect the good scalability of our approach.



(a) MIN_VERIFY on B-inverted index. (b) A_VERIFY on A-inverted index (L=20). (c) A_VERIFY on A-inverted index (L=100). (d) C_VERIFY on C-inverted index.

Fig. 10. Performance of exact pattern search using synthetic query workloads on 50 sequences ($q = 10$).



(a) 50 sequences, $|P|$ = 2000, $\tau = 20$. (b) MIN_VERIFY on B-inverted index ($|B_r|$=10 million, τ=$|P| \times 2\%$). (c) C_VERIFY on C-inverted index ($|B_r|$=10 million, τ=$|P| \times 2\%$). (d) 50 sequences, $|P|$ = 2000, $|B_r|$ = 10 million.

Fig. 11. Performance of approximate pattern search using edit distance and synthetic query workloads ($q = 10$).

E. Comparisons with Other Algorithms

We compare our techniques with Boyer-Moore algorithm (BM for short) [4], Fredriksson's filtration algorithm (FF for short) [5], RLCSA algorithm [6][4], LZ77Index algorithm [7][5], SLPIndex algorithm [8][6], and GenomeMapper software [9][7].

TABLE II
COMPARISON OF SPACE OCCUPANCIES IN MEMORY OF 50 SEQUENCES ($|B_r| = 10$ MILLION).

Algorithms	Space occupancies in memory		
	$\lvert\boldsymbol{\Delta}\rvert = 10$	$\lvert\boldsymbol{\Delta}\rvert = 30$	$\lvert\boldsymbol{\Delta}\rvert = 50$
Ours	18.5 MB	31.9 MB	49.0 MB
BM	95.8 MB	293.5 MB	477.3 MB
FF	103.4 MB	318.2 MB	545.8 MB
RLCSA	18.3 MB	31.4 MB	48.2 MB
LZ77index	117.3 MB	199.3 MB	302.3 MB
SLPindex	934.2 MB	1632.3 MB	2545.3 MB
GenomeMapper	252.6 MB	458.2 MB	712.2 MB

The first two algorithms BM and FF are not designed for compressed sequences, so we decompressed each data sequence before running these algorithms. The other four algorithms support pattern search on compressed sequences based on different compression format (see Section IX). We used the reference sequence with 10 million characters and 50 data sequences with a transformation of 498, 870 edit operations. We decompressed the sequence by applying these edit operations on the reference sequence, which took 4.31 seconds. Our approach used the C-inverted index. Table II gives the comparison of space occupancies in memory. Table III shows the corresponding query performances on exact pattern search, and Table IV shows the comparison performance of approximate pattern search. The comparison results show that our approaches could provide better running performance than other approaches.

[4]Notice that the released code at http://www.cs.helsinki.fi/group/suds/rlcsa/ only supports exact pattern search.

[5]Available at http://pizzachili.dcc.uchile.cl/repcorpus.html

[6]We thanks the author to provide us the source code.

[7]Available at http://1001genomes.org/downloads/genomemapper.html

IX. Related Work

The adopted compressed format, in this paper, for a collection of genomic sequences was firstly proposed in [1]. It uses a reference sequence as a base and each sequence in the collection is considered as a set of edit operations from the base to this sequence. The first practical compression algorithms were reported in [2] followed by many work more recently [10], [11], [12], [6].

Mäkinen *et al.* in [6] proposed an algorithm RLCSA on a self-index structure, which represents the reference sequence using a BWT format. It enables the sequences being compressed at a considerably high ratio close to their high-order entropy. Kreft and Navarro in [7] presented the first self-index based on LZ77 compression, called LZ77index. The work in [13] stores a self-index for a reference sequence and compresses every other sequence as an LZ77 encoding relative to the base. SimDNA in [14] is an indexing tool for indexing similar DNA sequences. It makes use of the indexing data structure BWT (Burrow-Wheeler Transform) and suffix array to support exact pattern searching.

Another approach, called COMRAD, builds on grammar-based compression and only allows for random access to sequences without decompressing the whole data set [15], [16]. A follow-up tool, called GDC [17], supports an LZ77-style compression scheme on multiple genomic sequences of the same species and improves COMRAD on random access. Neither of COMRAD nor GDC supports pattern search.

Another work for repetitive sequences was proposed in [8], which provides both q-gram based inverted index and grammar-based index. The software GenomeMapper described

TABLE III
COMPARISON OF EXACT PATTERN SEARCH ALGORITHMS ON 50 SEQUENCES ($|B_r| = 10$ MILLION)

Algorithms	Query time $\|P\| = 20$			Query time $\|P\| = 100$			Query time $\|P\| = 2000$		
	$\|\mathbf{\Delta}\| = 10$	$\|\mathbf{\Delta}\| = 30$	$\|\mathbf{\Delta}\| = 50$	$\|\mathbf{\Delta}\| = 10$	$\|\mathbf{\Delta}\| = 30$	$\|\mathbf{\Delta}\| = 50$	$\|\mathbf{\Delta}\| = 10$	$\|\mathbf{\Delta}\| = 30$	$\|\mathbf{\Delta}\| = 50$
Ours	0.43 ms	0.48 ms	0.54 ms	0.53 ms	0.62 ms	0.74 ms	5.54 ms	6.53 ms	7.79 ms
BM	264.21 ms	423.38 ms	683.23 ms	284.39 ms	493.43 ms	692.32 ms	328.21 ms	537.48 ms	724.32 ms
FF	10.89 ms	17.19 ms	24.38 ms	16.43 ms	23.33 ms	31.34 ms	18.48 ms	26.13 ms	34.27 ms
RLCSA	0.88 ms	1.04 ms	1.13 ms	1.49 ms	1.78 ms	1.92 ms	7.98 ms	9.51 ms	10.26 ms
LZ77index	1.49 ms	1.78 ms	1.92 ms	2.83 ms	3.37 ms	3.64 ms	18.34 ms	23.25 ms	26.33 ms
SLPindex	2.83 ms	–	–	7.84ms	–	–	75.08 ms	–	–

TABLE IV
COMPARISON OF APPROXIMATE PATTERN SEARCH ALGORITHMS ON MULTIPLE SEQUENCES USING EDIT DISTANCE ($|B_r| = 10$ MILLION).

Algorithms	Query time $\|P\| = 20$, $\tau = 1$			Query time $\|P\| = 100$, $\tau = 2$			Query time $\|P\| = 2000$, $\tau = 40$		
	$\|\mathbf{\Delta}\| = 10$	$\|\mathbf{\Delta}\| = 30$	$\|\mathbf{\Delta}\| = 50$	$\|\mathbf{\Delta}\| = 10$	$\|\mathbf{\Delta}\| = 30$	$\|\mathbf{\Delta}\| = 50$	$\|\mathbf{\Delta}\| = 10$	$\|\mathbf{\Delta}\| = 30$	$\|\mathbf{\Delta}\| = 50$
Ours	0.01 Sec.	0.01 Sec.	0.02 Sec.	0.01 Sec.	0.02 Sec.	0.04 Sec.	0.17 Sec.	0.53 Sec.	0.88 Sec.
GenomeMapper	0.34 Sec.	0.82 Sec.	1.29 Sec.	0.97 Sec.	1.82 Sec.	3.25 Sec.	69.42 Sec.	123.28 Sec.	192.42 Sec.

in [9] allows mapping of reads against several reference genomes, which was also based on a q-gram filter.

The work in [18] supports lossless compression on multiple inverted lists based on the assumption that lists are similar between each other. This setting is different from ours. In our setting, genomic sequences are similar, but there exists no identical position in all the inverted lists.

There are many other algorithms for pattern search in an uncompressed long sequence besides those used in compressed ones. Typical algorithms on this problem include Boyer-Moore algorithm [4] and Fredriksson's filtration algorithm [5]. Recent studies include [19], [20]. These approaches assume sequences being stored in their original representation, while the focus of our work is to index sequences in a compressed format using their high similarity.

X. CONCLUSIONS

In summary, we proposed several novel techniques to index large genomic sequences to answer queries, motivated by recent rapid growth of whole-genome sequencing. We gave a full specification of these techniques, and presented a series of optimizations to improve their space and time efficiency. We also demonstrated the utilities of these algorithms on real genomic sequences.

ACKNOWLEDGMENT

The work is partially supported by the National Basic Research Program of China (973 Program) (No. 2012CB316201), the National Natural Science Foundation of China (Nos. 60973018, 60973020, 61272178), the Joint Research Fund for Overseas Natural Science of China (No. 61129002), the National Natural Science of China Key Program (No. 60933001), the National Natural Science Foundation for Distinguished Young Scholars (No. 61025007), the Doctoral Fund of Ministry of Education of China (No. 20110042110028), the Fundamental Research Funds for the Central Universities (No. N110804002), and NIH grants LM010235 and HG006870, and it has been partially supported by a grant from Intel Corporation's University Research Office.

REFERENCES

[1] D. A. Wheeler, M. Srinivasan, and et al., "The complete genome of an individual by massively parallel DNA sequencing," *Nature*, vol. 452, pp. 872–876, 2008.
[2] S. Christley, Y. Lu, C. Li, and X. Xie, "Human genomes as email attachments," *Bioinformatics*, vol. 25, no. 2, pp. 274–275, 2009.
[3] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, p. 552C581, 2005.
[4] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Comm. ACM*, vol. 20, pp. 762–772, 1977.
[5] K. Fredriksson and G. Navarro, "Average-optimal single and multiple approximate string matching," *ACM JEA*, vol. 9, no. 1.4, 2004.
[6] V. Makinen, G. Navarro, J. Siren, and N. Valimaki, "Storage and retrieval of highly repetitive sequence collections," *Journal of Computational Biology*, vol. 17, no. 3, pp. 281–308, 2010.
[7] S. Kreft and G. Navarro, "Self-indexing based on LZ77," in *CPM*. LNCS 6661, 2011, pp. 41–54.
[8] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro, "Compressed q-gram indexing for highly repetitive biological sequences," in *IEEE BIBE*. IEEE Computer Society Press, 2010, pp. 86–91.
[9] K. Schneeberger, J. Hagmann, S. Ossowski, and et al, "Simultaneous alignment of short reads against multiple genomes," *Genome Biology*, vol. 10, p. R98, 2009.
[10] M. C. Brandon and D. C. Wallace, "Data structures and compression algorithms for genomic sequence data," *Bioinformatics*, vol. 25, no. 14, pp. 1731–1738, 2009.
[11] K. Daily, P. Rigor, S. Christley, X. Xie, and P. Baldi, "Data structures and compression algorithms for high-throughput sequencing technologies," *BMC Bioinformatics*, vol. 11, no. 1, p. 514, 2010.
[12] R. Leinonen, H. Sugawara, and M. Shumway, "The sequence read archive," *Nucleic Acids Res.*, vol. 39, no. Database issue, pp. 19–21, Jan 2011.
[13] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval," in *SPIRE*. LNCS 6393, 2010, pp. 201–206.
[14] S. Huang, T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-M. Yiu, "Indexing similar DNA sequences," in *AAIM*, 2010, pp. 180–190.
[15] S. Kuruppu, B. Beresford-Smith, T. Conway, and et al, "Iterative dictionary construction for compression of large DNA datasets," *IEEE ACM Trans Comput Biol Bioinformatics*, vol. 9, no. 1, pp. 137–149, 2011.
[16] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Repetition-based compression of large DNA datasets," in *RECOMB (poster)*, 2009, pp. 91–98.
[17] S. Deorowicz and S. Grabowski, "Robust relative compression of genomes with random access," *Bioinformatics*, vol. 27, no. 21, pp. 2979–2986, 2011.
[18] G. Beskales, M. Fontoura, M. Gurevich, and et al, "Factorization-based lossless compression of inverted indices," in *CIKM*, 2011, pp. 327–332.
[19] P. Papapetrou, V. Athitsos, G. Kollios, and D. Gunopulos, "Reference based alignment in large sequence databases," in *VLDB*, 2009.
[20] J. Venkateswaran, D. Lachwani, T. Kahveci, and et al, "Reference-based indexing of sequence databases," in *VLDB*, 2006, pp. 906–917.