Fast, Flexible Text Search Using Genomic Short-Read Mapping Model

Sung-Hwan Kim and Hwan-Gue Cho

The searching of an extensive document database for documents that are locally similar to a given query document, and the subsequent detection of similar regions between such documents, is considered as an essential task in the fields of information retrieval and data management. In this paper, we present a framework for such a task. The proposed framework employs the method of short-read mapping, which is used in bioinformatics to reveal similarities between genomic sequences. In this paper, documents are considered biological objects; consequently, edit operations between locally similar documents are viewed as an evolutionary process. Accordingly, we are able to apply the method of evolution tracing in the detection of similar regions between documents. In addition, we propose heuristic methods to address issues associated with the different stages of the proposed framework, for example, a frequency-based fragment ordering method and a locality-aware interval aggregation method. Extensive experiments covering various scenarios related to the search of an extensive document database for documents that are locally similar to a given query document are considered, and the results indicate that the proposed framework outperforms existing methods.

Keywords: Text similarity search, document search, short-read mapping, approximate string matching, plagiarism detection.

Manuscript received July 7, 2015; revised Dec. 21, 2015; accepted Jan. 25, 2016.

This work was supported by the Marine Biotechnology Program of Ministry of Oceans and Fisheries, Republic of Korea (PJT200620).

Sung-Hwan Kim (sunghwan@pusan.ac.kr) and Hwan-Gue Cho (corresponding author, hgcho@pusan.ac.kr) are with the Department of Electrical and Computer Engineering, Pusan National University, Rep. of Korea.

I. Introduction

As information repositories continue to store an ever-increasing amount of data, the capability to search for items among such data has become increasingly more important.

An inverted list-based information retrieval system is capable of conducting information searches well in certain domains. However, such a system is only able to accept a small number of words as an input query. Hence, for more complex input queries, we need an information search method other than an inverted list. Developing such a method (for handling complex input queries) is both complex and challenging. However, if achievable, it will have multiple potential applications including automatic document referencing [1] and text reuse detection [2]. In such applications, it is necessary to identify documents from a database that are locally similar to a given query document.

In bioinformatics, to reveal the nature and functionality of an unknown sequence, it is essential that we be able to identify locally similar sequences from among a large number of sequence databases. One such method for achieving this is short-read mapping [3], which has become popular with the development of second-generation sequencing technologies. We can trace the evolutionary paths of locally similar sequences by comparing sequences of phylogenetically close species.

With the development of compressed self-indexes such as the FM-index [4], it has become feasible to load genome-scale sequences in random-access memory, which in turn means that it is now possible to utilize the short-read mapping method in the field of sequence analysis.

If we consider documents as biological objects, edit operations such as insertions, deletions, and the substitutions of words or phrases can be viewed as an evolutionary process. We can then exploit biological sequence analysis tools to determine

the level of similarity of a document in relation to a given query document as if we were tracing evolutionary paths and identifying similarities among biological objects.

In this paper, using a method utilized in biological sequence analysis, we present a framework capable of finding documents that are similar to a given query document from among an extensive document database. Owing to the effectiveness and scalability of the short-read mapping method underpinning our proposed framework, the proposed framework is also effective and scalable. In addition, we propose both an interval aggregation method (for use with a mapping profile) and a strategy to determine the order in which fragments are processed. The proposed method and strategy were designed to help adapt the short-read mapping method into one that is appropriate for a textual document analysis. Furthermore, in this paper, we discuss a case in which only a limited amount of time is given to the search process, which is considered the norm among real-world applications. For example, a server may be required to abort resource-consuming tasks to ensure the quality of service, and can assign a deadline by which the tasks are to be terminated. Similarly, a server can deliver a termination command to tasks on a first-in, first-out manner based on the current load. Therefore, we must consider that an information searching task may be aborted at any time during its execution, with or without notice, and propose methods to address this problem.

The remainder of this paper is organized as follows. Background knowledge is provided in Section II, and related works are reviewed in Section III. The proposed framework is presented in Section IV, and details of both the proposed strategy for fragment ordering and the proposed interval aggregation method are given in Sections V and VI, respectively. Section VII presents the experimental results, and Section VIII provides some concluding remarks regarding this research.

II. Background

1. Burrows–Wheeler Transform

The Burrows–Wheeler transform (BWT) [5] of a character string results in a rearrangement of the string in such a way that it contains runs of similar characters.

Let SA denote a suffix array [6] of string T, which has a length of n. Further, let SA[i] denote the position of the ith smallest suffix of T. The BWT of T, denoted by T^{BWT}, is then defined as

$$T^{\mathrm{BWT}}[i] = T\left[\left((\mathrm{SA}[i]+n-2) \bmod n\right)+1\right].$$

The BWT is an algorithm that transforms a character string into in a matrix form. Such a matrix, called a Burrows–Wheeler matrix, can be computed by listing all of the strings obtained through rotational shift operations on T and sorting them in lexicographical order. The BWT of a Burrows–Wheeler matrix is taken to be the output of the algorithm itself, that is, the final column of the matrix.

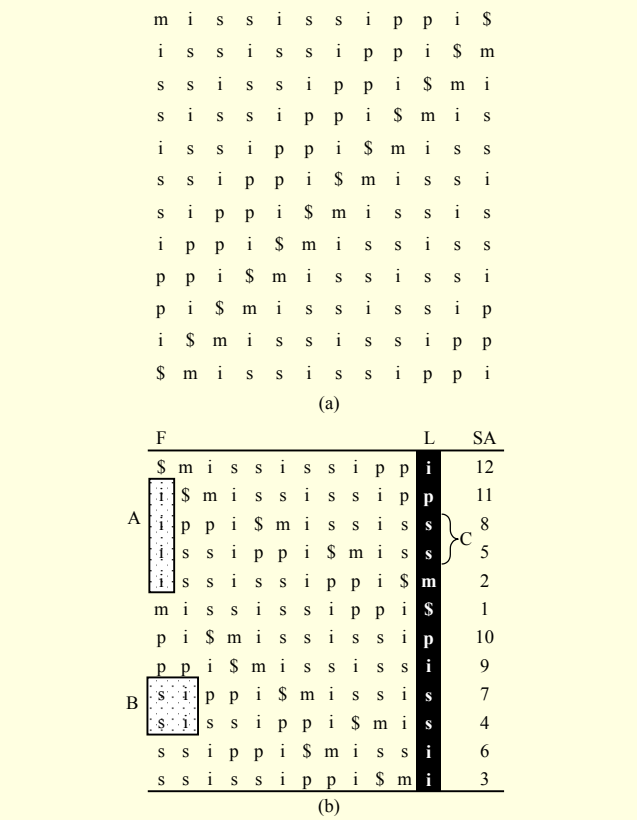

Fig. 1. BWT of the string "mississippi$": (a) all conjugate strings obtained through a rotational shift of the input string and (b) a matrix of sorted conjugate strings. The first column is indicated by the letter "F." The last column, indicated by the letter "L," is the BWT of the input string. The shaded rectangles indicated by the letters "A" and "B" are the search results for patterns "i" and "si," respectively. The letter "C" indicates the first and second occurrences of the character "s" in the final column, and is related to the search results for A and B. The column indicated by "SA" contains the corresponding suffix arrays.

The BWT of the input string "mississippi$" is described in Fig. 1. Conceptually, we compute conjugate strings of the input string by executing rotational shifts, as shown in Fig. 1(a). We then sort all strings lexicographically to obtain a corresponding Burrows–Wheeler matrix, as shown in Fig. 1(b). Concatenation of the characters in the last column results in the BWT of the input string.

2. FM-Index

The BWT was originally developed for textual data

compression. However, it has been determined [4] that the BWT can in fact be used with full-text indexes.

The FM-index is the first self-index to utilize a BWT and is the basis of many recent compressed indexes for string matching. It addresses the problem of space occupancy, which is a critical disadvantage of suffix trees and suffix arrays. The FM-index uses a property pertaining to the Burrows–Wheeler matrix. In such a matrix, the ith occurrence of a character in the first column corresponds to the ith occurrence of the same character in the final column; in other words, the two characters (one appearing in the first column and one appearing in the final column) have identical positions in relation to the initial input string and are in fact the same character. This is because the matrix is sorted in lexicographical order such that the ranks of strings having the same leading character are dominated by those of the same strings but with the first character removed. Using this property, we can efficiently execute string matching on a BWT. The result of a search for a string within a Burrows–Wheeler matrix is expressed in terms of the suffix range, $[l, r]$.

Suppose we have suffix range $[l, r]$ for query pattern P. The suffix range $[l', r']$ for the pattern xP is then computed as follows:

$$l' = C(x) + \text{rank}_x(l-1) + 1,$$

$$r' = C(x) + \text{rank}_x(r),$$

where $C(x)$ is the number of characters on T that are less than x, and $\text{rank}_x(i)$ is the number of occurrences of x in $T^{\text{BWT}}[1:i]$.

Suppose we are given the text string "mississippi$" and that we wish to search for the pattern "si" by using a BWT. The process for such a search is described in Fig. 1(b). Because a BWT supports backward searches, we may start with the last character of the pattern, that is, the letter "i." The search result for the character "i" is indicated by the letter "A." From this result, we can obtain the result for the pattern "si" by searching the final column for the character "s" among only those rows represented in "A." Given this, we can see that the third and fourth rows contain an "s" in the final-column position, as indicated by the letter "C" in the figure; these correspond to the first and second occurrences of the letter "s" in the final column. Thus, we seek the first and second occurrences of the letter "s" in the first column to yield the search result for pattern "si," as indicated by the letter "B" in Fig. 1(b).

If we wish to only compute the number of occurrences of a string pattern, we can then obtain this information by simply computing $r - l + 1$ for a given suffix range $[l, r]$. When we wish to determine the exact position of each occurrence of a string pattern within an input string, we can compute this information from the associated suffix array information. That is, if we have a suffix range $[l, r]$, then the occurrences of a given string pattern will be at positions SA[i], where $l \le i \le r$.

To save space, the FM-index does not store the entire suffix array. Instead, it stores only every kth element, which reduces the space requirement; however, it makes the computation of SA[i] more costly in terms of the amount of time taken. To emphasize the cost (that is, the time required) of such a computation on a suffix of rank i, we denote SA[i] by locate(i), thus removing the assumption behind the notation SA[i] that an array can be accessed within a fixed period of time. Similarly, count(P) is used to denote a function capable of computing the suffix range, $[l, r]$, of pattern P. Consequently, string matching on an FM-index can be represented as a call of count() followed by a call of locate() for as many times as the pattern occurs.

3. Short-Read Mapping

Bioinformatics is a research field wherein researchers attempt to understand biological data through the use of software tools and computational methods, and is employed primarily when one wishes to compare two or more biological sequences in an attempt to discover their functionalities by detecting their similarities.

The problem to be addressed here is that it is technically infeasible to directly read a sequence from a biological object.

Sequencing machines typically slice a sequence into a number of fragments. A significant number of such fragments, which are also referred to as *short-reads*, are then recognized and translated into digital data.

Given a set of short-reads taken from a target sequence and a previously decoded sequence that is known to be similar to the target sequence (also known as a reference sequence), we can attempt to reconstruct the target sequence; here, the main aim is to try to map each short-read to a specific position in the reference sequence (using the short-read mapping method, [3], [7]) and then aggregate the mapped positions to deliver a desired result. Short-read mapping is closely related to the problem of identifying local similarities between strings, and is the method implemented in this present study to address the problem of document searching.

III. Related Works

The search of an extensive document database for documents that are locally similar to a given query document and the subsequent detection of similar regions between such documents involve source retrieval and text alignment methods [8]. *Source retrieval* is a method used to conduct a search of an extensive document database for documents that are locally similar to a given query document [9]. In a typical source

retrieval problem, a system utilizes a search engine to perform such a search. Current related researches have tended to focus on the use of commercial search engines; however, such engines accept only limited types of input queries, usually consisting of only a couple of keywords. Consequently, we must look for ways to generate suitable search queries for such engines. However, for those able to construct their own database (as opposed to accessing the databases of commercial search engine companies), it is desirable to develop a tailored search engine, that is, one that is specifically suited to utilizing specific types of inputs, as opposed to the use of commercial search engines.

Text alignment is the task of aligning two documents to detect similar regions [10]. The Smith–Waterman algorithm, often referred to as a local sequence alignment, is a well-known method for determining similar regions between two strings. To overcome the quadratic complexity of such a local sequence alignment technique, seed-and-extend methods have been proposed [11]. Such methods first detect segment pairs having a certain level of similarity, which is a task that can be accomplished with the aid of a hash table containing keys (also known as *seeds*). Similar regions are then extended with regard to the context. The majority of state-of-the-art text alignment methods use seed-and-extend methods to accelerate the process of identifying a level of similarity between two or more documents. A seed-and-extend method is an effective method for the acceleration of a pairwise alignment between long sequences; however, such a method can consume a substantial amount of space if a significant number of documents need be indexed.

IV. Proposed Framework

In this paper, we wish to address the following problem: locate from a given set of documents only those segments that are locally similar to the query document.

When we construct a long string, D, by concatenating all documents in the given set, the problem can then be formulated as follows: find all (i, j) such that $\exists j, k, \text{sim}(D[i, j], Q[j, k]) \geq \theta$, where sim() and θ are a user-defined similarity function and a threshold, respectively, and where Q denotes a query document.

The proposed framework consists of four stages: (1) indexing, (2) query fragmentation, (3) string matching, and (4) interval aggregation. The overall procedure of the framework is described in Fig. 2.

The "string matching" stage can be aborted during the processing of query fragments.

Let $P(t)$ be the measured performance at time t. Assuming $P(t)$ is monotonically increasing, we can define the saturation time, t^*, as $t^* = \min \{ t \mid P(t) \geq (1 - \varepsilon)P^* \}$ for some ε, where

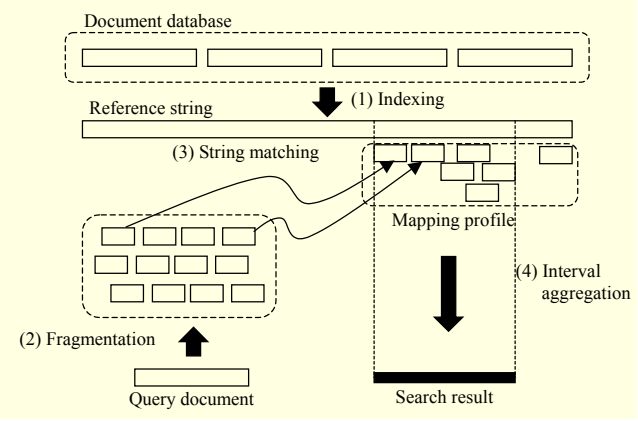

Fig. 2. Overall procedure of proposed framework comprising four stages: (1) indexing, (2) query fragmentation, (3) string matching, and (4) interval aggregation.

$P^* = \lim_{t \to \infty} P(t)$. Our aim is to minimize t^* such that the performance is maximized within the minimum amount of time as possible.

1. Indexing

The first stage is the preprocessing of the document database through which a search of documents similar to the query document will be conducted. Because such a search is based on string matching, we use a full-text indexing method such as FM-index to preprocess the documents. Front-end processing tasks such as alphabet sampling, tokenization, and stemming can be conducted before constructing the full-text index of a concatenated string. This can improve the search performance by executing approximate string matching in an implicit manner.

Suppose we are given documents $d_1, \cdots, d_n$. Let D be the string obtained by concatenating all given documents $d_1, \cdots, d_n$. We refer to D as the *reference string* from the term *reference sequence*, which refers to known sequences in bioinformatics. Let $\tau : \Sigma^* \rightarrow \Sigma'^*$ be a function from string to string holding the following: for any string x and y, $\tau(x)$ matches $\tau(y)$ if x matches y. That is, if we transform two strings with a function having this property, no matchings will be missed when comparing the transformed strings. Note that all context-free transforms satisfy this property. It is also worth remarking that the range of τ is not necessarily the same as its domain. For example, one can use a set of English words for the underlying alphabet of the domain and a binary set for the alphabet of the range.

2. Query Fragmentation

The remaining three stages address query processing. When a query is given, we transform the query document with the function used in the indexing phase. It holds that no fragments

of the transformed query are missing in the string matching. After transforming, we extract substrings from the transformed query string. We denote an extraction function by φ, which accepts a string and yields a set of strings, each of which is a substring of the given string. We also have a total order < on the extracted substring set, which determines the order in which the fragments are delivered during the string-matching phase. This ordering is important when we have a restriction in that only a limited numbers of fragments can be processed. In such cases, we must process the fragments using the smallest keys.

3. String Matching

After extracting substrings from the query string, we execute string matching using each of the extracted fragments as a query pattern. The matching profile, denoted by M, is a set of matchings each of whose element is a triplet (p_r, p_q, l), where p_r is the position on the reference string, p_q is the position in the query, and l is the matching length. The matching interval in the reference string is $[p_r, p_r + l - 1]$, and in the query string is $[p_q, p_q + l - 1]$. Both strings are assumed to be preprocessed by τ.

4. Interval Aggregation

When the string-matching phase is completed, we must determine the resulting intervals from the mapping profile. We denote an interval aggregation function by A, which accepts a mapping profile and yields a set of intervals that indicate the final similar regions.

5. Example

In this section, we present an example of the proposed framework. We set τ, which extracts English letters from a given string and converts them into lower case. Let φ be a function that accepts a string and yields the set of all character bigrams of the string. We define the fragment order < using the positions of the bigrams originated in the given query string, that is, fragments are sorted by their positions in the string. We define A to be a function that returns the union of the positions in the reference string. Next, suppose we are given a reference string D = "I am an example string!" and a query string Q = "Sample." To begin, we have the preprocessed strings $\tau(D)$ = "iamanexamplestring" and $\tau(Q)$ = "sample." After fragmenting the query, we have $\varphi(\tau(Q))$ = {"sa," "am," "mp," "pl," "le"}, where the fragments are ordered based on their positions in Q; in fact, this is the same as written. That is, "sa" will be used first and "le" will be processed last in the string-matching stage. When string matching is completed, we have M = {(2,2,2), (8,2,2), (9,3,2), (10,4,2), (11,5,2)}. Because the corresponding intervals in the reference string will be {[2,3], [8,9], [9,10], [10,11], [11,12]}, the final result given by A is their union {[2,3], [8,12]}. This indicates that "am" at position "2" and "ample" at position "8" in the reference string are locally similar to the given query. If the search process is aborted after processing the bigram "mp," we will have the mapping profile M = {(2,2,2), (8,2,2), (9,3,2)}, and the aggregated result will be {[2,3], [8,10]}.

V. Fragment Mapping

1. Least-Frequent-First Fragment Selection

Assume that a deadline in which we do not have sufficient time to process all of the fragments is given; the search will be terminated before completing this process. Only a portion of the fragments can therefore be processed. Hence, we must determine what selection of fragments will be the most effective. A basic strategy that chooses fragments in the order of their position in the query will fail under this scenario because many parts of the query may be excluded from the search process.

We focus on the observation that fragments have different frequencies; some occur very frequently, whereas others do not. In fact, it does not matter if we have sufficient time to process all of the fragments extracted from the query. However, when we have a time limit within which we are forced to return the search result, we must assign a priority to each fragment to allow us to process the more important fragments first. To address this problem, we select the fragments based on their frequency. A fragment with a lower frequency is selected before those with higher frequencies. That is, we sort the fragments in increasing order of frequency.

Compared with sequential ordering, frequency-aware fragment selection has three advantages.

First, we can process more fragments when only a limited numbers of locate() calls are permitted. If fragments are processed regardless of the number of occurrences, fragments with a high frequency can consume a substantial number of locate() calls and require an excessive amount of time owing to the time complexity related to the number of occurrences. Conversely, fragments having a low frequency require a reduced number of locate() calls; hence, we can process more fragments within the time restriction. In Fig. 3, the fragment "of" is expected to occur at enormous numbers of positions throughout the reference string; hence, the time for processing the fragment is also expected to be lengthy.

Frequency-based ordering can avoid this situation. Let $f(x)$ be the number of occurrences of fragment x. Suppose we have

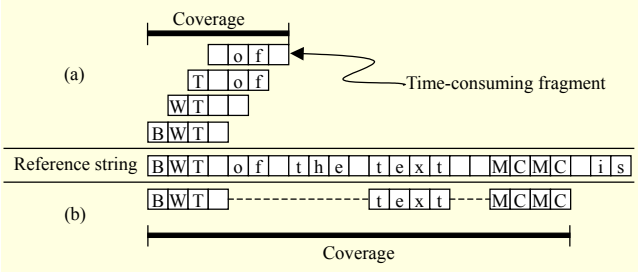

Fig. 3. Illustrative comparison of (a) sequential fragment ordering and (b) frequency-based fragment ordering. The latter can cover wider parts using a smaller number of fragments. Moreover, sequential fragment ordering is seriously influenced by the frequent occurrence of fragments.

a sequence $<x_i>$ of fragments arranged in their processing order. The number of distinct fragments we can process using t locate() calls is then expressed as the greatest index i such that $\Sigma_{j=0}^{i} f(x_j) < t$. It is clear that the strategy to maximize such i is to sort fragments in their order of frequency.

Second, we can cover a wider range across the query. Whereas only the front portion of the query can be processed in sequential ordering, as mentioned above, frequency-aware ordering is likely to select fragments with a higher variance in their positions in the query. Consequently, more of the query can be covered with a smaller number of fragments. An example is shown in Fig. 3. Suppose we have a query document identical to that shown in the figure. As illustrated in Fig. 3(a), sequential ordering covers only the initial portion of the query document. The frequency-based method, however, covers a wider range using a smaller number of fragments than the sequential ordering method. Consequently, when using the frequency-based ordering method, the performance reaches its peak significantly faster.

To formulate this, we define the coverage as the maximum interval length that covers the positions of the matched fragments, and assume that each fragment is processed during one unit of time. Suppose we have a query Q of length $|Q|$, and where its corresponding interval in the reference string has the same length. For simplicity, the interval consists of $|Q|/k$ independent k-length fragments. We can then represent this interval as a sequence of length $|Q|/k$, each element of which is the frequency of the fragment in that position. Let $S_{\text{freq}}(t)$ be the set of positions of t smallest values in the sequence. The coverage at time t can then be computed as $\max S_{\text{freq}}(t) - \min S_{\text{freq}}(t) + 1$. Note that $\max S_{\text{freq}}(t) - \min S_{\text{freq}}(t) + 1$ is at least t because $S_{\text{freq}}(t)$ has t distinct positive integers. In the case of sequential ordering, $S_{\text{seq}}(t)$ is defined as $\{1, \ldots, t\}$ such that the coverage at time t is always t. Therefore, frequency-aware fragment ordering has a higher degree of coverage than sequential ordering after processing the same number of fragments.

As the third reason, which is more important, a high fragment frequency is likely to deteriorate the performance because such fragments produce a greater number of false positives. In other words, infrequent fragments contribute the most to the level of performance. Let us assume that the reference string consists of randomly drawn fragments in regions other than the actual similar regions considered; in other words, false positive regions consist of independent random fragments. Let $p(x, y, d)$ be the probability that two fragments x and y will occur within a given interval comprised of d fragments. It is clear that if $p(x, y, d)$ is higher, a higher number of false positives are likely to be produced. Let $f(x)$ be the probability that fragment x will occur. The probability that neither fragments x and y will occur at all in the interval is $(1-f(x)-f(y))^d$. The probability that no fragments x (or y) will occur in the interval is $(1-f(x))^d$ (or $(1-f(y))^d$). Based on the inclusion-exclusion principle, the probability of either fragment x or fragment y not occurring in the interval will be $(1-f(x))^d + (1-f(y))^d - (1-f(x)-f(y))^d$. Hence, we have $p(x, y, d) = 1 - (1-f(x))^d - (1-f(y))^d + (1-f(x)-f(y))^d$. Because $\partial p(x, y, d)/\partial f(x) = d(1-f(x))^{d-1} - d(1-f(x)-f(y))^{d-1} \geq 0$, a greater number of false positives are likely to be produced as the fragment frequency increases, which has lead us to the use of infrequent fragments to reduce the number of false positive results.

2. Delayed Selection for Overlapping Fragments

When we use frequency-aware ordering, we must compute count(P) for each fragment P to determine their processing order. If the given time limit is overly short, such that only a small number of fragments can be processed, the computation of the number of occurrences of all fragments will be wasteful because the majority of the fragments will be unprocessed. If we can process only a portion of the fragments, it is better to select those fragments that cover as much of the query as possible. The simplest method to accomplish this is to avoid overlaps across the selected fragments. This strategy is also supported by the observation that overlapping fragments are also likely to appear as overlapped on the reference string because of their locality. Thus, if we have a limited capability to process fragments, it is wasteful to process overlapping fragments because they will occur simultaneously with a high probability.

To minimize the overlapping between the fragments being processed, we divide the fragments into several groups where the fragments do not overlap. We then consider a group index for determining the processing order of the fragments instead of simply breaking the ties randomly or based on their position. More formally, we define the total order $<_f$ on the set of extracted fragments as follows: $x <_f y$ if and only if $\text{occ}(x) + w\,\text{group}(x) < \text{occ}(y) + w\,\text{group}(y)$, where $\text{occ}(x)$ is the number

of occurrences of x, group(x) is the group id of x, and w is an integer parameter. If $w = 0$, we do not consider the groups, and each group is likely to be processed more separately as w increases. Note that if we want to insert any fragments from a group into the priority queue, we must compute the number of occurrences for all fragments within the group. After computing the frequencies of the fragments in group i, we place them into the priority queue using their keys, which are their frequencies increased based on the group weight. The computation for the next group, $i + 1$, will be delayed until the smallest key in the priority queue exceeds $w(i + 1)$, which is the lower bound of the key that a fragment in group $i + 1$ can have. We can also interpret w as the estimation of the cost required to compute the frequencies of the fragments in the next group. It is apparent that the sum of the length of the fragments in a group cannot exceed the length of the query document because they do not overlap. A length of time proportional to the query length is needed to insert the next group into the priority queue. Assuming that the cost of computing the frequency can be measured by the number of total characters, we simply set w as the length of the (preprocessed) query document.

We use a greedy method to assign the group id to the fragments. First, scanning from left to right along the query, we select any disjointed fragments. We then repeat this starting at a position where the overlapping length of the fragments in the previous groups and those in the current group can be minimized. If we use fixed-length fragments, this can be formulated using a bit representation. Suppose we are extracting length-k fragments from the query document and k is a power of 2. In the first phase, we can make group 0 with the fragments starting at positions that are a multiple of k. For group 1, we choose fragments starting at a position whose remainder divided by k is $k/2$. In this manner, we minimize the maximum length of the overlapping interval of a pair of fragments selected from both group 0 and group 1. Similarly, group 2 must choose position $k/4$ or 3 $k/4$ as the starting position of the first fragment. We can make k groups by repeating this process. We can represent the group id, ranging from 0 to k–1, using a bit sequence of length lg k. We sort this bit sequence in reverse lexicographical order. The group index of the fragment starting at position j is the rank of the bit representation of j, as previously defined. For example, if $k = 8$, we require lg 8 = 3 bits to represent the group id, and the bit sequences are sorted as 000, 100, 010, 110, 001, 101, 011, and 111. Accordingly, the starting positions of the first fragment of each group will be 0, 4, 2, 6, 1, 5, 3, and 7, respectively.

3. Suffix Range Reuse

Even though we can process fragments more efficiently through grouping, as discussed previously, it is necessary to scan the entire query document whenever a group of fragments is initiated. In computing these frequencies, the most costly task is updating the suffix range on a Burrows–Wheeler transformed text. In particular, if we use compressed bit vectors to implement a rank() data structure, the cost for updating the suffix ranges become much higher. In this case, we can reuse the suffix range of the previously searched fragments instead of computing the suffix range for each fragment all over again.

Computing the suffix range of similar strings has also been addressed in the area of bioinformatics, and some short-read mapping tools such as in [12] construct in advance a hash table that contains the suffix ranges of all strings of a specific length. Although this technique dramatically reduces the search time and can be directly adopted, manipulating a hash table is not a good idea for our situation. There are two main reasons for this. First, the size of the alphabet is much larger than that of biological sequences, which results in an excessively large hash table. A large hash table involves not only an out-of-memory problem, but also loading overhead. Second, the length of the fragment should be fixed in the indexing time to benefit from precomputing the frequencies.

We present a trie-based approach to address this problem. When computing the suffix range for a fragment, we traverse each trie whose nodes contain the suffix range for its corresponding string. We start with the root node of the trie. We traverse the trie to determine the reverse of the query fragment. If we encounter a node that does not have a child node for the current character, say in position i, we then compute the suffix range for $Q[i:|Q|]$. Because we already have the suffix range for $Q[i + 1:|Q|]$, it takes only O(1) time. We then create a new child node and save the suffix range into it. Actually, it also takes O(1) to descend a node in the trie, but its constant factor is much smaller than that when computing the suffix range again. As a result, we can save a significant amount of computational cost in practice, particularly when the rank() data structure is very slow owing to its compression ratio. Moreover, this trie-based method does not require a large amount of space; the trie has only O($k|Q|$) nodes in the worst case because there are O($k|Q|$) fragments for a given query document.

VI. Interval Aggregation

In this section, we propose different interval aggregation methods for effectively computing the results from a mapping profile. Because only a limited number of fragments are processed, and these fragments have low frequencies, the mapping profile has an extremely small number of matchings. Note that our assumption is that fragments that are close to each other in the query document are likely to be matched at

close positions in the reference string. Fragments are likely to be located more densely in similar regions, whereas most of the fragments are located sparsely throughout the reference string. However, the positions where the matchings are located are not necessarily contiguous even in similar regions because the number of fragments is insufficient, and the similar regions in the reference string can be slightly different from the query. An immediate result right after the string matching process is thus a set of short matching segments; hence, we must merge close intervals into a long interval to report the final results.

1. Simple Merging Method

We can simply merge two intervals that are closer than the threshold distance d. That is, two matchings $(p_r^{(1)}, p_q^{(1)}, l^{(1)})$ and $(p_r^{(2)}, p_q^{(2)}, l^{(2)})$ are merged if their corresponding intervals $[p_r^{(1)}, p_r^{(1)}+l^{(1)}]$, $[p_r^{(2)}, p_r^{(2)}+l^{(2)}]$ are closer than d. Because we use fixed-length fragments, we have $l = l^{(1)} = l^{(2)}$. Then, without a loss of generality, we can assume that $p_r^{(1)} < p_r^{(2)}$. Now, we can illustrate this as the merging of intervals $[p_r^{(1)}, p_r^{(1)}+l+d]$ and $[p_r^{(2)}, p_r^{(2)}+l+d]$ into one long interval $[p_r^{(1)}, p_r^{(2)}+l+d]$ if they are overlapped. After merging all of the overlapped intervals, we discard intervals shorter than C to remove any accidental matchings. A greater value of C can result in the filtering of more false positives; however, correct answers may be discarded.

2. Locality-Aware Merging Method

The simple merging method cannot resolve cases in which two unrelated fragments are accidentally matched close to each other in the reference string. For example, suppose there are fragments that are more than tens of sentences apart from each other; however, they are matched in the reference sting within a single sentence. The simple merging method will merge these intervals into one long interval; however, this is not reasonable because they actually have nothing to do with each other. To ensure the locality in the query, we consider not only the distance in the reference string but also that in the query string. Thus, we merge two matchings $(p_r^{(1)}, p_q^{(1)}, l^{(1)})$ and $(p_r^{(2)}, p_q^{(2)}, l^{(2)})$ if both pairs of intervals $[p_r^{(1)}, p_r^{(1)}+l^{(1)}]$, $[p_r^{(2)}, p_r^{(2)}+l^{(2)}]$ and $[p_q^{(1)}, p_q^{(1)}+l^{(1)}]$, $[p_q^{(2)}, p_q^{(2)}+l^{(2)}]$ are closer than threshold d. Similar to the simple merging method, assuming $l = l^{(1)} = l^{(2)}$, a matching $(p_r^{(1)}, p_q^{(1)}, l^{(1)})$ can be represented as an axis-parallel rectangle $(p_r^{(1)}, p_q^{(1)}, p_r^{(1)}+l, p_q^{(1)}+l)$, which is defined by the bottom-left $(p_r^{(1)}, p_q^{(1)})$ and top-right $(p_r^{(1)}+l, p_q^{(1)}+l)$ points. This can then be described geometrically as the merging of extended rectangles $(p_r^{(1)}, p_q^{(1)}, p_r^{(1)}+l+d, p_q^{(1)}+l+d)$ and $(p_r^{(2)}, p_q^{(2)}, p_r^{(2)}+l+d, p_q^{(2)}+l+d)$ if they are overlapped. The merged rectangle is their minimum surrounding rectangle. After merging the rectangles, we contract the resulting rectangles using d. Finally, we discard those rectangles whose side is shorter than C to reduce false positives.

Compared with the simple merging method, the locality-aware merging method reduces the number of false positives in a probabilistic manner. Suppose we use the simple merging method, and that a fragment x occurs in position i in the reference string. The occurrence probability of a false positive merging from a fragment x and another fragment y will then be the same as the probability that at least one y occurs within distance d from x. We denote this probability as ρ. When we use the locality-aware method, a matched fragment y should also occur close to x in the query. If we have m number of fragments y in the query, then the probability that no fragments y will occur within the interval of $2d+1$ centered at the position of fragment x can be roughly expressed as $1 - B(|Q| - 2d - 1, m)/B(|Q| - 1, m)$, where $B(n, m)$ is a binomial coefficient. Thus, the false positive probability can be estimated as $\rho(1 - B(|Q| - 2d - 1, m)/B(|Q| - 1, m)) < \rho$, which tells us that the false positive probability of the locality-aware merging method is much smaller than that of the simple merging method.

VII. Experimental Evaluation

1. Experiment Setting

For our evaluation, we used the PAN 2013 dataset [13], which has been utilized in plagiarism detection competitions, The dataset consists of four subsets, each of which contains its own type of plagiarism cases; among them, we used the random obfuscation cases. This subset consists of 1,000 pairs of suspicious documents and the source document. Suspicious documents are produced by artificially plagiarizing a portion of the source document. We also used the publicly available Pizza & Chili English corpus [14]. We excerpted it into the proper size and concatenated it with each of the source documents to generate a reference document. All alphabet letters were converted into lower case, and non-alphanumeric characters were removed.

The experiments were conducted for three reasons. First, we wanted to show that our method works well in the comparison of document pairs. To demonstrate this, we compared our method against a state-of-the-art text alignment method [2], and measured the character-level F-score. Second, we also hoped to show that our method is more robust to a large sized document database. We also compared our method against a winnowing-based near-duplicate document search method [15], and evaluated them both based on their document-level accuracy. Finally, we conducted experiments to demonstrate the superiority of the newly proposed technique, which improves the performance of a genomic read-mapping model

Table 1. Performance of one-to-one text alignment.

	Precision	Recall	F-score
Existing method [2]	0.810	**0.834**	0.822
Proposed method ($f \leq 1$)	**0.960**	0.691	0.803
Proposed method ($f \leq 2$)	0.909	0.759	**0.827**
Proposed method ($f \leq 3$)	0.870	0.781	0.823

Table 2. Search performance for the most similar document.

	True	False	Accuracy
Existing method [18]	438	562	0.438
Existing method [18] ($f \leq 3$)	504	496	0.504
Proposed method	**614**	**386**	**0.614**

based document search method.

2. Pairwise Comparison

We conducted text alignment experiments to demonstrate the performance of a pairwise comparison. The dataset used has 1,000 pairs of documents, each of which consists of a suspicious document and a source document. We attempted to find similar regions in the source document for a suspicious document given as a query. We measured the F-score using the sum of the true positive, false positive, and false negative intervals in the source document at the character level. Our method used fragments having a frequency of less than or equal to f in the first group as compared to a state-of-the-art text alignment method [2]. For the parameters, we used $k = 8$ and $d = 128$.

As described in Table 1, the experimental results indicate that our method is competitive with the state-of-the-art method. Note that the text alignment method focuses solely on a pairwise comparison, and thus the expansion for a large document set is not trivial, whereas our method is robust regardless of the database size.

3. Searching in Large Document Set

To simulate a large volume document database, we constructed a corpus with a size of 100 MB. The corpus size after removing non-alphanumeric characters is about 80 M. Because the corpus we used does not have a document boundary, we split it into 22 K documents, each of which at a length of about 3.6 K, which is the average length of the source documents in the given dataset. In this experiment, we did not aggregate the matching fragments. Instead, we counted the number of fragments located in each document, and chose the top document with the greatest number of matches. Similarly, we generated a document signature according to [15] using the parameters $q = 4$ and $w = 146$, counted the number of shared signatures between the query and documents in the database, and returned the top results. As shown in Table 2, our method outperforms the existing winnowing-based method.

4. Performance Convergence

We also compared the proposed method against the methods presented in [16] and [17]. For each pair of suspicious and source documents in the dataset, we concatenated the source document and a string of length 1 M excerpted from the corpus, and then processed the suspicious document as a query. The combinations of parameters used were $k \in \{8, 10, 12, 14, 16\}$, $d \in \{32, 64, 128, 256, 512\}$, where C was fixed at 100. We measured the best F-score at every 0.5 ms during the search. The results are shown in Fig. 4. As expected, the performance of the method using all of the proposed strategies converged the most quickly. The non-grouping method was the worst during the first tens of milliseconds because it consumed too much time in computing the frequencies of all fragments in the

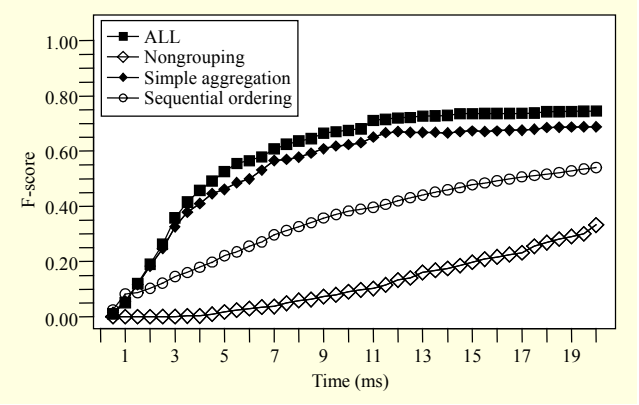

Fig. 4. Convergence performance with respect to time. The method using all of the proposed strategies outperformed the other combinations.

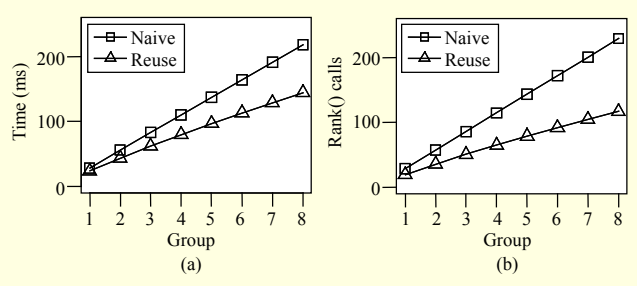

Fig. 5. Performance of naive and suffix range reuse method in terms of the processing time (a) and number of rank() calls (b).

initial phase. The simple aggregation method performed well but slightly worse than the locality-aware aggregation method.

5. Suffix Range Reuse

We measured the processing time and number of calls of function rank() to demonstrate the efficiency of a trie-based reuse of the suffix range. For this experiment, we used an RRR bit vector [18] with a block size of 7, as provided by the Succinct Data Structure Library [19]. The experimental results are shown in Fig. 5. The processing time and number of rank() calls can be improved by up to 50% when we reuse the suffix range, as proposed herein.

VIII. Conclusion

Searching for similar documents in an extensive database is an important task in recent applications of information retrieval and data management. This paper addressed the problem of finding documents that are locally similar to a given query document by borrowing the short-read mapping method from the field of bioinformatics, in which the local similarity search problem has been actively discussed. We also determined specific issues arising in the text search problem. The main contributions of the paper can be summarized as follows:

We proposed a framework for similar document searches. The framework is flexible because we can designate the parameters used for the appropriate target document characteristics.

For the efficiency and effectiveness of the search process, we proposed a frequency-based fragment ordering and fragment grouping method. We also presented a trie-based suffix range reuse method that improves the performance in terms of the search time, particularly when slow compressed bit vectors are used in the implementation of the full-text index.

Locality-aware interval aggregation methods were also proposed, and were confirmed to be effective in improving the search results by preventing accidental matches.

We conducted extensive experiments using various search scenarios including a one-to-one text alignment and a search from a large document set. The results showed that the proposed method outperforms previous existing methods.

References

[1] Y. Yang et al., "Query by Document," *ACM Int. Conf. Web Search Data Mining*, Barcelona, Spain, Feb. 9–12, 2009, pp. 34–43.

[2] M.A. Sanchez-Perez, G. Sidorov, and A. Gelbukh, "A Winning Approach to Text Alignment for Text Reuse Detection at PAN 2014," *Notebook PAN CLEF*, Sheffield, UK, Sept. 15–18, 2014.

[3] C. Trapnell and S.L. Salzberg, "How to Map Billions of Short Reads onto Genomes," *Nature Biotechnology*, vol. 27, 2009, pp. 455–457.

[4] P. Ferragina and G. Manzini, "Opportunistic Data Structures with Applications," *Ann. Symp. Foundations Computer Sci.*, Redondo Beach, CA, USA, Nov. 12–14, 2000, pp. 390–398.

[5] M. Burrows and D.J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," *Technical Report 124*, Digital Equipment Corporation, 1994.

[6] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-line String Searches," *SIAM J. Comput.*, vol. 22, no. 5, Oct. 1993, pp. 935–948.

[7] H. Li and R. Durbin, "Fast and Accurate Short Read Alignment with Burrows-Wheeler Transform," *Bioinformatics*, vol. 25, no. 14, 2009, pp. 1754–1760.

[8] M. Potthast et al., "Overview of the 6th International Competition on Plagiarism Detection," *Notebook PAN CLEF*, Sheffield, UK, Sept. 15–18, 2014.

[9] K. Williams, H. Chen, and C. Giles, "Supervised Ranking for Plagiarism Source Retrieval," *Notebook PAN CLEF*, Sheffield, UK, Sept. 15–18, 2014.

[10] M.A. Sanchez-Perez, G. Sidorov, and A. Gelbukh, "A Winning Approach to Text Alignment for Text Reuse Detection at PAN 2014," *Notebook PAN CLEF*, Sheffield, UK, Sept. 15–18, 2014.

[11] S.F. Altschul et al., "Basic Local Alignment Search Tool," *J. Molecular Biology*, vol. 215, no. 3, Oct. 1990, pp. 403–410.

[12] R. Li et al., "SOAP2: An Improved Ultrafast Tool for Short Read Alignment," *Bioinformatics*, vol. 25, no. 15, Aug. 2009, pp. 1966–1967.

[13] PAN 2013, Accessed June 19, 2015. http://pan.webis.de

[14] P. Ferragina and G. Navarro, *Pizza & Chili Corpus*, Accessed June 29, 2015. http://pizzachili.dcc.uchile.cl

[15] Y. Sun, J. Qin, and W. Wang, "Near Duplicate Text Detection Using Frequency-Biased Signatures," *Web Inf. Syst. Eng., Int. Conf.*, Nanjing, China, Oct. 13–15, 2013, pp. 277–291.

[16] C.S. Ock et al., "A Fast Searchong for Similar Text Using Genomc Read Mapping Method," *IEEE Int. Conf. Comput. Sci. Eng.*, Sydney, Australia, Dec. 3–5, 2013, pp. 219–226.

[17] S.-H. Kim and H.-G. Cho, "A New Approach for Approximate Text Search Using Genomic Short-Read Mapping Model," *ACM Int. Conf. Ubiquitous Inf. Manag. Commun.*, Bali, Indonesia, Jan. 8–10, 2015.

[18] R. Raman, V. Raman, and S.S. Rao, "Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees and Multisets," *ACM-SIAM Symp. Discrete Algorithms*, San Francisco, CA, USA, Jan. 6–8, 2002, pp.233–242.

[19] S. Gog, *Succinct Data Structure Library 2.0*, Accessed Dec. 1, 2015. https://github.com/simongog/sdsl-lite

Sung-Hwan Kim received his BS and MS degrees from Pusan National University, Rep. of Korea. Since 2013, he has been a PhD student at Pusan National University. His research interests include string processing algorithms and data visualization.

Hwan-Gue Cho received his BS degree from Seoul National University, Rep. of Korea, and his MS and PhD degrees from Korea Advanced Institute of Science and Technology, Daejeon, Rep. of Korea. Since 1990, he has been a professor at Pusan National University, Rep. of Korea. His research interests include computer algorithms, bioinformatics, data mining, and computer graphics.