

# Querying Large Similar Sequences in a Compressed Format Efficiently

Xiaochun Yang #, Bin Wang #, Chen Li \*, Xiaohui Xie \*

#School of Information Science and Engineering, Northeastern University  
Liaoning 110819, China  
{yangxc,binwang}@mail.neu.edu.cn

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

**Abstract**—With the advances in next-generation sequencing technologies, the amount of genomic sequence data being produced continues to grow at an exponential rate. A unique characteristic of these sequences is that they are over 99% similar, and therefore highly compressible using their differences with respect to a reference sequence. Still, an increasingly pressing challenge is how to efficiently query these massive amounts of sequence data in their compressed format. In this paper we study the problem of answering subsequence-search queries on a compressed set of long sequences. We develop novel index structures for the differences and algorithms for answering queries. We present various optimization techniques to further reduce the space requirement and query response time. We demonstrate the efficiency of these techniques using a thorough experimental study on real genomic data.

## I. INTRODUCTION

**Motivation:** Recently the 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, the 1000 genomes project (<http://www.1000genomes.org>), which aims at sequencing genomes of 1000 individuals in two years, is generating about 8.2 billion bases per day, the equivalent of more than two human genomes every 24 hours. Indeed, several companies are now actively competing for the “X prize,” which sets the goal of sequencing 10 human genomes per day for the cost of less than \$10,000 per genome. Many scientists believe that we should be able to reach the goal of sequencing a human genome for the cost of less than \$1,000 within a couple of years. The ushering of personal genomics era brings significant computational challenges on how to deal with 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 would be much more efficient to store the differences between each genome and a reference genome rather than the genome itself, because they are very similar (with a similarity of more than 99%) [14]. 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 [3].

**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  $\mathcal{S} = \{S_1, \dots, S_n\}$ , in which each  $S_i$  is very long and stored in a compressed format as follows. Let  $B$  be a base sequence (a.k.a. reference 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  $\Delta_i$  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 subsequence-search queries, each of which specifies a relatively short pattern (e.g., in tens or thousands), and asks for all subsequences in  $\mathcal{S}$  that match the query pattern exactly or approximately.

As an example, consider a hospital setting, where a doctor wants to find out which genomes of 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 [3].

**Our contributions:** In this paper, we develop novel techniques to solve the problem. (1) In Section III we present a tree index (called “transformation tree”) to efficiently store the differences between the sequences and the reference sequence. This index has the advantage of not only storing, but also indexing the sequences using a small amount of space. Furthermore, we can use this index and a proposed gram-based inverted index on the reference sequence to efficiently answer queries. We present an algorithm for answering queries using these indexes. (2) In Section IV we study how to improve query

performance by storing additional information in the indexes. We present an analysis to show that we only need to consider a small subset of the positions in the reference sequence in order to find the answers to a query. We further describe an algorithm to select this subset of positions. (3) In Section V we study how to reduce the size of the inverted index. We show that we can safely discard some of the positions on the reference sequence when constructing the inverted index. The new inverted index, which is significantly smaller, can still be used to compute all the answers to a query. (4) In Section VII we extend the techniques for exact queries to answer approximate subsequence-matching queries. The extended techniques work for commonly used distance metrics, including substitution distance, edit distance, and Smith-Waterman similarity. (5) In Section IX we present experimental results on real data sets to demonstrate the practical space and time efficiency of the proposed techniques.

## II. PRELIMINARIES

### A. Sequences in a Compressed Representation

Let  $\Sigma = \{A, C, G, T\}$  be the alphabet of characters in genomic sequences. For a sequence  $S$  of the characters in  $\Sigma$ , we use  $|S|$  to denote its length,  $S[i]$  to denote its  $i$ -th character (starting from 0), and  $S[i, j]$  to denote the subsequence from its  $i$ -th character to its  $j$ -th character. The typical length of a genomic sequence is from millions to a few billions.

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

Fig. 1 shows an example transformation, consisting of four edit operations. The edit operation  $O_1$  deletes the two characters TA starting at the third position of the base sequence  $B$ . We let  $l(O_1) = -2$  since  $O_1$  reduces the sequence length by 2. The edit operation  $O_2$  inserts a new character C after the 10-th position of  $B$  and  $l(O_2) = 1$ . The edit operation  $O_3$  replaces the 14-th character T with a character C. This substitution operation does not change the length of  $S$ , so  $l(O_3) = 0$ . The edit operation  $O_4$  inserts a new subsequence AC after the 18-th position of  $B$ .  $B[0, 2]$ ,  $B[5, 10]$ ,  $B[11, 13]$ ,  $B[15, 18]$ , and  $B[19, 21]$  are preserved subsequences with respect to  $S_1$ . Correspondingly, the two C characters resulting from operations  $O_2$  and  $O_3$ , and AC from  $O_4$ , are incremental subsequences.

### B. Queries

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

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

**Approximate Subsequence Search.** Given a query sequence  $P$ , find all subsequences 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 following three widely-adopted measures.

*Substitution Distance:* Two sequences  $S_1$  and  $S_2$  are said to have a *substitution distance of  $k$* , also called a “ $k$ -mismatch,” if  $|S_1| = |S_2|$  and  $S_1$  can be transformed into  $S_2$  by substituting  $k$  characters. For instance, given a query TATA and the sequence  $S_1$  in Fig. 1, the corresponding subsequence that has a 1-mismatch is  $S_1[11, 14]$ .

*Edit Distance:* The edit distance between two sequences  $S_1$  and  $S_2$ , denoted by  $ed(S_1, S_2)$ , is the minimum number of single-character insertions, deletions, and substitutions that are needed to transform  $S_1$  to  $S_2$ . When using edit distance, we want to find all the subsequences in the data set whose edit distance to the query pattern is within a given threshold. In Fig. 1, suppose a query pattern  $P = CAGTA$  and the edit-distance threshold is 1. The shown sequence  $S_1$  has two subsequence  $S_1[4, 8]$  and  $S_1[9, 12]$  that are similar to the query pattern, since their edit distances to  $P$  are equal to the threshold 1.

*Smith-Waterman Similarity:* It is a well-known similarity measurement between two sequences  $S_1$  and  $S_2$  [12]. In this function, each identity mapping has a positive score, whereas a gap (insertion of one character or deletion of one character) or a substitution of one character has a negative score. In this paper, we use one of the most common scoring schemes for DNA, which uses  $score(\text{match}) = 1$ ,  $score(\text{substitution}) = -3$ ,  $score(\text{gap}) = -2$ . We did not consider affine gap penalty in this paper for the simplicity of discussion. However, our method can be extended to deal with such a case. The Smith-Waterman similarity between two sequences  $S_1$  and  $S_2$  is denoted by  $sw(S_1, S_2)$ . For example,  $sw(\text{GACTACGG}, \text{ACTACGT}) = 1$ , since the deletion of G at the start position of GACTACGG gets a penalty score of  $-2$ . The replacement of its last character G with T gets a penalty score of  $-3$ , and the identity mappings of ACTACG gets a reward score of  $6 \times 1 = 6$ .

## III. EXACT SUBSEQUENCE SEARCH

In this section, we develop an efficient algorithm for answering subsequence-search queries. We focus on the case of a single sequence  $S$  in  $\mathcal{S}$ . In Section III-A we classify answers into two categories, and study how to find answers in the first category. In III-B we give an algorithm for finding answers. In Section III-C we present index structures needed to implement

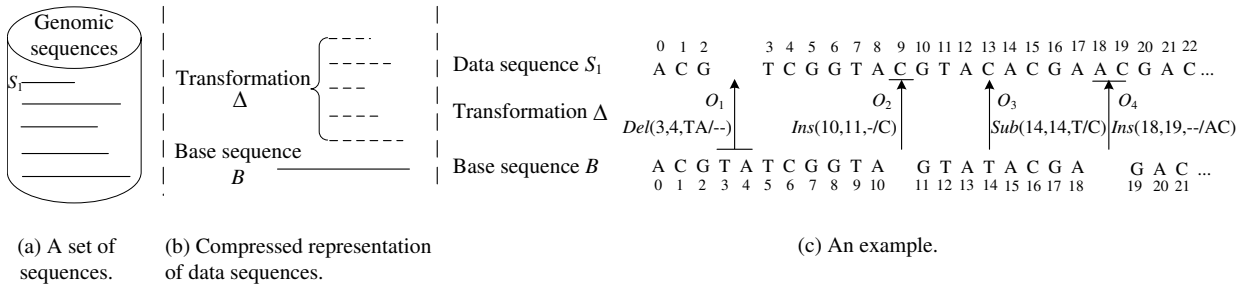


Fig. 1. Genomic sequences represented as differences to a base sequence.

the algorithm. In Section III-D we study how to use the indexes to efficiently support the operations in the algorithm.

### A. Classification of Answers

Suppose  $\mathcal{S}$  has a single data sequence  $S$ . In this case, the problem becomes the following. We have a sequence  $S$  stored as its differences with respect to the base sequence  $B$ , represented as a transformation  $\Delta$  from  $B$  to  $S$ , which includes a set of edit operations. For a query sequence  $P$  (generally  $|P| \ll |B|$ ), we want to find the subsequences of  $S$  that match  $P$  exactly.

Ideally, for every subsequence  $S[a, b]$  matching  $P$ , we hope to find a *seed* in preserved subsequences in  $B$  that could be transformed to a subsequence of  $S[a, b]$  using identity mappings. A seed could be expressed as a  $q$ -gram, where  $q$  is a positive integer. We could decompose  $B$  into a set of *positional  $q$ -grams*, denoted as a pair  $(i, g)$ , where  $g = B[i, i + q - 1]$  ( $0 \leq i \leq |B| - q$ ).

If  $S[a, b]$  is stored physically, at least one seed ( $q$ -gram) in  $S[a, b]$  must match a  $q$ -gram in  $P$ . However, since  $S[a, b]$  is not stored physically, a natural question is whether there exists a matching  $q$ -gram in preserved subsequences in  $B$  if  $S[a, b]$  matches  $P$ . Let  $len(B[c, d])$  be the total number of characters that are mapped to  $S[a, b]$  using identity mappings. Then  $P$  and  $B[c, d]$  should share the following number of seeds:

$$N_c = len(B[c, d]) - h(q - 1),$$

where  $h$  is the number of preserved subsequences in  $B[c, d]$ .

Consider our running example with the query pattern  $P=TACG$ . It has an answer  $S[7, 10]$  that is transformed from  $B[9, 11]$  and an edit operation  $O_2$ .  $P$  and  $B[9, 11]$  shared one gram  $TA$  at position 9 in  $B$ . Based on the above analysis, a subsequence  $S[a, b]$  that matches  $P$  can be in one of two categories. (1) We can find at least one seed in preserved subsequence of  $B$  matching  $P$  if  $N_c > 0$ . (2) Such a seed does not exist. In Section VI we will study how to find such  $S[a, b]$  subsequences in the second category by searching in those incremental sequences.

### B. Algorithm Description

**BASIC Algorithm:** This algorithm computes answers in category (1) by searching in preserved subsequences in  $B$ . It first decomposes the query pattern  $P$  into a set of  $q$ -grams. For each gram  $g_j$  in  $P$ , it finds the occurrences of  $g_j$  in the

base sequence  $B$ . For each such gram  $B[t, t + q - 1]$ , the algorithm ignores the gram if  $\Delta$  modifies some characters in this subsequence. Otherwise, it calculates  $B[t]$ 's corresponding position in  $S$  under  $\Delta$ , denoted by  $\Delta(t)$  and calls a function “VERIFY()” to check if the subsequence starting at  $S[\Delta(t) - j]$  of length  $|P|$  matches the pattern  $P$ .

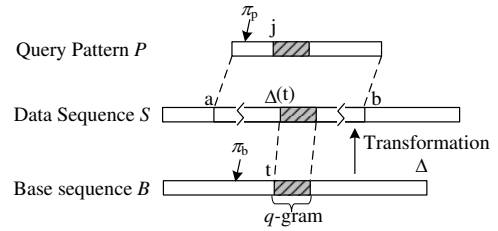


Fig. 2. A subsequence in  $S$  matching the query pattern  $P$  has a matching  $q$ -gram in the base sequence  $B$ .

The main idea of the procedure “VERIFY()” is as follows. We want to check if the subsequence  $S[a, b]$  matches  $P$  exactly (see Fig. 2). Since the original sequence  $S$  is not physically stored, we need to construct the subsequence  $S[a, b]$  on-the-fly using  $B$  and  $\Delta$ , and compare its characters with those in  $P$ . The function returns FALSE whenever it finds a mismatch between a character in  $B$  and the corresponding character in  $S$ .

The algorithm first verifies subsequence  $P[0, j - 1]$  (lines 1 – 21). We use two cursors,  $\pi_p$  and  $\pi_b$ , to represent the positions in  $P$  and  $B$ , respectively, that are being compared. We start by setting  $\pi_p = j - 1$  and  $\pi_b = t - 1$  (lines 1 – 2). For each position  $\pi_b$ , we check whether there is an insertion operation located between  $B[\pi_b]$  and  $B[\pi_b + 1]$  (line 4). If so, we compare the inserted subsequence  $I$  with the corresponding subsequence in  $P$  (lines 5 – 8). We then check  $\Delta$ 's operation on  $B[\pi_b]$  (line 11). We consider three possible cases of this operation, namely identity mapping (lines 12 – 14), substitution (lines 15 – 17), and deletion (lines 18 – 19). We do the corresponding character comparison and adjust the two cursors accordingly. If the verification on the subsequence  $P[0, j - 1]$  passes, we then proceed to verify the subsequence  $P[j + q, |P| - 1]$  starting from  $P[j + q]$  similarly (line 22). The function returns TRUE if all verification steps have passed (line 23).

---

**Algorithm 1: BASIC** – Subsequence search.

---

**Input:** A query sequence  $P$ , a base sequence  $B$ , a sequence  $S$  stored as a transformation  $\Delta$  from  $B$  to  $S$ ;  
**Output:** Starting positions of subsequences of  $S$  matching  $P$ ;  
1 Result set  $R = \emptyset$ ,  $R' = \emptyset$ ;  
  // Category (1)  
2 Decompose  $P$  to  $q$ -grams;  
3 **foreach**  $q$ -gram starting at  $B[t]$  matching a  $q$ -gram starting at  $P[j]$   
  **do**  
4   **if**  $\Delta$  does not modify  $B[t, t + q - 1]$  **then**  
5      $\Delta(t) = B[t]$ 's target position in  $S$  under  $\Delta$ ;  
6     **if**  $\text{VERIFY}(P, j, B, t, \Delta, \Delta(t))$  **then**  
7        $R = R \cup \{\Delta(t) - j\}$ ;  
  // Category (2)  
8  $R' =$  search incremental subsequences (see Section VI);  
9 **return**  $R \cup R'$ ;

---

**Function VERIFY**

---

**Input:** A query sequence  $P$ , a position  $j$  on  $P$ , a base sequence  $B$ , a position  $t$  on  $B$ , a transformation  $\Delta$  from  $B$  to  $S$ ,  $B[t]$ 's corresponding position  $\Delta(t)$  in  $S$  under  $\Delta$ ;  
**Output:** TRUE if subsequence  $S[\Delta(t) - j, \Delta(t) - j + |P| - 1]$  matches  $P$ , and FALSE otherwise;  
  // verify subsequence  $P[0, j - 1]$   
1  $\pi_p = j - 1$ ; // set cursor on  $P$   
2  $\pi_b = t - 1$ ; // set cursor on  $B$   
3 **while**  $\pi_p \geq 0$  and  $\pi_b \geq 0$  **do**  
4   **if**  $\Delta$  inserts a subsequence  $I$  immediately after  $B[\pi_b]$  **then**  
5      $\alpha = |I| - 1$ ;  
6     **while**  $\pi_p \geq 0$  and  $\alpha \geq 0$  **do**  
7       **if**  $P[\pi_p] \neq I[\alpha]$  **then** **return** FALSE;  
8        $\pi_p --$ ;  $\alpha --$ ;  
9     **if**  $\pi_p == -1$  **then**  
10      **continue**; // no letters left in  $P$   
11   **switch**  $\Delta$ 's operation on character  $B[\pi_b]$  **do**  
12     **case** Identity mapping:  
13       **if**  $P[\pi_p] \neq B[\pi_b]$  **then** **return** FALSE;  
14        $\pi_p --$ ;  $\pi_b --$ ;  
15     **case** Substitution:  
16       **if**  $P[\pi_p] \neq$  the new character **then** **return** FALSE;  
17        $\pi_p --$ ;  $\pi_b --$ ;  
18     **case** Deletion:  
19        $\pi_b --$ ;  
20 **if**  $\pi_p <> -1$  **then**  
21   **return** FALSE; // no letters left in  $B$   
22 **verify**  $P[j + q, |P| - 1]$  similarly;  
23 **return** TRUE;

---

### C. Storing Transformation and Indexing

In this section we discuss how to store the transformation  $\Delta$  and how to index the base sequence  $B$ .

**Inverted Index of Grams:** We build an inverted index of  $q$ -grams for the base sequence  $B$ . For each  $q$ -gram in  $B$ , we generate an inverted list of positions in  $B$ . For instance, in the running example, the list of the gram CG includes three matching positions “1”, “6”, and “16” of the grams in  $B$ .

**Indexing the transformation:** We use a variant of  $B^+$ -tree, called *transformation tree*, to store the transformation  $\Delta$ . Fig. 3 shows an example transformation tree. We use the end positions of edit operations as the values to build the tree. All information of changes is stored at the leaf nodes. Each leaf node stores the following transformation information for an edit operation: (1) its start and end positions in  $B$ ; (2) the type of operation, and (3) changes made by each operation. For instance, node  $n_3$  is a leaf node with two key values 4 and 11,

and two records with respect to these two keys. The key 4 is the end position of the deleted subsequence TA. In its record, we maintain the start position 3 of the deleted subsequence, the type of this operation (“DEL”), and the corresponding change “TA/--”. Two adjacent leaf nodes have pointers to each other.

For each downward edge, we store the length difference between  $B$  and  $S$  due to the operations stored in its subtree, which is called an *offset difference*. The number on the edge to the tree root (“57” in the figure) is the length difference between  $B$  and  $S$ . For example, the edge from node  $n_3$  to the record with a key value “4” stores an offset difference “ $l(O_1) = -2$ ”, which is the length difference due to  $O_1$  in Fig. 1.

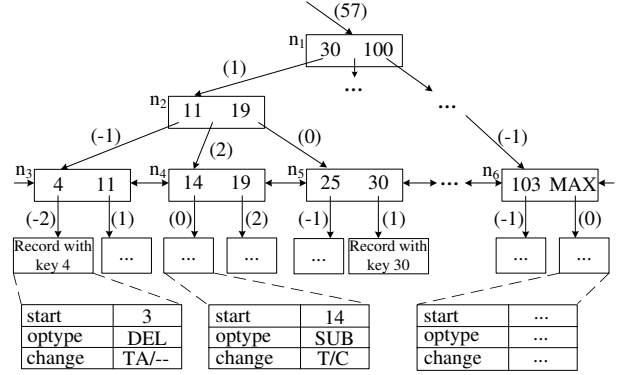


Fig. 3. A transformation tree.

### D. Efficient Lookups

The index structures could support the following operations needed in the BASIC algorithm.

**Gram lookups:** For a  $q$ -gram in a query pattern  $P$ , find all grams in  $B$  that match the gram. This operation can be done efficiently using the inverted index of the grams.

**Neighboring edit-operation lookups:** Given a position  $\pi_b$  on  $B$ , we want to find its left and right immediate positions that contain an edit operation. This operation can be done efficiently using the transformation tree. Suppose we want to find its left immediate position with an edit operation. On the tree we can just find the record with the largest ending position less than  $\pi_b$ . For instance, suppose  $\pi_b = 12$  (see Fig. 1). The largest ending position that is less than 12 is the ending position 11 in node  $n_3$ . Similarly, if  $\pi_b = 15$ , then the largest ending position that is less than 15 is the ending position 14 in node  $n_4$ .

**Calculating  $\Delta(t)$ :** Given a position  $t$  in  $B$  whose character is not modified by  $\Delta$ , we want to calculate the corresponding position  $\Delta(t)$  in  $S$ .  $\Delta(t)$  is the length difference between  $B[0, t]$  and  $S[0, \Delta(t)]$ , which could be calculated by accumulating length differences due to the edit operations from position 0 to  $t$  in  $B$ .

To do it, we compute its left immediate position with an edit operation using the transformation tree (using the method described above). Let  $n_l$  be the corresponding leaf node. For each node  $n_a$  on the path from the root to this node  $n_l$

(excluding  $n_l$ ), we take the summation of the offset differences of the left siblings of  $n_a$ . We also add the offset difference of  $n_l$ . Let the final summation be  $d$ . Then  $\Delta(t) = t + d$ . For example, consider the tree in Fig. 3. Let  $t = 15$ . We find the record with value 14, which is largest among those key values less than 15. Consider the path from the root to this record, and let  $d$  be the summation of the offset differences of  $n_3$  (left sibling of node  $n_4$ ) and record 14. That is,  $d = (-1) + 0 = -1$ . Thus  $\Delta(15) = 15 + (-1) = 14$ . Notice that we can compute and store this number for each leaf node to save the computation in the future.

#### IV. REDUCING MATCHING GRAMS TO BE VERIFIED

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. The performance of BASIC depends on the number of matching grams in  $B$  that need to be verified. The time complexity of BASIC approach is  $O(l \cdot \log w + l \cdot m)$ , where  $l$  is the number of positions in all matching gram lists,  $w$  is the number of operations in the transformation tree, and  $m$  is the length of  $P$ . We use  $O(l \cdot \log w)$  to calculate  $\Delta(t)$  and  $O(l \cdot m)$  to do verifications.

Often an answer subsequence in  $S$  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 `ACGT`. The two grams  $g_0$  and  $g_1$  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  $B[0, 5]$ . Our goal is to avoid this type of *duplicate verification*.

A straightforward way to avoid duplicate verification is to first create a list of candidate positions  $\Delta(t) - j$  for each matching gram  $g_j$  at position  $t$  and then eliminate duplicate positions on them before verifying. We could use a classic heap-based algorithm to eliminate duplicate positions. The time complexity of this straightforward approach is  $O(l \cdot \log w + l \cdot m \log m + r \cdot m)$ , where  $r (\ll l)$  is the number of candidate positions. We need to use time of  $O(l \cdot \log w)$  to calculate  $\Delta(t)$ , time of  $O(l \cdot m \log m)$  to eliminate duplicate positions, and time of  $O(r \cdot m)$  to verify candidates. Obviously, eliminating duplicates makes this approach costly.

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 small amount of additional information, we can improve search performance significantly.

In Section IV-A, we study the case where there is a single data sequence in  $S$ , and present an optimization that can avoid duplicate verifications. In Section IV-B we study to find a minimal number of matching-gram positions to find answers.

##### A. Avoiding Duplicate Verifications

Suppose we have a single data sequence  $S$ . Our first optimization is to avoid duplicate verifications that generate the same answer subsequence from different matching grams.

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

*Lemma 1:* In the algorithm BASIC, we can find the same answer subsequences by only considering

- $q$ -grams of  $B$  that match  $g_0 = P[0, q - 1]$ , and
- head grams of  $B$  that match gram  $g_i$  of  $P$  for any  $i > 0$ .

*Proof:* Consider the gram  $g_j = P[j, j + q - 1]$  in the query pattern  $P$  (see Fig. 4). Suppose it matches the gram at position  $k$  in  $B$ , and the corresponding verification step produces an answer in the data sequence  $S$ . If  $j > 0$ , and there is no edit operation on character  $B[k - 1]$ , then the immediate left gram  $g_{j-1}$  matches the gram  $B[k - 1]$  as well. Furthermore, the verification step of  $g_{j-1}$  for  $B[k - 1]$  will produce the same answer in  $S$ . We can generalize this observation to  $B[k]$ 's *leftmost* gram  $B[h]$ , such that  $k - h \leq j$ , and there is no edit operation (including insertions) on the characters between  $B[h]$  and  $B[k]$ . There are two cases:

- $k - h = j$ : In this case, the corresponding gram  $g_{j - (k - h)}$  is  $g_0$ , i.e., the first gram in the query pattern  $P$ .
- $k - h < j$ : In this case, since  $g_h$  is the leftmost gram meeting the requirement above, we know the head gram  $B[h, h + q - 1]$  has a unique property: there is an edit operation immediately before this head gram, i.e., a deletion/substitution on  $B[h - 1]$  or an insertion between  $B[h - 1]$  and  $B[h]$ .

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

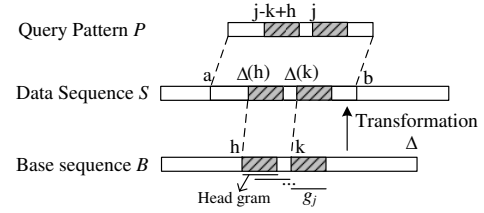


Fig. 4. Intuition of ignoring matching grams.

**H\_VERIFY algorithm:** Based on Lemma 1, we can modify the algorithm BASIC as follows. We call the verification procedure for (1) each gram in  $B$  that matches the  $g_0$  gram in the query pattern; and (2) each head gram in  $B$  that matches a gram  $g_i$  in  $P$ , where  $i > 0$ . All other matching grams can be ignored. We call this modified algorithm “H\_VERIFY”, where “H” stands for “Head.” For instance, the head grams are those at  $B[0]$ ,  $B[5]$ ,  $B[11]$ ,  $B[15]$ , and  $B[19]$  (We set the first gram in  $B$  a head gram) in Fig. 1. Consider the query pattern  $P = \text{ACGT}$ . To find the answers, we only need to consider the following grams: (1) those grams matching  $g_0 = \text{AC}$ , i.e., the grams at  $B[0]$ ,  $B[15]$ , and  $B[20]$ ; (2) those head grams

matching  $g_1 = \text{CG}$  or  $g_2 = \text{GT}$ . For  $g_1$ , it has three matching grams at  $B[1]$ ,  $B[6]$  and  $B[16]$ . We could ignore all these three grams since none of them is a head gram. For  $g_2$ , it has three matching grams at  $B[2]$ ,  $B[8]$  and  $B[11]$ , of which we only need to consider the head gram  $B[11]$ .

*Finding head grams:* For each gram  $g_i$  ( $i > 0$ ) of the query pattern, we want to find its matching grams that are head grams. To achieve this goal, we modify the inverted lists by adding a flag to each position: a flag of “1” means that this gram is a head gram, and a flag of “0” means it is not. Fig. 5 shows the modified inverted lists for the base sequence  $B$  in Fig. 1. The list of gram  $\text{GT}$  contains three flagged positions  $\langle 2, 0 \rangle$ ,  $\langle 8, 0 \rangle$ ,  $\langle 11, 1 \rangle$ , indicating that the first two grams are not a head gram, while the last one is.

Grams	Positions in $B$	Grams	Positions in $B$
AC	$\langle 0, 1 \rangle$ , $\langle 15, 1 \rangle$ , $\langle 20, 0 \rangle$	GA	$\langle 17, 0 \rangle$ , $\langle 19, 1 \rangle$
AG	$\langle 10, 0 \rangle$ , $\langle 18, 0 \rangle$	GT	$\langle 2, 0 \rangle$ , $\langle 8, 0 \rangle$ , $\langle 11, 1 \rangle$
AT	$\langle 4, 0 \rangle$ , $\langle 13, 0 \rangle$	TA	$\langle 3, 0 \rangle$ , $\langle 9, 0 \rangle$ , $\langle 12, 0 \rangle$ , $\langle 14, 0 \rangle$
CG	$\langle 1, 0 \rangle$ , $\langle 6, 0 \rangle$ , $\langle 16, 0 \rangle$	TC	$\langle 5, 1 \rangle$
GG	$\langle 7, 0 \rangle$		

Fig. 5. Inverted lists of  $q$ -grams with head-gram flags.

## B. Minimizing Matching Grams to Verify

Let  $|g|$  denote the size of the inverted list of gram  $g$ , and  $H(g_i)$  denote the set of head grams in  $B$  matching  $g_i$  of query  $P$ . Then the number of matching grams that need to be verified in the algorithm  $\text{H\_VERIFY}$  is equal to the sum of  $|g_0|$  and the size of  $H(g_i)$  for each  $i > 0$ , i.e.,  $|g_0| + \sum_{i=1}^{|P|-q} |H(g_i)|$ . If the inverted list of  $g_0$  is long, there are still many grams that need to be verified. A question arises naturally: can we avoid accessing  $g_0$  if it is very long? Next we give an analysis that suggests a positive answer. We can generalize the observation in Section IV-A and replace  $g_0$  with an arbitrary gram  $g_s$ . In particular, we need to decide, in addition to accessing and verifying the grams on the list of  $g_s$ , which other grams on other gram lists need to be verified in order to find all the answer subsequences. Similar to the concept of “head grams,” we introduce the concept of “tail grams.”

*Definition 2: (Tail Grams)* A gram  $B[t, t + q - 1]$  in the base sequence  $B$  is called a *tail gram* with respect to the transformation  $\Delta : B \rightarrow S$  if there is an edit operation immediately *after*  $B[t + q - 1]$ , i.e.,  $\Delta$  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 2:* Let  $g_s$  be an arbitrary gram in  $P$ . In the algorithm  $\text{BASIC}$ , we can find the same answer subsequences by only considering the following grams:

- matching grams for the gram  $g_s$ ,
- head grams matching a gram  $g_i$  ( $i > s$ ), and
- tail grams matching a gram  $g_i$  ( $0 \leq i < s$ ).

To see why Lemma 2 is true, consider an answer subsequence in  $S$  that matches the query pattern  $P$ . Let  $g_j$  be a query gram matching a gram  $B[k]$  in the base sequence  $B$ , such that the corresponding verification can produce this answer subsequence (see the assumption in Section III-B). There are three cases:

- $j = s$ : In this case,  $B[k]$  is a matching gram for the gram  $g_s$ .
- $j > s$ : Consider  $B[k]$ ’s *leftmost* gram  $B[h]$ , 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  $B[h]$  is a matching gram of  $g_s$ . Otherwise,  $B[h]$  is a head gram matching  $g_{j-(k-h)}$ .
- $j < s$ : Consider  $B[k]$ ’s *rightmost* gram  $B[t]$ , 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  $B[h]$  is a matching gram of  $g_s$ . Otherwise,  $B[t]$  is a tail gram matching  $g_{j+(t-k)}$ .

Clearly Lemma 1 is a special case of Lemma 2 where  $s = 0$ . The number of matching grams that need to be verified is:

$$|g_s| + \sum_{i=0}^{s-1} |T(g_i)| + \sum_{i=s+1}^{|P|-q} |H(g_i)|, \quad (1)$$

where “ $T(g_i)$ ” is the set of tail grams on the list of gram  $g_i$ . We want to choose an optimal gram  $g_s$  that can minimize the above summation.

In order to find tail and head grams, we modify the inverted lists by adding two bits for each positional gram: one bit indicating whether the positional gram is a head gram, and one indicating whether it is a tail gram. For example, consider the location  $\langle 5, \underline{10} \rangle$  of the  $g_2$  gram  $\text{TC}$  in Fig. 6. It means that the gram  $\text{TC}$  at position 5 is a head gram but not a tail gram.

id	grams	lists of matching grams
$g_0$	CG $\rightarrow$	$\langle 1, \underline{01} \rangle$ , $\langle 6, \underline{00} \rangle$ , $\langle 16, \underline{00} \rangle$
$g_1$	GT $\rightarrow$	$\langle 2, \underline{00} \rangle$ , $\langle 8, \underline{00} \rangle$ , $\langle 11, \underline{10} \rangle$
$g_2$	TC $\rightarrow$	$\langle 5, \underline{10} \rangle$

Fig. 6. Query  $\text{CGTC}$ , its inverted lists with bits of head grams and tail grams, and an optimal gram  $g_2$ .

**MIN\_VERIFY algorithm:** Based on Lemma 2, we modify the algorithm  $\text{BASIC}$  to a new algorithm, called  $\text{MIN\_VERIFY}$ , which uses a minimal number of matching grams to do subsequence search.  $\text{MIN\_VERIFY}$  chooses an optimal gram  $g_s$  with a minimal number of matching grams. We call the verification procedure for each gram that satisfies one of the three conditions in Lemma 2. All other matching grams can be ignored.

For example, in Fig. 6, if we choose  $g_2$  as the optimal gram, we only need to verify locations  $\langle 1, \underline{01} \rangle$  on the  $g_0$  list and  $\langle 5, \underline{10} \rangle$  on the  $g_2$  list. In particular,  $\langle 1, \underline{01} \rangle$  is a tail gram matching the gram  $g_0$ , and  $\langle 5, \underline{10} \rangle$  is a matching gram for the optimal gram  $g_2$ . Therefore, the total number of positions to be verified is 2. Compared to the approach that chooses  $g_0$  as the optimal gram, we have 3 fewer positions to verify.

## V. REDUCING INDEX SIZE

In this section, we study how to reduce the 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 show that if there is a lower bound on query lengths, we can discard certain positions

on the inverted lists while we can still use the remaining positions to answer queries. We develop a compression technique that can significantly reduce the inverted-index size.

### A. Discarding Positional Grams

Suppose the collection  $\mathcal{S}$  contains a single data sequence  $S$ . Given a query  $P$ , we choose a gram  $g_s$  of  $P$  and do the corresponding verifications using the method described in Section IV-B. That is, we verify all the positional grams on the  $g_s$  list, the tail grams on the list of each gram  $g_i$  ( $i < s$ ), and the head grams on the list of each gram  $g_j$  ( $j > s$ ), as shown in Fig. 7(a). A gram is called a *regular gram* if it is neither a head gram nor a tail gram. Since most regular grams do not need to be verified, a natural question is: which of the regular grams can be discarded while the remaining grams can still be used to answer queries?

The following analysis shows that if we are not cautious about which regular grams to remove, we could miss some answers to a query. Let  $r$  be a position on the inverted list of the gram  $g_s$ , where the gram at position  $r$  in  $B$  is a regular gram. Let  $[a, b]$  be the subsequence in  $B$  that contains the position  $r$ , whose corresponding target subsequence in  $S$  under  $\Delta$  is an answer to the query  $P$ . We call  $[a, b]$  a “candidate subsequence” of  $B$ . Suppose there is no gram  $g_i$  ( $i < s$ ) with a matching tail gram in  $[a, b]$ , nor a gram  $g_j$  ( $j > s$ ) with a matching head gram in  $[a, b]$ . In this case, if we remove the positional gram at  $r$  from the inverted list of  $g_s$ , then we will miss the chance of doing the verification to find this answer!

To decide which positional grams can be discarded safely, we introduce a new kind of grams, called *anchor grams*, so that each candidate subsequence 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  $B$ . Let  $p$  be the start position of the sliding window. Initially,  $p = 0$ . For each  $p$  position, we check the subsequence  $B[p, p + L - 1]$ . If there is no edit operation in this subsequence, we choose the last gram in the subsequence as an anchor gram. The benefit is that when we slide the window to the next position  $p + 1$ , this anchor gram is still inside the new window.

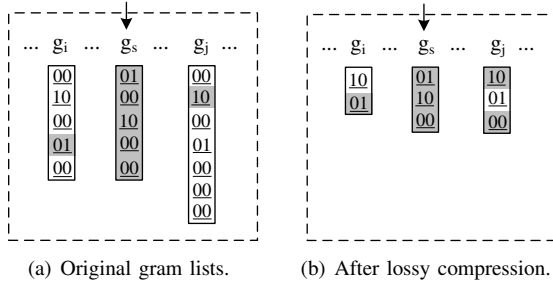


Fig. 7. Compressing gram lists. “00” represents a regular gram, “10” represents a head gram, “01” represents a tail gram, and shaded grams need to be verified. The remaining regular grams in (b) are anchor grams.

If the transformation  $\Delta$  has an operation  $O$  in the current sliding window, then there must be a tail gram in the window.

We change the window size to  $L - l(O)$ , where  $l(O)$  is the offset difference between  $B$  and  $S$  due to this operation  $O$ . There are two possible cases. (1) Case 1: the new window has a head gram immediately after the position of this operation  $O$ . In this case, we do not need to set an anchor gram inside this window. The reason is that no matter which gram in the query is chosen as the  $g_s$  gram, there will be a tail gram or a head gram inside this window that needs to be verified. (2) Case 2: The next head gram is not in this window. Then we set the tail gram as an anchor gram. The goal is to make sure that this tail gram can always be chosen to do verification. When we move the sliding window to a new position, such that it does not contain the previous tail gram, but contains the next head gram and those after this 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 (see Fig. 7(b)).

### B. Search Using Compressed Inverted Index

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

*Lemma 3:* Consider the algorithm BASIC. Let  $g_s$  be the selected gram of the query pattern  $P$ . We can find the same answers by only considering the following grams:

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

Using the algorithm on the compressed inverted index, we need to verify the following number of matching grams:

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

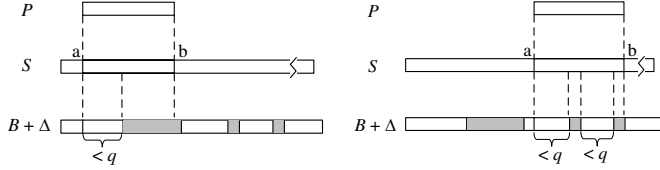
where “ $A(g_i)$ ” is the set of anchors on the list of gram  $g_i$ .

**A\_VERIFY algorithm:** We modify the algorithm MIN\_VERIFY slightly by using the compressed inverted index. We first choose an optimal gram  $g_s$  using Equation 2. Based on the above analysis, for each positional gram on the inverted list of  $g_s$ , we need to do the corresponding verification for each data sequence. In addition, for each gram  $g_i$  ( $i < s$ ), we consider the tail grams and anchor grams on the inverted list of  $g_i$  and do the corresponding verification. Furthermore, for each gram  $g_j$  ( $j > s$ ), we consider all the head grams and anchor grams on the inverted list of  $g_j$  and do the corresponding verification. We call this modified algorithm “A\_VERIFY”, where “A” stands for “Anchor.”

## VI. SEARCHING IN INCREMENTAL SUBSEQUENCES

In Section III we showed that answers to a subsequence-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  $S[a, b]$  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 subsequences to find answers. As illustrated in Fig. 8(a), there are two possible cases for such an answer  $S[a, b]$ . For the case in Fig. 8(a), if  $S[a, b]$  matches  $P$ , then  $S[a + q - 1, b - q + 1]$  must match a subsequence of an incremental subsequence. For the case in Fig. 8(b), we could not find a subsequence  $S[a, b]$  unless we uncompress the whole data sequence  $S$  using  $B$  and  $\Delta$ . This approach is computationally expensive.



(a) A matching subsequence in a long incremental subsequence. (b) A matching subsequence mapped from adjacent edit operations.

Fig. 8. Matching subsequences using  $B$  and  $\Delta$  (white areas in  $B$  represent preserved subsequences and grey areas represent incremental subsequences).

In order to avoid the case in Fig. 8(b), we preprocess the data sequence to change short preserved subsequences (with a length less than  $q$ ) to incremental subsequences. We first extend the definition of *incremental subsequence* by allowing identical-character matching in its edit operations. In the preprocessing step, we combine adjacent incremental subsequences into a single incremental subsequence. In addition, for each short preserved subsequence  $X$  with a length less than  $q$ , we combine it with its adjacent edit operations to form a new incremental subsequence, and will no longer treat  $X$  as a preserved subsequence. For example, consider our running example in Fig. 1. Let  $q = 4$ , we combine  $B[11, 13]$  and  $O_2$  and  $O_3$  into one incremental subsequence 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 4:** After preprocessing the data sequences, for every subsequence  $S[a, b]$  matching a query pattern  $P$ , there is either a matching  $q$ -gram of the answer in a preserved subsequence to match a  $q$ -gram in  $P$  or a subsequence of an incremental subsequence 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 subsequence is  $\geq q$ . Suppose the query  $P$  has an answer subsequence  $G$ . If  $G$  has a subsequence of length  $q$  that can be mapped into a preserved subsequence, then the BASIC algorithm can find this answer. Otherwise, since each preserved subsequence has a length  $\geq q$ , then  $G$  must be matching an incremental subsequence and its adjacent preserved subsequences (at most two). Each of the adjacent preserved subsequences can have most  $q - 1$  characters matching  $G$ , thus  $G$  should have at least  $|P| - 2(q - 1)$  characters matching the incremental subsequence. Such an incremental subsequence can be found by the above procedure. ■

Algorithm 2 shows that we need to check those incremental

subsequences with a length  $\geq |P| - 2(q - 1)$ . Let  $R$  be such an incremental subsequence. We need to check whether  $P[q - 1, |P| - q]$  is a subsequence 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 subsequences.

---

**Algorithm 2:** Search in incremental subsequences.

---

**Input:** A query sequence  $P$ , a base sequence  $B$ , a sequence  $S$  stored as a transformation  $\Delta$  from  $B$  to  $S$ ;  
**Output:** Starting positions of subsequences of  $S$  matching  $P$ ;

```

1 Result set  $R = \emptyset$ ;
2 foreach Incremental subsequence  $R$  with length  $\geq |P| - 2(q - 1)$  do
3   if  $P[q - 1, |P| - q]$  is a subsequence of  $R$  then
4      $t =$  start position of the edit operation w.r.t.  $R$ ;
5     if  $R$  is a subsequence of  $P$  at position  $x$  then
6        $R = R \cup \{\Delta(t) + x\}$ ;
7     if  $P[q - 1, |P| - 1]$  matches  $R[0, |P| - q]$  then
8       if  $\text{VERIFY}(P[0, q - 2], 0, B, t, \Delta, \Delta(t))$  then
9          $R = R \cup \{\Delta(t)\}$ ;
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  $\text{VERIFY}(P[|P| - q + 1, |P| - 1], |P| - q + 1, B, t', \Delta, \Delta(t'))$  then
13         $R = R \cup \{\Delta(t) + y\}$ ;
14 return  $R$ ;
```

---

It is interesting to note that the real data sets we tested contain only a few incremental subsequences have lengths  $\geq |P| - 2(q - 1)$  (we will report it in Section IX-C). Therefore, the above checking can be done very efficiently.

## VII. APPROXIMATE SUBSEQUENCE SEARCH

In this section, we extend our exact-matching techniques to support approximate subsequence matching, i.e., finding subsequences in the data sequences similar to a given query pattern.

**Substitution distance:** Using this function, the goal is to find all subsequences of the data sequence whose substitution distance to a query pattern  $P$  is less than or equal to a threshold  $k$ , where  $k \ll |P|$ . Notice that an answer subsequence must have the same length  $|P|$ . We can find answers by slightly modifying the “VERIFY()” function in Algorithm 1 as follows. In each verification step, we keep track of the number of mismatches between the query subsequence and the corresponding subsequence in the data sequence. We return this subsequence in  $S$  only if the number of mismatches is less than or equal to the given threshold.

**Edit distance:** The goal is to find subsequences whose edit distance to  $P$  is less than or equal to a threshold  $\tau$ , where  $\tau \ll |P|$ . The problem is more challenging than the case of using substitution distance since the length of an answer subsequence could be different from  $|P|$ . One way to compute answers is the following. For each gram in  $B$  that matches a query gram, we use the transformation tree to construct the corresponding subsequence of  $S$ . In order to get all answers, let the length of this subsequence be  $|P| + 2\tau$ . We then run

a classic dynamic programming algorithm to verify if there are substring of this subsequence and the query  $P$  have edit distance less than or equal to  $\tau$ . The time complexity of this checking is  $O(|P|^2 + 2\tau|P|)$ . A limitation of this method is that we need to run the dynamic programming algorithm many times, which can be expensive.

We can improve the method by adopting the well-known pigeon-hole principle to split  $P$  to  $\tau + 1$  disjoint pieces. For each piece, we search its exact matches in the reference sequence using our proposed exact subsequence search methods described above. For each found subsequence, we do the corresponding verification of its adjacent characters to verify if the corresponding subsequence matches the pattern  $P$  under the given distance threshold.

**Smith-Waterman similarity:** Using Smith-Waterman similarity function, the goal of approximate subsequence search is to find all subsequences of each data sequence in  $\mathcal{S}$  whose Smith-Waterman similarity to a query pattern  $P$  is greater than or equal to a threshold  $\gamma$ .

In order to compute answers efficiently, we want to transform this problem to the problem of approximate subsequence search using the edit-distance threshold. Now we analyze the relationship between these two problems. Given a query pattern  $P$ . Let  $S[a, b]$  be a subsequence that could answer  $P$  under a Smith-Waterman threshold  $\gamma$ . We use the following variables to define the number of different single-character edit operations that are needed to transform  $P$  to  $S[a, b]$ :

- let  $y$  ( $\geq 0$ ) be the number of matchings from  $P$  to  $S[a, b]$ ,
- let  $x_s$  ( $\geq 0$ ) be the number of substitutions from  $P$  to  $S[a, b]$ ,
- let  $x_d$  ( $\geq 0$ ) be the number of deletions from  $P$  to  $S[a, b]$ , and
- let  $x_i$  ( $\geq 0$ ) be the number of insertions from  $P$  to  $S[a, b]$ .

Therefore, we get the following equations:

$$|P| = y + x_s + x_d; \quad (3)$$

$$\tau = x_s + x_d + x_i; \quad (4)$$

$$\gamma \leq \text{score}(\text{match}) \times y + \text{score}(\text{gap}) \times (x_i + x_d) + \text{score}(\text{substitute}) \times x_s \quad (5)$$

From Inequation 5 and Equation 4, we get

$$\begin{aligned} \text{score}(\text{match}) \times y - \gamma &\geq |\text{score}(\text{gap})| \times (x_i + x_d) \\ &\quad + |\text{score}(\text{substitute})| \times x_s \\ &\geq \min\{|\text{score}(\text{gap})|, |\text{score}(\text{substitute})|\} \times (x_i + x_d + x_s) \\ &\geq \min\{|\text{score}(\text{gap})|, |\text{score}(\text{substitute})|\} \times \tau \end{aligned}$$

Therefore,

$$\tau \leq \frac{\text{score}(\text{match}) \times y - \gamma}{\min\{|\text{score}(\text{gap})|, |\text{score}(\text{substitute})|\}} \quad (6)$$

From Equation 3, we know  $|P| \geq y$ . Then,

$$\begin{aligned} \tau &\leq \frac{\text{score}(\text{match}) \times |P| - \gamma}{\min\{|\text{score}(\text{gap})|, |\text{score}(\text{substitute})|\}} \\ &\leq \lceil \frac{\text{score}(\text{match}) \times |P| - \gamma}{\min\{|\text{score}(\text{gap})|, |\text{score}(\text{substitute})|\}} \rceil \quad (7) \end{aligned}$$

Based on the above calculation, we could transform the problem of approximate subsequence search under a Smith-Waterman threshold  $\gamma$  to the problem of approximate subsequence search under the following edit-distance threshold:

$$\tau \leq \lceil \frac{\text{score}(\text{match})}{\min\{|\text{score}(\text{gap})|, |\text{score}(\text{substitute})|\}} \times (|P| - \lfloor \frac{\gamma}{\text{score}(\text{match})} \rfloor) \rceil.$$

We use this edit-distance threshold to compute candidate subsequences, and verify them by computing their Smith-Waterman similarity. In our experiments, the edit-distance threshold was  $\lceil \frac{1}{2} \times (|P| - \gamma) \rceil$ .

## VIII. EXTENSION TO MULTIPLE SEQUENCES

Suppose  $\mathcal{S}$  consists of  $n$  data sequences,  $S_1, \dots, S_n$ . Again, we use  $B$  to denote the base sequence. Each  $S_i$  is stored as its differences from the same base sequence  $B$ . We want to answer the following type of queries: given a query pattern, find all subsequences in each of the sequences in  $\mathcal{S}$  that match the query pattern. In this section, we describe how to extend the index structures and the corresponding search algorithms.

### A. Extending Index Structures

We extend the transformation tree as follows. We use a single transformation tree to store the differences between each data sequence and the base sequence. Each leaf node stores an extra key that points to an array of records. For example, consider the first record with the key value 4 pointed by the node  $n_3$  in Fig. 9. The record stores a deletion operation at an interval  $[3, 4]$  for  $S_1$  (see Fig. 1) and a substitution operation at position 4 for  $S_2$ .

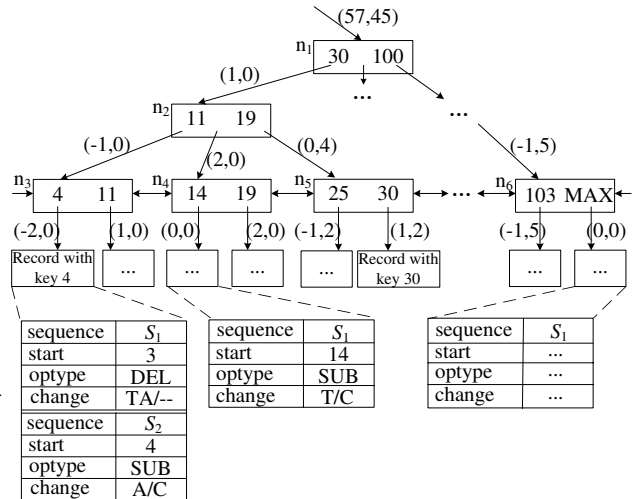


Fig. 9. A transformation tree for two sequences  $S_1$  and  $S_2$ .

On each edge of the tree, we store a list of offset differences, each of which corresponding to a base sequence  $S_i$ . The edge also stores the offset difference between  $S_i$  and the base sequence, i.e., the length difference between  $B$  and  $S_i$  due to the operations of  $S_i$  in this subtree.

Notice that a simple way to extend the transformation tree is to simply create a transformation tree for each sequence. We can then run our search algorithm on each of them. Compared to this solution, the “single-tree-for-all” solution described above is more efficient in both time and space, as shown in the experiments in Section IX-G.

In Fig. 9, consider the edge below the key “25” in node  $n_5$ , with two numbers “(-1, 2)”. The number “-1” is the offset difference between  $B$  and  $S_1$ , i.e., the length difference between  $B$  and  $S_1$ . It means that the corresponding operations of  $S_1$  in this subtree reduce the sequence length by 1. Similarly the number “2” is the offset difference between  $B$  and  $S_2$ , meaning that the operations of  $S_2$  in this subtree increase the sequence length by 2.

We also extend the inverted index to keep track of information about head grams and tail grams for the data sequences. For each position in a gram list and for each  $S_i$ , we use two bits to annotate the type of the gram.

In the case where we use the compressed index described in Section V-A, we modify the inverted index as follows. For each position of the inverted list of a gram and for each data sequence  $S_i$ , if the gram at this position is an anchor gram for this sequence, we set the two bits for this sequence to be 11.

If a position annotated 00 for each of the data sequences  $S_1, \dots, S_n$ , then we remove this position from the inverted index.

### B. Extending Search Algorithms

For a query pattern  $P$ , we want to find all subsequences in the data sequences that match  $P$ . We can extend the algorithm BASIC as follows. In line 4, for each matching gram at position  $t$  in  $B$ , we find the corresponding leaf node of position  $t$  using the transformation tree. We then need to add an inner loop to do the verification for every sequence  $S_i$  by looking forward and backward at the leaf node level. For the verification for the data sequence  $S_i$ , we return the value “TRUE” if  $S_i[\Delta_i(t) - j, \Delta_i(t) - j + |P| - 1]$  matches  $P$ , and “FALSE” otherwise. We can use the transformation tree to do the verification efficiently. In particular, we do the lookup operations on the single transformation tree as follows.

(1) *Finding neighboring edit-operation position on  $B$  for  $S_i$ :* Given a position  $\pi_b$  on  $B$ , we want to find its left and right immediate positions with an edit operation for  $S_i$ . Suppose we want to find its left position with an edit operation. On the tree we can find the record with the largest ending position less than  $\pi_b$ . If we could find  $S_i$  in this record, we return the end position of this record. Otherwise, we keep searching to the left until we find a record of  $S_i$ .

(2) *Calculating  $\Delta_i(t)$  for a sequence  $S_i$ :* We can compute the value using the method described in Section III-D. During the calculation, on each related edge, we can use its  $i$ -th offset difference that is for this sequence  $S_i$ .

For instance, in Fig. 9, for the position  $\pi_b = 12$  in  $B$ , suppose we want to find its left immediate position with an edit operation of the sequence  $S_2$ . The largest ending position that is less than 12 is the ending position 11 in the node  $n_3$ . However, since the corresponding record of node  $n_3$  does not contain the sequence  $S_2$ , we need to continue going to the left and find the record with the key 4. We return 4 as the left immediate position with an edit operation.

We also extend the algorithms MIN\_VERIFY and A\_VERIFY as follows. We choose an optimal gram  $g_s$  as follows. For each sequence  $S_j$ , we can use Eq. 1 to compute the number of matching grams that need to be verified for this sequence  $S_i$ . We compute the summation of these numbers for all sequences  $S_1, \dots, S_n$ . We then choose an optimal  $g_s$  whose corresponding summation is minimal. We then use MIN\_VERIFY and A\_VERIFY on each sequence  $S_j$  to get answers.

## IX. EXPERIMENTS

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

### A. Experiment Setup

We considered the James Watson’s genomic sequences (called “JWB”) that contained 48 human chromosome sequences, each with 48.3 to 240.5 million characters [14]. Every two sequences have one common base sequence. Each sequence in JWB is represented as a base sequence  $B_i$  and a transformation  $\Delta_i$ .<sup>1</sup> In addition, we obtained 24 other sequences provided by the BGI organization. Each sequence is also represented as the same base sequence  $B_i$  in JWB and a transformation  $\Delta'_i$ .<sup>2</sup> This data set was an Asian Diploid Genome assembled based on 3.3 billion reads generated by the Illumina Genome Analyzer.

In order to obtain more transformations based on the same base sequence, we choose one chromosome with large size and relatively high quality (called “chr1.fa”) as a base sequence. We then used a software package called MUMmer v3<sup>3</sup> to compute the differences between the other chromosomes and this base sequence.

To generate queries, we chose a base sequence, randomly selected its subsequence, and modified 1% to 5% of its characters. The query lengths varied from 20 to 2,000. For each query, we constructed a query workload including 100 queries with the same query length. We ran each query workload 10 times and computed the average performance numbers.

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

<sup>1</sup>Available at <http://www.ics.uci.edu/dnazip/>

<sup>2</sup>Available at <http://yh.genomics.org.cn/download.jsp#pd>, in the section “YH variants and annotations.”

<sup>3</sup>Available at <http://mummer.sourceforge.net/>

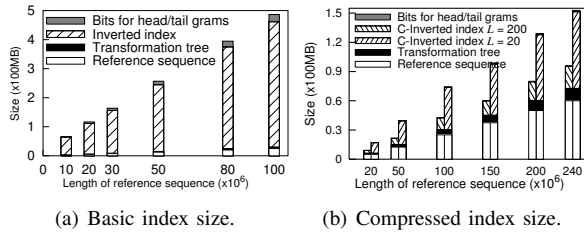


Fig. 10. Index size.

500GB disk, running a Ubuntu (Linux) operating system. Index structures were in memory. The fan out of each node in the transformation tree was 4.

### B. Index Size on Single Sequence

We first evaluated our techniques for the case of a single data sequence. We considered the single JWB sequence represented as its differences from the base sequence  $B$ . We generated base sequences with different sizes by selecting subsequences of  $B$  (let  $B_r$  be such a base sequence).

1) *Basic Indexes*: We first measured the size of the basic indexes described in Section III. We separately measured the size of the bits for head and tail grams, as described in Section IV. Fig. 10(a) shows the sizes of the base sequence, transformation tree, inverted index, and the bits for head and tail grams. When  $|B_r| = 10$  millions and the gram length  $q = 11$ , the size of storing the base sequence was 2.5MB (two bits for each character). The size of the transformation tree was only 0.613 MB (too small to be seen in the figure), while the size of the inverted lists was 60.64MB. The size of the bits of head and tail grams was 2.5MB. The index sizes increased linearly when the length of the base sequence  $|B_r|$  increased from 10 million to 100 million. When the base sequence  $B_r$  had 100 million characters, the transformation tree was 5.54MB, and the size of the inverted lists was 430.87MB. The size for the bits of head and tail grams was 25MB. The results for other  $q$  values were similar.

2) *Compressed Indexes Using Anchor Grams*: We then measured the sizes of the indexes using anchor grams as described in Section V. In the remaining figures, we use “C-Inverted index” to refer to such a compressed inverted index. Fig. 10(b) shows the results for different  $L$  values, where  $L$  should be less than or equal to the lower bound on query lengths in a query workload. When  $|B_r| = 100$  millions and  $L = 200$ , the size of C-Inverted index was 11.75MB, which was only 2.7% of the size of the original index! When the base sequence  $|B_r| = 240$  millions and  $L = 200$ , the size of C-Inverted index was 22.44MB. The value  $L$  affected the compressed-index size greatly. As  $L$  decreased, the index size increased.

### C. Performance of Exact Queries

We measured the performance of exact-subsequence queries using the proposed techniques.

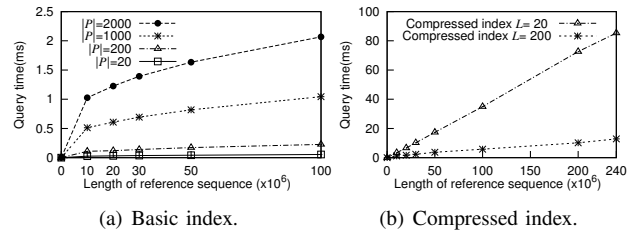


Fig. 11. Performance of exact-subsequence search ( $q = 11$ ).

(i) *Number of Verified Grams*: We used a base sequence of 100 million characters and a data sequence with a transformation of 118,084 edit operations. Table I shows the numbers of verified grams. When  $q = 11$  and the average query length was 200, the algorithm BASIC needed to verify 114,766 grams, while algorithm H\_VERIFY reduced the number by 99.67%! The algorithm MIN\_VERIFY was able to further reduce the number by 56.7%. We also evaluated the number of verified grams using a compressed inverted index. We got similar results when using other gram lengths and query lengths.

(ii) *Query Time*: Fig. 11(a) shows the performance of exact subsequence-search queries on one data sequence using the basic index and the algorithm MIN\_VERIFY for choosing the best “ $g_s$ ” gram. When  $|B_r| = 10$  million and  $|P| = 2000$ , each query took only 1.03ms. As  $|B_r|$  increased to 100 millions, the average query time increased slowly, each query took only 2.09ms. When the query length was 20, the time was only 0.02ms. Fig. 11(b) shows that A\_VERIFY also achieved a high query performance. With a base sequence  $|B_r| = 240$  millions,  $|P| = 2000$ , the average query time was 96.91ms when  $L = 20$  and 12.84ms when  $L = 200$ . As the  $L$  value increased, the search process became more efficient, since a larger  $L$  value can help us discard more positions in the inverted index.

Notice that, for a base sequence with 240 million characters and 270,631 edit operations, the number of incremental subsequences that need to be checked is zero when  $|P|=2000$ . This number increased to 4 and 261 when  $|P|=200$  and  $|P|=20$ , respectively.

### D. Performance of Approximate Queries

We implemented the approaches in Section VII for doing approximate subsequence search. We set  $|B_r| = 100$  millions, the gram length  $q = 11$  and  $|P| = 2000$ . The results are shown in Fig. 12.

Using compressed index required longer time to get answers since we need to verify more matching grams as shown in Table I. As the number of data sequences increased as shown in Fig. 16(b), both basic index and compressed index can provide a good scalability.

**Substitution Distance**: Fig. 12(a) shows the query performance for the substitution distance threshold  $k = 20$ . Our approach for choosing the best “ $g_s$ ” gram needed 3.44ms to compute the similar subsequences using the basic index and

TABLE I

NUMBER OF VERIFIED GRAMS ON A BASE SEQUENCE OF 100 MILLION CHARACTERS AND A TRANSFORMATION OF 118,084 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		H_VERIFY		MIN_VERIFY	A_VERIFY ( $L = 100$ )		A_VERIFY ( $L = 200$ )	
		# of verified grams	# of verified grams	Reduction	# of verified grams	Reduction	# of verified grams	Reduction	# of verified grams	Reduction
200	5	27,637,645	156,369	99.43%	53,797	99.81%	282,753	98.98%	147,132	99.47%
	7	2,374,355	15,124	99.36%	4,120	99.83%	24,134	98.98%	12,445	99.48%
	9	338,732	1,538	99.55%	514	99.85%	3,184	99.06%	1,645	99.51%
	11	114,766	380	99.67%	165	99.86%	954	99.17%	488	99.57%
2,000	5	278,745,673	429,278	99.85%	316,092	99.89%	2,182,316	99.22%	1,136,194	99.59%
	7	24,822,348	40,173	99.84%	28,657	99.88%	195,770	99.21%	101,671	99.59%
	9	3,842,857	5,647	99.85%	4,592	99.88%	31,488	99.18%	16,604	99.57%
	11	1,398,560	1,895	99.86%	1,676	99.88%	12,151	99.13%	6,600	99.53%

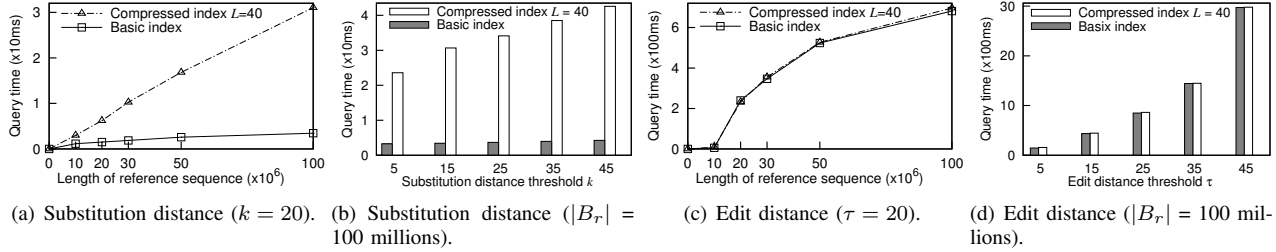


Fig. 12. Performance of approximate subsequence search on a single sequence ( $q = 11$ ,  $|P| = 2000$ ).

31.04ms using the compressed index when  $L = 40$ . Fig. 12(b) shows the query performance of different substitution distance thresholds. We can see the time increased slowly as we increased the threshold value  $k$ .

**Edit Distance:** Fig. 12(c) shows the query performance when the edit distance threshold  $\tau=20$ . When  $|B_r|=10$  millions, our approach needed 5.50ms using the basic index and 8.70ms using the compressed index when  $L=40$ . When  $|B_r|$  increased to 100 millions, it took 681.61ms using the basic index and 696.54ms using the compressed index. The reason of such similar performance numbers was that most of the time was spent on verifying the candidate pieces using the dynamic programming algorithm, which was time consuming. Fig. 12(d) shows the results for different edit distance thresholds.

**Smith-Waterman similarity:** The results using Smith-Waterman similarity are shown in Fig. 13. We set  $|P| = 2000$  and wanted to find all answer subsequences, each of which contains at least 98.5% identity mappings. The corresponding Smith-Waterman similarity value was 1910.

Fig. 13(a) shows the query performance when the threshold  $\gamma = 1960$  for a single data sequence. It means that each answer subsequence contained at least 99% identity mappings. When  $|B_r| = 10$  millions, our approach needed 0.039 seconds using both the basic index and the compressed index for  $L = 40$  due to the time-consuming verification. When  $|B_r|$  increased to 100 millions, our approach took the same time, 1.62 seconds, using the two indexes. Fig. 13(b) shows the results for different thresholds, where the threshold 1970 means that each answer subsequence contained at least 99.5% identity mappings. The average query time was still good; we could get all answer

subsequences within 3 seconds. Fig. 13(c) shows the query performance on 10 data sequences when  $\gamma = 1960$ . Fig. 13(d) shows the query performance when we increased the number of data sequences.

### E. Index Size on Multiple Sequences

Fig. 14(a) shows the index sizes of a base sequence with 100 million characters when the number of data sequences increased. We used  $q = 11$ . The size of the base sequence was 25MB. When we had 6 data sequences, the transformation was only 29.06MB, while the size of the inverted lists was 430.87MB. The size of the inverted lists for multiple sequences did not change since this size only depended on the base sequence (see Fig. 10(a)). The size for the bits of head and tail grams was 150MB. When the number of data sequences increased to 15, the transformation tree grew to 71.41MB, and the size for the bits of head and tail grams was 375MB. The figure also shows that all the sizes increased slowly. The reason was that in the inverted index, we only used two bits to store the flags of a positional gram for each data sequence. In addition, there were not too many differences stored in the transformation tree due to the high similarity between the data sequences and the base sequence.

Fig. 14(b) shows the advantages of our compression technique. When we had 40 sequences, each of which containing 100 million characters, the total index size was only 367.74MB for  $L = 200$  and 687.45MB for  $L = 20$ . When we stored more sequences as differences from the same base sequence, the size of compressed inverted index increased more slowly than the size of the transformation tree.

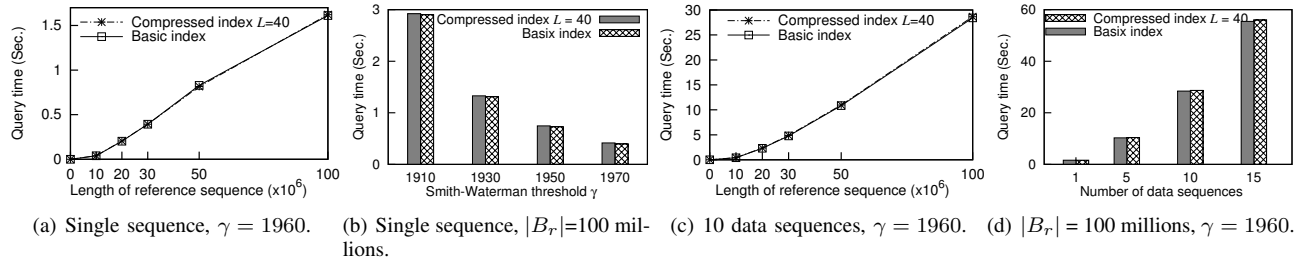


Fig. 13. Performance of approximate subsequence search using Smith-Waterman similarity ( $q = 11$ ,  $|P| = 2000$ ).

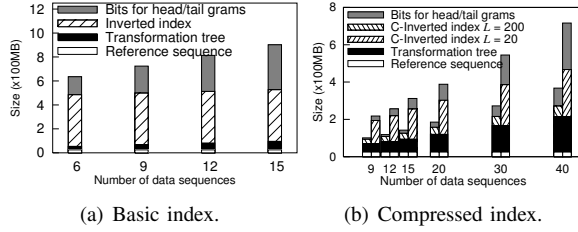


Fig. 14. Indexes sizes on multiple sequences (base sequence  $B_r$  had 100 million characters).

#### F. Query Performance on Multiple Sequences

We evaluated the scalability of our techniques by increasing the number of data sequences. Each sequence had 100 million characters and  $q=11$ . Fig. 15(a) shows the scalability results using the algorithm MIN\_VERIFY on the basic indexes. With 6 data sequences, a query with length 2000 took an average of 7.21ms. When there were 15 data sequences, the average query time increased to 22.07ms. As seen in Fig. 15(b), A\_VERIFY found subsequences took 63.46ms for 10 data sequences and 505.28ms for 40 data sequences when  $L=200$  and  $|P|=2000$ . Figs. 16 and 17 show the approximate-query performance on multiple sequences when  $q = 11$  and  $|P| = 2000$ .

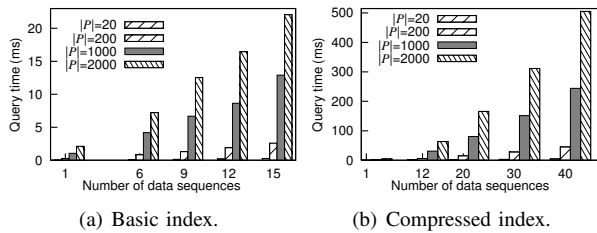


Fig. 15. Performance of exact-subsequence search on multiple sequences ( $|B_r| = 100$  millions,  $q = 11$ ).

#### G. Effect of Transformation Tree

We compared the effects of two extensions of transformation tree discussed in Section VIII. One extension is using a single tree for all the data sequences, and the other extension uses different transformation trees for different sequences. Fig. 18 shows when we used 10 sequences, each of which contained 100 million characters, the size of using a single tree was 47.89MB, and the size of using multiple trees was 48.32MB. When each sequence contained 10 million

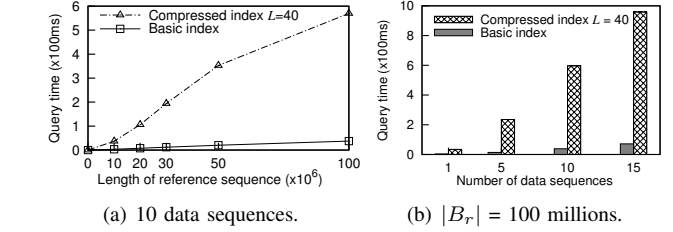


Fig. 16. Performance of approximate subsequence search using substitution distance ( $q = 11$ ,  $|P| = 2000$ ,  $k = 20$ ).

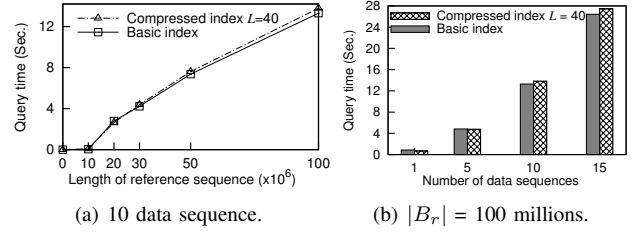


Fig. 17. Performance of approximate subsequence search using edit distance ( $q = 11$ ,  $|P| = 2000$ ,  $\tau = 20$ ).

characters, the single-tree solution used 0.89ms, and the multi-tree solution used 1.74ms. The time difference became larger when we increased the length of data sequences. When each sequence contained 100 million characters, the single-tree solution used 8.11ms and the multi-tree solution used 16.56ms.

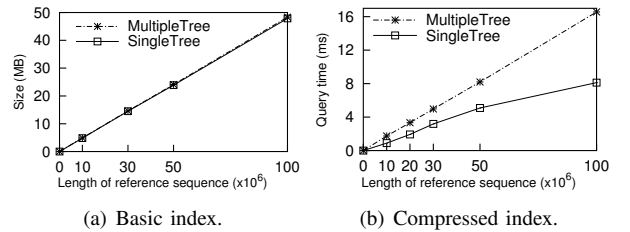


Fig. 18. Effect of transformation tree for 10 sequences, each of which contained 100 million characters.

We also test the effect of different fanout values. Table II shows that as the fan-out increased, the index size decreased slowly. when we used 10 sequences, each of which contained 100 million characters, as fanout values increased, the query time increased. When fanout was 2, the average query time was 0.703ms, when fanout was 4, the query time was 0.731ms, when fanout was 8, the query time was 0.773ms, and when

fanout was 16, the query time was 0.819ms.

TABLE II  
EFFECT OF FANOUT.

Fanout	Transformation tree size	
	Number of sequences = 10	Number of sequences = 40
2	47.368MB	189.434MB
4	46.383MB	185.604MB
8	45.971MB	183.997MB
16	45.785MB	183.262MB

#### H. Comparisons with Other Algorithms

We compared our techniques with the following commonly used subsequence-search algorithms.

- Boyer-Moore algorithm (BM for short) [1]. It is a typical online algorithm for exact subsequence matching.
- Fredriksson’s filtration algorithm (FF for short) [4]. It is an average-optimal algorithm for multiple approximate subsequence matching.
- BWT-SW algorithm [7]. It can find all local alignments for searching nucleotide sequences.
- BLASTn (“Basic Local Alignment Search Tool”), which is a widely used tool for comparing DNA sequences.
- RLCSA algorithm [8]. It provide exact subsequence search on the same compressed structure with us.

Notice that, we focus on finding all answer subsequences, however, BLASTn does not provide this guarantee.

The first four algorithms are not designed for compressed sequences. We first decompressed each data sequence, then ran these algorithms. We used the base sequence with 240 million characters and the data sequence with a transformation of 270,631 edit operations. We decompressed the sequence by applying these edit operations on the base sequence, which took 2.07 seconds. Our approach used a compressed index for  $L = 40$ , and could support both exact subsequence search and approximate subsequence search (see Table III). For the FF algorithm, we set the default gram length  $l = 8$ , which could achieve the best query time in our dataset. In order to compare our approach with BWT-SW and BLASTn, we set its  $E$  value to be  $10^{-10}$ . We collected the best local alignment for each query in the workload. We then calculated the corresponding Smith-Waterman similarity value for each best local alignment, and used each Smith-Waterman similarity value as a threshold in our approach to get answer subsequences for each query. Table III shows the comparison performance.

We also compared our approach with RLCSA (see Table IV). Using our approach, the index size was only 32.54MB, however, we need extra space to store the base sequence. We used 3.8 seconds to build up the index structure, whereas RLCSA required 76.7 seconds to do it. Our approach spent less time to answer queries using different query lengths.

#### I. Results on Individual Genomes on a Base Sequence with 3 Billion Characters

We would like to point it out that the 48 and 24 sequences used in our experiments are the assembled contigs of the genome. The numbers by themselves are not indicative of the

TABLE IV  
COMPARISON OF EXACT SUBSEQUENCE QUERY USING 2 SEQUENCES,  
EACH OF WHICH HAS 240 MILLION CHARACTERS.

	Storage of base size	Index size	Construction time	Query time: Exact search	
				$ P  = 200$	$ P  = 2000$
Ours	240MB	32.54MB	3.8 Sec.	0.73ms	5.01ms
RLCSA	–	143.822MB	76.7 Sec.	1.42ms.	6.63ms

size of the genome, since these sequences, when combined together, should be all about 3 billion bases.

In order to test our algorithms on individual genome sequences, we run our algorithm on an Dell PowerEdge R710 server with an 2x Intel Xeon Processors X5670 CPU and 12x8GB DDR3 (96GB) memory, running a Ubuntu (Linux) 64-bit server 10.04 LTS operating system.

For each of the 24 chromosomes, we used a software package MUMmer v3 to compute the differences (delta) between the corresponding data sequence and the chromosome. We concatenated these 24 deltas to have a big “delta”.

Using our approach, We used 540MB to store the differences in a transformation tree. When  $L = 40$ , the size of inverted index was 730MB for a single sequence and 3242MB for 10 sequences. We used average 0.84 seconds to answer a query with length 2000 and average 1.18 seconds to answer a query with length 3000.

Table V shows the results of approximate subsequence search.

TABLE V  
RESULTS OF APPROXIMATE SUBSEQUENCE SEARCH FOR INDIVIDUAL GENOMES.

Type	Number of data sequences	Threshold	Query time (Sec.)
Substitution distance	1	20	1.3
	10	20	9.4
Edit distance	1	20	14.7
	10	20	328.0
Smith-Waterman similarity	1	3960	8.1
	10	3960	297.2

When using Smith-Waterman similarity, our algorithm only used 8.1 seconds to calculate approximate answers on 1 sequence. In order to compare to BWT-SW approach, we used 94.7 seconds to decompress the sequence first. Then BWT-SW used 55.5 seconds to get answers.

#### X. RELATED WORK

There are primary two kinds of compression techniques on long sequence collections. One kind of compression technique is based on finding repetitions in the collection, and replacing them with references to similar strings previously appeared. One typical technique includes a family of self-indexes, which is developed to suite for the repetitive sequence collection setting [8]. Another typical technique includes a family of Ziv-Lempel compression formats such as LZ77, LZ78, and LZW [5], [6]. See [10] for an excellent survey.

Another data-compression techniques proposed in [2], [3], [14] is based on the high similarity of sequences. Based on this data-compression techniques, RLCSA [8], [9] supports

TABLE III

COMPARISON OF ALGORITHMS USING JWB DATASET WHEN THE BASE SEQUENCE HAS 240 MILLION CHARACTERS AND  $|P| = 2000$ .

Algorithms	Description	Decompressing time	Query time:	Query time:		Space occupancies in memory
			Exact search	Approximate search		
				$ED(\tau = 20)$	SW measure	
Ours	Query on compressed sequences	–	0.06 Sec.	1.23 Sec.	0.87 Sec.	247.4MB (including 125.3MB index size)
BM	Query on uncompressed sequences	2.07 Sec.	0.53 Sec.	–	–	231.5MB
FF	Query on uncompressed sequences	2.07 Sec.	1.15 Sec.	2.79 Sec.	–	307.8MB
BWT-SW	Query on uncompressed sequences	2.07 Sec.	–	–	27.37 Sec.	249.7MB
BLAST	Query on uncompressed sequences, does not guarantee to find all answers	2.07 Sec.	–	–	3.6 Sec.	142.9MB

exact retrieval using suffix trie or suffix array. Literatures donot report their results on approximate subsequence search.

Our techniques support both exact and approximate subsequence search, and guarantee to find all answers using the idea of  $q$ -grams. As the results shown in Section IX, our approach could provide better running performance than RLCSA and required less time and space to build up index structure (including compressed index) than RLCSA.

Except subsequence search in compressed sequences, there are many algorithms for subsequence search in a uncompressed long sequence. Typical algorithms on this problem include Boyer-Moore algorithm (also called BM) [1], Fredriksson’s filtration algorithm [4], BWT-SW algorithm [7], and BLAST. Recent studies include [11], [13]. These approaches assume sequences stored in their original representation, while the focus of our work is to index sequences in a compressed format using their high similarity. Our experimental results show our new technique requires small memory space and provides better running performance than the above approaches on uncompressed sequences.

## XI. CONCLUSIONS

In this paper we have developed novel techniques to index large similar sequences to answer queries, motivated by the recent advances in next-generation sequencing technologies. We gave a full specification of these techniques, and presented a series of optimizations to improve their space and time efficiency. Our extensive experiments on real genomic sequences showed the high efficiency of the technique.

## REFERENCES

- [1] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.
- [2] M. C. Brandon and D. C. Wallace. Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, 25(14):1731–1738, 2009.
- [3] S. Christley, Y. Lu, C. Li, and X. Xie. Human genomes as email attachments. *Bioinformatics*, 25(2):274–275, 2009.
- [4] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM J. of Experimental Algorithmics*, 9(1.4), 2004.
- [5] R. González and G. Navarro. Compressed text indexes with fast locate. In *CPM*. LNCS 4580, 2007.
- [6] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over ziv-lempel compressed text. In *CPM*. LNCS 1848, 2000.
- [7] T. W. Lam, W. K. Sung, S. L. Tam1, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of dna. *Bioinformatics*, 24(6):791–797, 2008.
- [8] V. Makinen, G. Navarro, J. Siren, and N. Valimaki. Storage and retrieval of individual genomes. In *RECOMB*, pages 121–137, 2009.
- [9] V. Makinen, G. Navarro, J. Siren, and N. Valimaki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
- [10] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 61 pages, 2007.
- [11] P. Papapetrou, V. Athitsos, G. Kollios, and D. Gunopulos. Reference based alignment in large sequence databases. In *VLDB*, 2009.
- [12] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
- [13] J. Venkateswaran, D. Lachwani, T. Kahveci, and C. Jermaine. Reference-based indexing of sequence databases. In *VLDB*, pages 906–917, September 2006.
- [14] D. A. Wheeler, M. Srinivasan, and et al. The complete genome of an individual by massively parallel dna sequencing. *Nature*, 452:872–876, 2008.