JIS

An efficient DNA sequence searching method using position specific weighting scheme

Woo-Cheol Kim, Sanghyun Park, Jung-Im Won

Department of Computer Science, Yonsei University, Korea

Sang-Wook Kim

College of Information and Communications, Hanyang University, Korea

Jee-Hee Yoon

Division of Information Engineering and Telecommunications, Hallym University, Korea

Received 23 May 2005
Revised 13 Sepember 2005

Abstract.

Exact match queries, wildcard match queries, and *k*-mismatch queries are widely used in various molecular biology applications including the searching of ESTs (Expressed Sequence Tags) and DNA transcription factors. In this paper, we suggest an efficient indexing and processing mechanism for such queries. Our indexing method places a sliding window at every possible location of a DNA sequence and extracts its signature by considering the occurrence frequency of each nucleotide. It then stores a set of signatures using a multi-dimensional index such as the R*-tree. Also, by assigning a weight to each position of a window, it prevents signatures from being concentrated around a few spots in indexing space. Our query processing method converts a query sequence into a multi-dimensional rectangle and searches the index for the signatures overlapping with the rectangle. Experiments with real biological data sets have revealed that the proposed approach is at least 4.4 times, 2.1 times, and several orders of magnitude faster than the previous one in performing exact match, wildcard match, and *k*-mismatch queries, respectively.

Keywords: DNA database; indexing; query processing; exact match; wildcard match; *k*-mismatch

1. Introduction

DNA sequences hold the code that determines the characteristics of living organisms, and can be represented as a long list using the alphabet of nucleotides. In molecular biology, a basic way to understand a newly discovered DNA sequence is to infer its characteristics from the existing similar DNA sequences that have been understood [1]. *DNA sequence searching* is an operation that finds, from a DNA database, DNA (sub)sequences whose nucleotide arrangements are similar to that of a given query sequence. This operation helps molecular biologists infer the role, evolutional process, and chemical structure of a new DNA sequence. To cater for the evolutionary mutations and noises in DNA sequences, approximate match queries are preferred to exact match queries for DNA sequence searching.

The most fundamental way for processing approximate match queries is to use the Smith-Waterman alignment algorithm [2], which is based on dynamic programming to find an optimal local alignment between two sequences. The similarity model used in

Correspondence to: Woo-Cheol Kim, Department of Computer Science, Yonsei University, Korea. E-mail: twelvepp@cs.yonsei.ac.kr

this algorithm has been adopted by a majority of biologists, and thus has affected subsequent researches significantly. This algorithm, however, takes a long processing time of $O(mn)$, where m and n are the lengths of the two sequences to be aligned, respectively. Also, it has to access the whole data sequence from disk before processing.

The drawbacks of the method employing the Smith-Waterman alignment algorithm arise from comparing a whole data sequence with a query sequence. A natural idea to resolve this kind of drawback is to employ the filtering and refinement approach, which quickly finds only some candidate subsequences having a high possibility of being matched with the query sequence in the filtering stage, and then examines whether each candidate actually matches the query sequence in the refinement stage. BLAST [3, 4] is a typical example that follows this approach. Due to performance reasons, it uses a heuristic algorithm based on a similarity model that is slightly different from the one adopted in the Smith-Waterman alignment algorithm. Recently, Kaheci et al. [5] proposed the MR-Index, which also follows the filtering and refinement approach, for efficient processing of *k*-difference queries. A *k*-difference query is to find data subsequences that can be matched with a given query sequence by performing at most *k* replacing, inserting, and deleting operations.

In this paper, we address efficient processing of DNA sequence searching, especially exact match queries, wildcard match queries, and *k*-mismatch queries. Exact match queries search a DNA database for the subsequences that exactly match a query sequence. Wildcard match queries contain wildcard characters marked as '*' in a query sequence, and find the subsequences that match a query sequence. Note that a wildcard is regarded as matching any single nucleotide. *k*-Mismatch queries retrieve the data subsequences that have at most *k* nucleotides mismatched with those of a given query sequence. Note that the definition of a *k*-mismatch does not allow any insertions or deletions of nucleotides, but just inspects matches and mismatches. These exact match, wildcard match, and *k*-mismatch queries are widely used in various molecular biology applications such as retrieval of expressed sequence tags and DNA transcription factors [6].

The method employing the Smith–Waterman alignment algorithm and the method based on the MR-index can be also considered to process these three types of query. However, the query answers targeted by these methods are quite different from those targeted by the exact match, wildcard match, and *k*-mismatch queries. That is, these two methods retrieve more data than those retrieved by these three types of queries. Thus, they are not directly applicable to the applications this paper currently focuses on.

The Boyer–Moore algorithm [7] and the Knuth–Morris–Pratt algorithm [8] have been proposed for exact match queries. Also, the method combining the Aho–Corasick algorithm [9] and a scan vector has been proposed for wildcard match queries. These methods aim at optimizing the CPU performance in determining whether two sequences match each other. However, they suffer from a large processing time since they compare every possible subsequence in a data sequence with a query sequence. Moreover, they have to access the entire data sequence sequentially from disk.

The method based on suffix trees [6, 10], which is a kind of filtering and refinement approach, traverses the suffix tree to find candidates and then examines whether each candidate really matches a query sequence. This method can process the exact match, wildcard match, and *k*-mismatch queries, and achieves a relatively high performance since it accesses only the candidates from disk in the refinement stage. However, it incurs large storage overheads due to the characteristics inherited from the suffix tree. Suffix arrays [11] and compact suffix arrays [12–14] are also widely used as a variant of suffix trees. Both of them need less space than suffix trees, but are less efficient in processing the wildcard match and *k*-mismatch queries.

In this paper, we propose an approach for processing approximate queries that overcomes the problems mentioned above. We first suggest an effective method for indexing DNA databases. The method places a sliding window at every possible location of a data sequence, and extracts its signature by considering the occurrence frequency of each nucleotide. It then stores a set of signatures using a multi-dimensional index such as the R*-tree [15]. In addition, by assigning a weight to each position of a window, it tries to scatter the signatures over indexing space and thus successfully reduces the number of false alarms [16]. Using the proposed indexing method, we also suggest an algorithm which processes all the exact match queries, wildcard match queries, and *k*-mismatch queries efficiently. The algorithm converts a query sequence into a multi-dimensional rectangle, searches the index for signatures overlapping with the rectangle, and obtains the final answers by examining the signatures retrieved by index search.

To reveal the superiority of the proposed approach, we perform a variety of experiments with real biological

data sets and compare its performance with that of other previously adopted methods. The experimental results show that the proposed method is at least 4.4 times, 2.1 times, and several orders of magnitude faster than the suffix-tree-based method in performing exact match, wildcard match, and *k*-mismatch queries, respectively.

This paper is organized as follows. Section 2 defines some terminologies for further presentation, and also formulates the problem we are trying to solve. Section 3 briefly reviews previous methods related to approximate query processing in DNA databases, and then points out their drawbacks. Section 4 presents our basic approach for indexing and query processing. Section 5 addresses the limitations of our basic approach and then proposes an enhanced approach that overcomes them. Section 6 shows the effectiveness of the proposed approach via performance evaluation. Finally, Section 7 summarizes and concludes our work.

2. Definitions

In this section, we define the notations and terminologies used in this paper, and formulate the problem we are going to solve.

Definition 1: DNA sequence
A DNA sequence $T = <t_1, t_2, \dots, t_n>$ is an ordered list of characters in the alphabet of nucleotides. $|T|$ denotes the length of T, i.e. the number of characters in T. A DNA sequence stored in a database is called a *data sequence* and a DNA sequence submitted as a query is called a *query sequence*. $Q = <q_1, q_2, \dots, q_m>$ denotes a query sequence with m characters. We use T' to denote a contiguous subsequence of T. All the characters in T' must occur contiguously in T. In this paper, we use the term 'subsequence' to mean 'contiguous subsequence' for brevity.

Definition 2: alphabet Σ of nucleotides
The *alphabet* Σ of nucleotides consists of 15 characters that can occur in DNA sequences.

$$\Sigma = \{A, C, G, T, R, Y, S, W, K, M, B, D, H, V, N\}$$

Four characters, A, C, G, and T, are used to express the regions of a DNA sequence whose characteristics are discovered completely. We call these four characters 'principal nucleotides'. The remaining characters of the alphabet express the regions that have not been understood perfectly. For example, character Y denotes either C or T. Table 1 shows the characters included in the alphabet of nucleotides.

Table 1
Characters included in the alphabet of nucleotides

Code	Bases	Mnemonic
A	A	A-denine
C	C	C-ytosine
G	G	G-uanine
T(or U)	T	T-hymine (or U-racil)
R	A or G	pu-R-ine
Y	C or T	p-Y-rimidine
S	G or C	S-trong(3 H-bonds)
W	A or T	W-eak(2 H-bonds)
K	G or T	K-eto
M	A or C	a-M-ino
B	C or G or T	not-A
D	A or G or T	not-C
H	A or C or T	not-G
V	A or C or G	not-(T or U)
N(or X)	any base	a-N-y(or Unknown)

Definition 3: window
DNA data sequences are usually much longer than query sequences and also are of arbitrary lengths. Therefore, we use a window-based scheme to build an index for DNA data sequences. A 'window' is defined as a subsequence of fixed length taken from a DNA sequence. W and $|W|$ denote a window and its length, respectively. The window beginning at the ith position of a DNA sequence is denoted as W_i.

Definition 4: matching of two characters, s and q
Any two characters s and q are said to be *matched* with each other if the intersection of the set of characters represented by s and the set of characters represented by q is not empty. For example, let us consider two characters R and K. Characters R and K representing $\{A, G\}$ and $\{G, T\}$ respectively (see Table 1) are matched because their intersection is not empty (i.e. $\{A, G\} \cap \{G, T\} = \{G\}$). However, characters R and Y are not matched because their intersection is empty (i.e. $\{A, G\} \cap \{C, T\} = \emptyset$).

Definition 5: matching of query sequence Q and data subsequence T'
There can be three types of match between query sequence Q and data subsequence T'.

- Exact match: Q and T', both of which are from the alphabet Σ, are said to be in '*exact match*' when they satisfy both of the following conditions: (1) $|Q| = |T'|$, and (2) for each i between 1 and $|Q|$, the ith element of Q matches the ith element of T' (i.e. q_i matches t'_i).
- Wildcard match: T' from the alphabet Σ and Q from

$\Sigma \cup \{*\}$ are said to be in '*wildcard match*' when they satisfy both of the following conditions: (1) $|Q| = |T'|$, and (2) for each i between 1 and $|Q|$, the ith element of Q matches the ith element of T' (i.e. q_i matches t'_i). Note that wildcard '*' matches any single character in the alphabet Σ.
- K-mismatch: T' from the alphabet Σ and Q from $\Sigma \cup \{*\}$ are said to be in '*k-mismatch*' when they satisfy both of the following conditions: (1) $|Q| = |T'|$, and (2) there are at least $|Q| - k$ characters in Q that are matched to the characters of their corresponding positions in T'.

Definition 6: DNA sequence searching problem
Given DNA data sequence T stored in a database and query sequence Q, the DNA sequence searching problem is to find all the subsequences T' of T that match Q.
In the definition of the DNA sequence searching problem, we focus on just a single DNA sequence stored in a database. In real applications, however, we may need to search a set of DNA data sequences. This can be handled simply by creating a long sequence by concatenating all DNA data sequences, and then searching this long sequence. Therefore, from now on, we do not discuss the searching of a set of DNA data sequences any more. Table 2 summarizes the notations used throughout the paper.

3. Related work

In this section, we introduce practical applications that necessitate the exact match queries, wildcard match queries, and k-mismatch queries targeted in this research, and review prior work on processing such queries.

Table 2
Notations used throughout the paper

Symbol	Definition
T	DNA data sequence
t_i	ith character of DNA data sequence T
T'	contiguous subsequence of T
Q	DNA query sequence
q_i	ith character of DNA query sequence Q
$\|T\|$	length of DNA data sequence T
$\|Q\|$	length of DNA query sequence Q
Σ	alphabet of nucleotides
W	window
$\|W\|$	length or size of window W
W_i	window beginning at the ith position of a DNA data sequence

3.1. Exact match queries

3.1.1. Applications: retrieval of expressed sequence tags. The expressed sequence tag (EST) is a DNA sequence of length 200–300, and appears only once in an entire gene sequence [6]. More than five million ESTs have been found and stored in databases. When a new DNA sequence is found, it is important to identify which ESTs are contained in the sequence. To achieve this, biologists perform an exact match query by using each EST as a query sequence. As a result, they can conjecture the characteristics of the newly discovered DNA sequence.

3.1.2. Previous methods. Without using any extra space, the Boyer–Moore algorithm [7] and the Knuth–Morris–Pratt (KMP) algorithm [8] effectively locate the positions within a data sequence, where a query sequence appears. Their worst-case time complexity for processing an exact match query is $O(n + m)$ where n and m are the lengths of data and query sequences, respectively. These algorithms, however, should access the entire data sequence from disk because they are based on the sequential scan.

The method based on suffix trees [6, 10] processes exact match queries with time complexity $O(m + k)$ where m is the length of query sequence and k is the number of answers. This method, however, incurs large storage overheads because the storage requirement of a suffix tree is $12n$ bytes even with careful implementation [17]. For example, Hunt et al. [18] reported a suffix tree of size 19G bytes for a DNA sequence of length 286M bases. Suffix arrays [11] are also widely used for the same purpose. A suffix array is constructed by sorting all suffixes of a sequence in lexicographic order and storing pointers to the suffixes in this order. Exact match queries can be processed efficiently by performing binary searches on suffix arrays. Suffix arrays, however, take more than $4n$ bytes. Other competing structures include the compact suffix array (CSA) [12, 14] and the compressed compact suffix array (CCSA) [13]. On reducing the redundancy in the suffix array, the CSA takes less than $2n$ bytes and the CCSA takes about $1.6n$ bytes. Time complexities for performing exact match queries using the suffix array, the CSA, and the CCSA are $O(m \log n + k)$, $O(m \log n + k)$, and $O((m + k) \log n)$, respectively; these bounds are much worse than the bound of the suffix tree.

3.2. *Wildcard match queries*

3.2.1. Applications: retrieval of DNA transcription factors. The DNA transcription factor binds to specific locations in DNA and regulates, either enhancing or suppressing, the transcription of the DNA into RNA [6]. In this way, the production of the protein that the DNA codes for is regulated. The studies of transcription factors have exploded in the past decade; many transcription factors have been discovered and can be separated into groups. Multiple DNA transcription factors in each group are represented by a sequence that includes wildcards. Hence, if a new sequence is found, one way to identify its features is to examine which DNA transcription factors belong to the sequence. The wildcard match query would be a good means for this.

3.2.2. Previous methods. The method combining the Aho-Corasick algorithm [9] and the scan vector has been proposed for this purpose [6]. By eliminating all the wildcards from a query sequence, this method first obtains a set of subpatterns and their starting positions within a query sequence. Next, by using a one-dimensional array called a scan vector, it finds the data subsequences, each of which contains all those subpatterns in order. This method, however, has a large storage overhead since it maintains a scan vector as large as the data sequence. Also, it requires a large processing time because it accesses the whole data sequence from disk.

3.3. K*-mismatch queries*

3.3.1. Applications: retrieval of expressed sequence tags. As stated earlier, the retrieval of ESTs is an application of exact match queries. However, DNA sequences frequently include deformations due to evolutionary mutations as well as contaminations. Thus, k-mismatch queries are preferred to exact match queries to cater for such deformations. By using the k-mismatch queries, we can allow at most k deformations to occur in a data sequence.

3.3.2. Previous methods. The suffix-tree-based method [10] constructs a suffix tree on data and query sequences. Next, it finds, from the suffix tree, the lowest one among the common ancestor nodes of both sequences. It then traverses down the subtree of that node until it encounters k mismatches. This method can be applied to the processing of exact match and wildcard match queries in a similar way. However, it suffers badly from a large storage overhead and the high cost of maintaining and traversing a huge suffix tree.

The method proposed in [19] also employs a suffix tree on data sequences, and efficiently handles both wildcard match queries and k-mismatch queries by using the concept of 'centroid path decompositions' of the suffix tree. This method assumes the suffix tree resides in main memory, and thus aims at optimizing the CPU processing time. Hence, it is not efficiently applicable to large database environments where data sequences and suffix trees should reside in disk rather than in main memory. While we can also consider storing the suffix tree in disk, this approach still has the problems caused by the aforementioned nature of the suffix tree.

Recently Amir et al. [20] used the occurrence frequency of each character in an alphabet and the pigeonhole principle to filter out unpromising answers at an early stage of their k-mismatch algorithm. This algorithm runs in time $O(n\sqrt{k \log k})$ for a data sequence with n elements, but it is effective only when the size of the alphabet is larger than $2\sqrt{k}$.

4. Basic signature index

This section proposes a new indexing method called BSI (Basic Signature Index) which efficiently supports approximate queries in large DNA databases, and also suggests a query processing method based on the proposed index.

4.1. *Index construction*

We first locate a sliding window of size W on every possible position of data sequence T. We then extract a *basic signature* from each window, considering the minimum and maximum frequencies of each principal nucleotide.

Definition 7: basic signature
Let $BS(W_i)$ be a basic signature of window W_i. $BS(W_i)$ is expressed as follows:

$$BS(W_i) = (([min_A,max_A], [min_C,max_C], [min_G,max_G], [min_T,max_T])\ i)$$

Here, min_A and max_A denote the minimum and maximum numbers of occurrences of character A, respectively, in W_i. The meanings of min_C, max_C, min_G, max_G, min_T, and max_T are analogous.

Note that a sliding window is represented as a rectangle rather than a point. This is because the data

sequence may contain some characters that represent more than a single principal nucleotide. For example, character B can be viewed as one of three characters: C, G, and T. Therefore, if a sliding window contains character B, we increase the maximum frequencies of C, G, and T by one, respectively.

Let us explain how to extract a basic signature from an example window W_{100} = 'ACTBGT'. Remember that character B can be replaced by one of three characters C, G, and T. Character A occurs once in the window and no character can be substituted for character A. Therefore, we obtain $min_A = 1$ and $max_A = 1$. Character C occurs only once in the window but character B may be substituted by C. Therefore, we obtain $min_C = 1$ and $max_C = 2$. Similarly, we obtain $min_G = 1$ and $max_G = 2$. Character T occurs twice, once in the third position and again in the sixth position. Furthermore, character B may be replaced by character T. Therefore, we obtain $min_T = 2$ and $max_T = 3$. In summary, the basic signature of W_{100} = 'ACTBGT' becomes (([1,1], [1,2], [1,2], [2,3]), 100).

$BS(W_i)$ is regarded as a four-dimensional rectangle of ($[min_A,max_A]$, $[min_C,max_C]$, $[min_G,max_G]$, $[min_T,max_T]$), along with the identifier i and thus can be stored in a multi-dimensional index such as the R*-tree [15] and the X-tree [21]. The total number of windows taken from a data sequence T is $|T| - |W| + 1$. Since $|T| >> |W|$ in most cases, $|T| - |W| + 1 \cong |T|$. That is, the number of windows to be stored in an index is almost the same as the number of characters in T. Note that the space required to represent a single window in index space is several times larger than that for storing a single character.

To reduce this storage space, we only store the MBRs (Minimum Bounding Rectangles) which cover the signatures for consecutive c data windows extracted from a data sequence. Note that the signatures for any two consecutive data windows are not that different from each other and thus are located closely in four-dimensional indexing space. Therefore, we expect that the MBR covering consecutive c signatures will not be enlarged much. By using this approach, we are able to reduce storage space for indexing to $1/c$. We call c the 'index compression coefficient'.

4.2. Query processing

The first step for query processing is to construct a query rectangle from query sequence Q. A query rectangle is formed depending on the types of a query submitted. Let us first suppose $|Q| = |W|$.

Exact match query. We construct a four-dimensional query rectangle, ($[min_A,max_A]$, $[min_C,max_C]$, $[min_G,max_G]$, $[min_T,max_T]$), which represents the minimum and maximum numbers of occurrences of the four principal nucleotides, A, C, G, and T, respectively, on the query sequence. That is, the minimum and maximum values of each dimension of a query rectangle are decided by the minimum and maximum frequencies of the corresponding nucleotide.

Wildcard match query. We first construct a four-dimensional query rectangle by using the procedure for exact match queries. We then increase max_A, max_C, max_G, and max_T by the number of occurrences of the wildcard in the query sequence.

***K*-mismatch query.** We construct a four-dimensional query rectangle by using the procedure for wildcard match queries. We then increase max_A, max_C, max_G, and max_T and by the value of k, and also decrease min_A, min_C, min_G, and min_T by the value of k. This implies that k-mismatches allow each principal nucleotide in a data window to occur k times more than or less than that in a query signature. If an adjusted minimum value becomes less than 0, we set it to 0. For example, given a query sequence 'ACTT' for a one-mismatch query, we obtain a four-dimensional rectangle ([0,2], [0,2], [0,1], [1,3]).

After constructing a query rectangle from a query sequence, we search the index for the data rectangles overlapping with the query rectangle. We call them 'candidate rectangles'. Then, we perform a post-processing step to discard false alarms, those candidates that are not real answers. Using the identifier of each candidate rectangle, this step reads its corresponding data window from the database, and then verifies whether the data window actually matches with the query sequence. Only the candidate rectangles which pass this verification are returned as final answers.

Remember that the proposed index stores only the MBRs covering signatures for consecutive c data windows. The identifier of each candidate rectangle is the beginning position of its consecutive c data windows. Therefore, by using the identifier of each candidate rectangle, we actually retrieve and verify the corresponding c data windows together in the post-processing step.

Until now, we have assumed $|Q| = |W|$. This assumption, however, does not hold in real applications since a query sequence and a data window may differ in their size. To handle this situation, we generalize our method as follows.

Case 1. $|Q| = |W|$
We construct a query rectangle, and search the index for the data rectangles overlapping with the query rectangle.
Case 2. $|Q| < |W|$
We generate a new query sequence Q' of length $|W|$ by appending $|W| - |Q|$ wildcards to the end of Q. Then, we apply the procedure used for Case 1 to Q'.
Case 3. $|Q| > |W|$
We first partition a query sequence Q into p subquery sequences, Q_1, Q_2, . . . , and Q_p, such that $p = \lceil |Q| / |W| \rceil$ and $|Q_i| = |W|$ for every i between 1 and p. Here, the last subquery sequence Q_p can be overlapped with Q_{p-1} as shown in Figure 1 to satisfy the constraint $|Q_i| = |W|$. Next, we apply the procedure for Case 1 to every subquery sequence, and then obtain final answers by merging all the results.

The performance of query processing is greatly affected by window size. As the window size gets larger, the performance of index searching gets better since the discriminating power of the signature extracted from a window becomes greater. However, when $|Q| < |W|$ (as described in Case 2), we append some wildcards to the end of Q to make a new query sequence Q' become of length $|W|$. These wildcards enlarge the search space of the index and thus increase the time for query processing. For efficient query processing, we could construct multiple indexes for different window sizes [5]. In our approach, however, rather than maintaining multiple indexes, we just keep a single index for an optimal window size which is carefully chosen by analyzing query patterns and considering the space–time trade off.

5. Weighted signature index

In this section, we point out the limitations of BSI and then propose a new index called WSI (Weighted Signature Index) to overcome them.

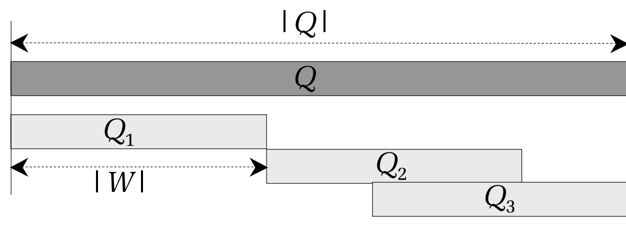

Fig. 1. Partitioning of a long query sequence into a set of sub-query sequences.

5.1. Limitations of BSI

BSI has a couple of intrinsic limitations.
Limitation 1: There are many distinct windows represented by the signature. In BSI, the signature of a window is decided only by the number of occurrences of each principal nucleotide. Therefore, there may be a large number of windows that are different from one another but are represented as an identical signature. It causes a large number of false alarms, resulting in high index-searching and post-processing costs. According to the result shown in Appendix A, there are $4^{|W|}/{}_4H_{|W|}$ disparate windows on the average that are represented by the same signature. Here, H is a symbol denoting *permutation with repetition*. Among these windows, only a single window matches a query sequence exactly and the others are just false alarms.
Limitation 2: The signatures are not uniformly distributed over indexing space. In most DNA sequences, the occurrence ratios of the four principal nucleotides A, C, G, and T are roughly 30%, 20%, 20%, and 30%, respectively [5]. The windows taken from such sequences also show similar occurrence ratios regardless of their beginning positions. Therefore, it is likely that many windows are represented by the signatures located close to this center ($0.3 \times |W|$, $0.2 \times |W|$, $0.2 \times |W|$, $0.3 \times |W|$). By this skew in distribution, as a query signature approaches the center, more false alarms appear after index searching. On the contrary, as a query signature gets away from the center, fewer false alarms occur. This makes the query response time unpredictable.

To overcome the above limitations, we need to increase the number of distinct signatures and spread them evenly across the indexing space.

5.2. Basic strategy

The simplest way to overcome the limitations of BSI is to extract more features from windows. For instance, we can consider representing the signature of an window as <$value_1$, $value_2$, $value_3$, . . . , $value_{|W|}$>, where $value_i$ corresponds to an integer number implying a character located at the ith position within the window. In this case, we note that a signature becomes large. This increases the dimensionality of the underlying index, and thus leads to the well-known 'dimensionality curse' [16]. To represent windows more discriminately without increasing the dimensionality, we propose a simple but effective method that assigns a weight to each position within a window. This makes it possible to express both occurrence

frequencies and occurrence positions of nucleotides with a signature of the same dimensionality. To incorporate this method into our indexing approach, we first define a weight function $w(j)$ $(1 \le j \le |W|)$ which assigns a weight to each position j within a window. We then extract a weighted signature from each window.

Definition 8: weighted signature

Let $WS(W_i)$ be a weighted signature of window W_i beginning at the ith position of a DNA sequence to be indexed. $WS(W_i)$ is expressed as follows:

$$WS(W_i) = (([wmin_A,wmax_A], [wmin_C,wmax_C], [wmin_G,wmax_G],[wmin_T,wmax_T])\ i)$$

Here, $wmin_A$ is the sum of the weights of the positions at which character A *must* occur in window W_i, and $wmax_A$ is the sum of the weights of the positions at which character A *may* occur in W_i. The meanings of $wmin_C$, $wmax_C$, $wmin_G$, $wmax_G$, $wmin_T$, and $wmax_T$ are analogous.

For example, consider window W_{200} = 'ACTBGT' which includes character B that can be substituted for one of three principal nucleotides C, G, or T. When the weight function is defined as $w(j) = j$, the weighted signature of W_{200} is computed as follows: character A occurs only at the first position whose weight is 1. Other than the first position, there are no other positions where character A may occur. Therefore, we obtain $wmin_A = 1$ and $wmax_A = 1$. Character C occurs only at the second position but it may occur at the fourth position as a substitute for character B. Since the weights of the second and fourth positions are 2 and 4, respectively, we obtain $wmin_C = 2$ and $wmax_C = 6$. By a similar computation, we obtain $wmin_G$, $wmax_G$, $wmin_T$, and $wmax_T$. In summary, we obtain (([1,1], [2,6], [5,9], [9,13]), 200) as the weighted signature of W_{200}.

By taking the above weighting scheme, disparate windows that were represented by the same basic signature may now be expressed by different weighted signatures. For example, consider two windows, W_{100} = 'ACTGGT' and W_{150} = 'CGAGTT'. Both of them are represented by the same basic signature ([1,1], [1,1], [2,2], [2,2]), but they are expressed differently by their weighted signatures, ([1,1], [2,2], [9,9], [9,9]) for W_{100} and ([3,3], [1,1], [6,6], [11,11]) for W_{150}. We incorporate the above weighting scheme into the proposed indexing method, thus producing a very effective index structure called WSI (Weighted Signature Index). WSI solves the problems of BSI by scattering the disparate windows, which were represented by the same basic signature, over indexing space as shown in Figure 2.

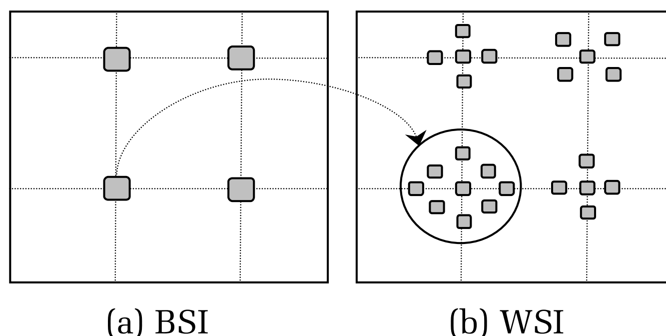

Fig. 2. Scattering the disparate windows, which were represented by the same basic signature, over indexing space by using a weighting scheme.

The query processing algorithm for WSI is not that different from that for BSI. However, when we construct a query rectangle for answering a k-mismatch query by using WSI, we need to consider the positions at which mismatches may occur. The procedure to build a query rectangle for a k-mismatch query is as follows.

(1) For each principal character X, we compute $wmin_X$ and $wmax_X$, the smallest and largest sum of weights of character X, respectively.

(2) We need to reduce $wmin_X$ further since mismatches are allowed at the positions where character X must occur. Among the positions at which X must occur, we select the positions with the top k weights. Let m be the sum of weights of such k positions. If characters other than X occur at all these positions, the smallest weighted sum for X becomes $wmin_X - m$.

(3) Similarly, we need to increase $wmax_X$ further since mismatches are allowed at the positions where character X does not occur. Among the positions at which X does not occur, we select the positions with the top k weights. Let M be the sum of weights of such k positions. If X occurs at all these positions by mismatches, the largest weighted sum for X becomes $wmax_X + M$.

(4) Thus, the smallest and largest weighted sums for character X now become $wmin_X - m$ and $wmax_X + M$, respectively. According to these new values, we enlarge the search range of the corresponding dimension of the query rectangle.

5.3. Weight function

Since the weight function determines the distribution of signatures in indexing space, it has to be carefully

designed. Consider a set of data windows which have the same *basic* signature. Their weighted signatures get scattered over indexing space by the weight function. Let us consider an MBR that covers all such weighted signatures. A larger MBR implies that the weighted signatures are scattered over a larger region. However, if the weighted signatures are scattered too widely, the corresponding MBR may overlap with its neighboring MBRs, producing new false alarms. Therefore, we have to choose a weight function which enlarges MBRs as much as possible without making them overlap with their neighboring MBRs.

Let us give a formal discussion on this issue. For each principal character X, let $R_{min}(X,s)$ denote the minimum of all $wmin_X$ values obtained from the set of all windows in which X occurs s times. That is, $R_{min}(X,s) = \sum_{j=1}^{s} sw(j)$ where $sw(j)$ denotes the jth smallest weight value in a window. Similarly, let $R_{max} = (X,s)$ denote the maximum of all $wmax_X$ values obtained from the set of all windows in which X occurs s times. That is, $R_{max}(X,s) = \sum_{j=|W|-s+1}^{|W|} sw(j)$.

To prevent neighboring MBRs from overlapping, $R_{max}(X,s) < R_{min}(X,s+1)$ should be satisfied for every s between 0 and $|W| - 1$.

$$
\begin{aligned}
& R_{max}(X,s) < R_{min}(X,s+1) \\
\Leftrightarrow\ & R_{max}(X,s) - R_{min}(X,s+1) < 0 \\
\Leftrightarrow\ & \sum_{j=|W|-s+1}^{|W|} sw(j) - \sum_{j=1}^{s+1} sw(j) < 0
\end{aligned}
$$

We are able to achieve the best distribution of signatures when a weight function satisfies the above inequality. Supposing $w(j) = j + C$, let us solve the inequality. Note that $sw(j)$ is identical to $w(j)$ in this case.

$$
\begin{aligned}
& \sum_{j=|W|-s+1}^{|W|} sw(j) - \sum_{j=1}^{s+1} sw(j) < 0 \\
\Leftrightarrow\ & \sum_{j=|W|-s+1}^{|W|} w(j) - \sum_{j=1}^{s+1} w(j) < 0 \\
\Leftrightarrow\ & Cs - C(s+1) + \sum_{j=|W|-s+1}^{|W|} j - \sum_{j=1}^{s+1} j < 0 \\
\Leftrightarrow\ & -C - (s^2 + (1-|W|)s + 1) < 0
\end{aligned}
$$

Since the above inequality should be satisfied for every s between 0 and $|W| - 1$, we obtain the following:

$$
\begin{aligned}
& -C - (s^2 + (1-|W|)s + 1) < 0 \\
\Leftrightarrow\ & C > max_{s=0}^{|W|-1}\{-(s^2 + (1-|W|)s + 1)\} \\
\Leftrightarrow\ & C > \frac{(|W|-1)^2}{4} - 1
\end{aligned}
$$

Among the values of C which satisfy $C > \frac{(|W|-1)^2}{4} - 1$, we choose $|W|^2$ for the sake of simplicity. That is, we use $w(j) = j + |W|^2$ for a weight function.

5.4. Consideration of index compression

Since the basic signatures of neighboring data windows are located close together in indexing space, MBRs covering the basic signatures of consecutive c data windows will not be enlarged much. Exploiting this property, in Section 4.1 we suggested representing consecutive c data windows with a single MBR. In order to apply the same idea to WSI, we have to design a weight function which locates the weighted signatures of neighboring data windows close together in index space. Since the weight function discussed in Section 5.3 restricts the distance between weighted signatures and corresponding basic signatures within a certain range, MBRs covering the weighted signatures of consecutive data windows would not be too large. Therefore, WSI also uses a single MBR to express consecutive c data windows in index space.

6. Performance evaluation

This section verifies the superiority of the proposed method via performance evaluation with extensive experiments. Sections 6.1 and 6.2 present the environment and parameter settings for experiments, respectively. Section 6.3 presents and analyzes the results.

6.1. Environment for experiments

In our experiments, as data sequence T, we used six sets of DNA sequences downloaded from NCBI [22]: human chromosome 3 (2.5 Mbp), 17 (5 Mbp), 1 (7.5 Mbp), 2 (10 Mbp), 10 (20 Mbp), and 5 (40 Mbp). As a query sequence, we used 1000 DNA sequences of length 256 to 2048. Half of them were randomly selected from T, and the other half were obtained from DNA sequences [23] frequently used by biologists at laboratories.

We evaluated the performances of four approaches: BSI, WSI, SeqScan, and Suffix. BSI and WSI are the methods proposed in this paper. SeqScan is the method based on the sequential scan, and Suffix is the method that uses the suffix tree as an index structure. As performance criteria, we employed the *index size* and the *average elapsed times* for processing the exact match queries, wildcard match queries, and k-mismatch queries.

The hardware platform was a Pentium IV 2.6 GHz PC equipped with 512MB main memory and 80 GB hard disk with 7,200 RPM. The software platform was Redhat Fedora core2 with a buffer size set to 2 MB.

6.2. Parameter settings

6.2.1. Window size. As pointed out in Section 4.2, when a query sequence is shorter than a window, we should process queries after padding wildcards to the end of the query sequence, which results in performance degradation. Thus, the window size is set slightly smaller than the typical size of a query sequence.

To determine a window size, we analyzed the lengths of 35,685 query sequences downloaded from NCBI. From the results, we observed that 62% of them have lengths of 256 to 2048. Thus, we set the basic window size to 256 for further experiments.

6.2.2. Index compression coefficient. In order to reduce the size of an index, we store MBRs, each of which encloses signatures extracted from c adjacent data windows, rather than storing individual signatures. The index compression coefficient c directly affects the size of an index as well as the performance of query processing time. In order to find an optimal value for the compression coefficient, we evaluated the index size and the query processing time of BSI and WSI. We used human chromosome 2 of 10 Mbp as a data sequence and 1000 sequences of length 256–2048 as query sequences. After setting the value of k to 1% of the length of query sequences, we measured an average time for processing k-mismatch queries.

Figure 3 shows how the index size changes as the compression coefficient increases. In both BSI and WSI, the index size gets noticeably smaller as the compression coefficient increases. Figure 4 indicates the relationship between the size of the compression coefficient and the time for query processing. Note that the query processing time is represented as the sum of the index search time and the post-processing time. As the compression coefficient increases up to 80, the query processing time of both BSI and WSI decreases. This is because a larger compression coefficient contributes to a reduced index searching time owing to a smaller index. From that point, however, the query processing time of both BSI and WSI increases as the compression coefficient gets larger. The reason for this is that a larger compression coefficient also causes more false alarms, thus enlarging the post-processing time. In the subsequent experiments, we set the base value for the compression coefficient to 80.

Let us consider Figure 3 again to discuss the issue on the size of the proposed indexing method. When the index compression coefficient is set to 80, the sizes of BSI and WSI are 7.43 and 7.30 Mbytes, respectively, for the underlying data sequence of size 10 Mbp. The experiments with various data sets reveal that both BSI and WSI take about $0.7n$ bytes when an optimal value of compression coefficient is employed for indexing. This space requirement is much less than the $12n$ bytes needed by the suffix tree, the $4n$ bytes needed by the suffix array [11], and the $1.6n$ to $2n$ bytes needed by other compressed suffix arrays [12–14].

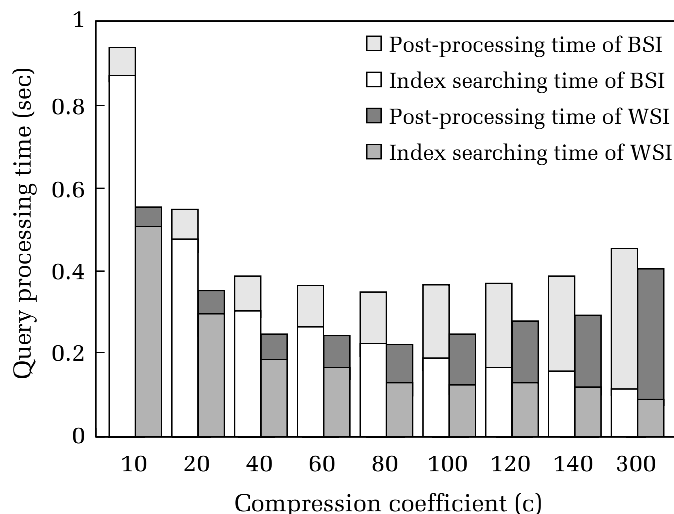

Fig. 4. Query processing time with various values for compression coefficient.

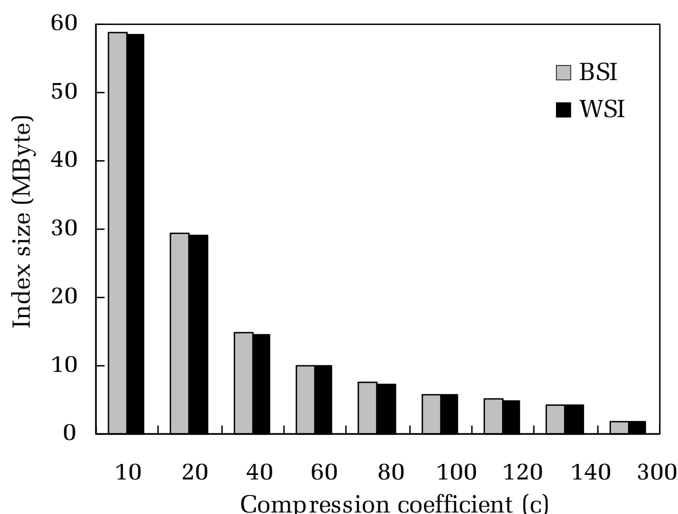

Fig. 3. Index size with various values for compression coefficient.

6.3. Results and analyses

To show the effectiveness of our approach, we conducted performance evaluation in terms of query processing time via experiments.

6.3.1. Experiment 1: query processing time with various query lengths. In this experiment, we compared the query processing times of all the approaches while changing the length of query sequences. We set the window size and the index compression coefficient to 256 and 80, respectively, as mentioned earlier, and used human chromosome 2 of 10 Mbp as a data sequence.

Figure 5 depicts the query processing times of all the approaches for exact match queries. In SeqScan and Suffix, whether a sequence matches with a given query sequence is decided by inspecting the first few characters in most cases. Therefore, SeqScan and Suffix show nearly constant performance regardless of the length of query sequences. In BSI and WSI, if query sequences get longer, the time for index searching becomes larger while the time for post-processing gets smaller due to the reduced number of candidates. Thus, we observe that the query processing time decreases until the length of a query sequence reaches a point (i.e. 512), and then grows gradually after that point.

The query processing times for wildcard match queries are shown in Figure 6(a) and (b) where the numbers of wildcards are 1 and 5%, respectively, of the length of query sequences. Let us first analyze the

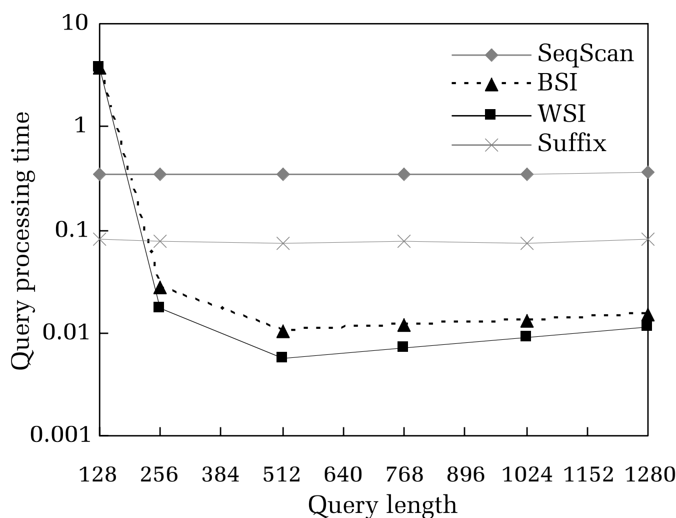

Fig. 5. Time for processing exact match queries with various lengths of query sequences.

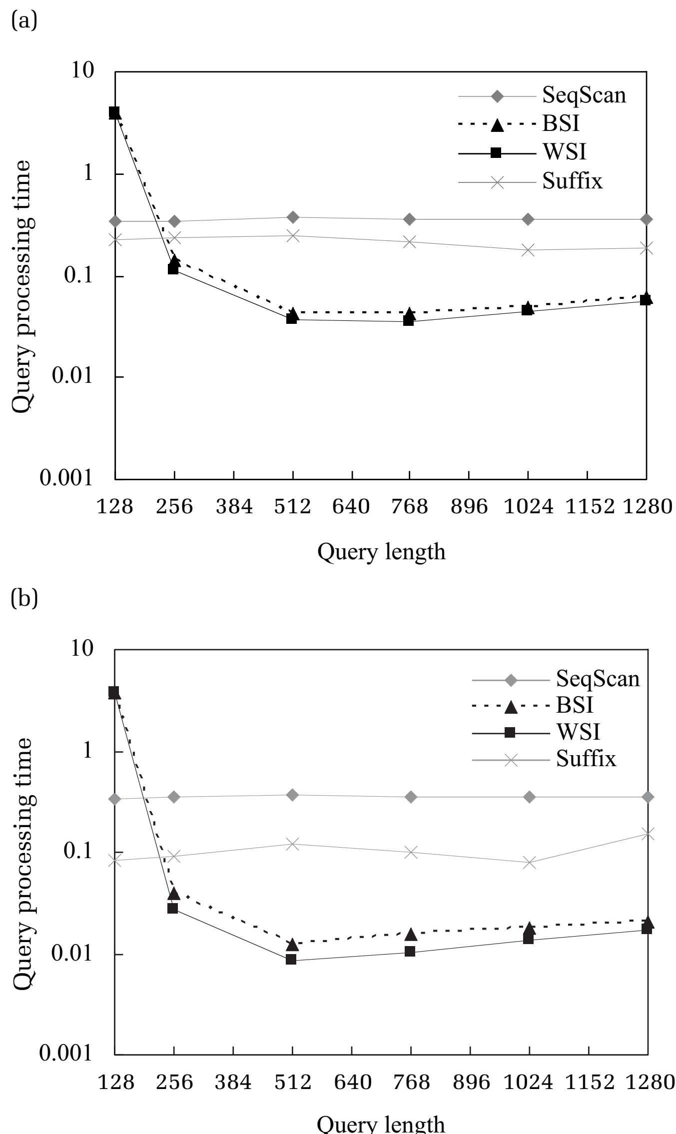

Fig. 6. Time for processing wildcard match queries with various lengths of query sequences. (a) The number of wildcards is 1% of the length of query sequences. (b) The number of wildcards is 5% of the length of query sequences.

graphs in Figure 6(a). Every approach spends more time to process wildcard match queries than exact match queries. If a query sequence contains wildcards, query processing tends to produce more answers, and also traverses a wider part of the indexes. In BSI and WSI, wildcards in a query sequence enlarge the corresponding query rectangle and increase the number of candidates, which leads to a long query processing time. As a query sequence gets longer,

 Journal of Information Science, 32 (2) 2006, pp. 176–190 © CILIP, DOI: 10.1177/0165551506062329

however, the number of candidates decreases remarkably at the cost of multiple index searches. As a result, the query processing performance of both BSI and WSI does not deteriorate abruptly even when query sequences become very long. The above analysis also holds for the graphs shown in Figure 6(b). The comparison of the graphs of Figure 6(a) with those of Figure 6(b) indicates that the performance improvement of BSI and WSI over SeqScan and Suffix is larger when the number of wildcards is 1% of the length of query sequences.

The query processing times for *k*-mismatch queries are shown in Figure 7(a) and (b) where the values of '*k*' are 1 and 5%, respectively, of the length of query sequences. Let us first analyze the graphs in Figure 7(a). Every approach spends much more time processing *k*-mismatch queries than exact match queries and wildcard match queries. In particular, Suffix shows even worse performance than SeqScan since the part of the index to be traversed increases explosively. In BSI and WSI, as a query sequence gets longer, the query processing times of BSI and WSI grow gradually. The above analysis also holds for the graphs shown in Figure 7(b). The comparison of the graphs of Figure 7(a) with those of Figure 7(b) indicates that the performance improvement of BSI and WSI over SeqScan and Suffix is larger when the value of *k* is 1% of the length of query sequences.

The above discussion on query processing performance does not consider an exceptional case where the length of a query sequence is half of the window size (i.e. 128 vs 256). In this case, BSI and WSI may get slower than SeqScan because 128 wildcards padded to the end of a query sequence enlarge the search space significantly. If this situation is anticipated from the relationship between the query length and the window size, it would be better to execute SeqScan rather than adhering to the proposed indexing system.

Let us compare the four approaches quantitatively without considering the query sequences of size 128. In exact match queries, WSI outperforms SeqScan, Suffix, and BSI 19–61 times, 4.4–13 times, and 1.3–2 times, respectively. In wildcard match queries with the number of wildcards set to 1% of query length, WSI performs better than SeqScan, Suffix, and BSI 12–43 times, 3.3–14 times, and 1.3–1.6 times, respectively. In wildcard match queries with the number of wildcards set to 5% of query length, WSI performs better than SeqScan, Suffix and BSI 3.1–10 times, 2.1–7 times, and 1.1–1.3 times, respectively. In *k*-mismatch queries with the value of *k* set to 1% of query length, WSI performs faster than SeqScan, Suffix, and BSI 17–51 times, more than 267 times, and 1.2–1.6 times, respectively. Finally, in *k*-mismatch queries with the value of *k* set to 5% of query length, WSI performs faster than SeqScan, Suffix, and BSI 1.5–6 times, more than 10,000 times, and 1.0–1.1 times, respectively.

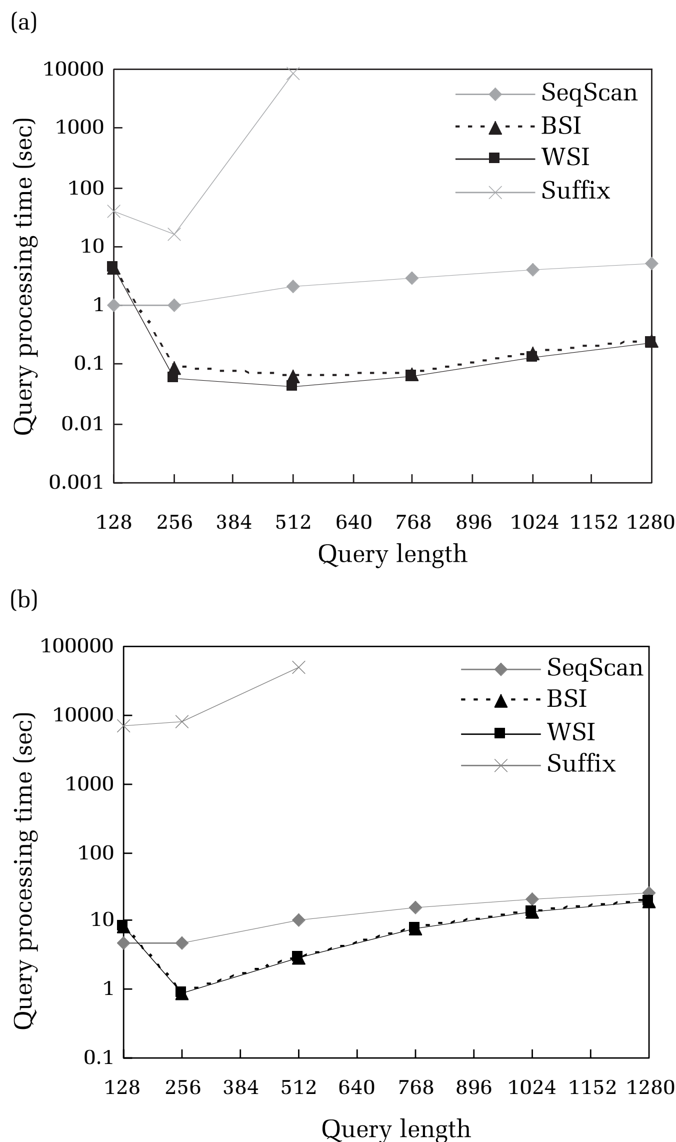

Fig. 7. Time for processing *k*-mismatch queries with various lengths of query sequences. (a) The value of '*k*' is 1% of the length of query sequences. (b) The value of '*k*' is 5% of the length of query sequences.

6.3.2. Experiment 2: processing time of *k*-mismatch with various *k* values. In this experiment, we compared the processing times of *k*-mismatch queries

of different approaches with various k values. We used human chromosome 2 of 10 Mbp as a data sequence. Figure 8 shows an average query processing time for each approach while setting k from 0 to 15% of the length of a query sequence. We observe that the query processing time of WSI, BSI, Suffix, and SeqScan gets higher as k grows. In WSI and BSI, a higher k value increases the part of an index to be traversed, and thus increases the query processing time gradually. In Suffix, however, as k grows, the part of an index to be traversed becomes explosively larger, and thus, the query processing time grows abruptly. The results reveal that WSI shows the best performance, and performs better than SeqScan, Suffix, and BSI 1.9 to 31 times, 3 to several thousand times, and 1.0 to 2.3 times, respectively.

6.3.3. Experiment 3: query processing time with various data sizes. In this experiment, we measured the query processing times of different approaches with various data sizes. As mentioned earlier, we set the window size, the index compression coefficient, and the length of query sequences to 256, 80, and between 256 and 2048, respectively. We excluded Suffix in this experiment since its performance degradation in performing k-mismatch queries on a large database is too serious to conduct experiments. We set both k for k-mismatch queries and the number of wildcard characters for wildcard match queries to 1% of the length of query sequences. Figure 9 shows an average processing time for each approach for exact match, wildcard match, and k-mismatch queries.

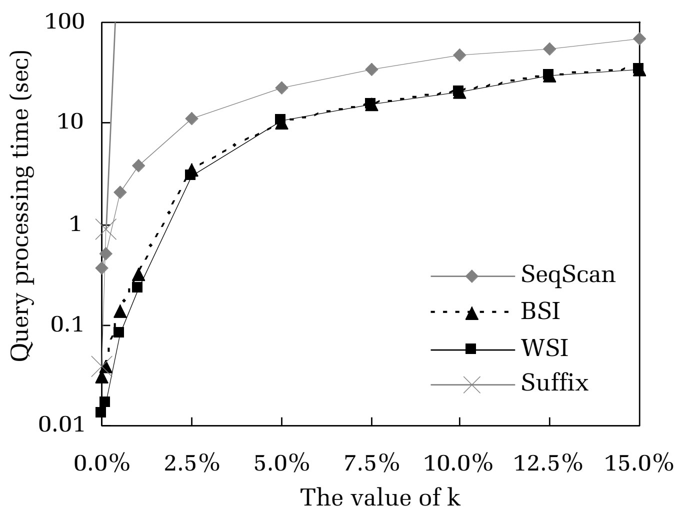

Fig. 8. Processing time of k-mismatch with various k values.

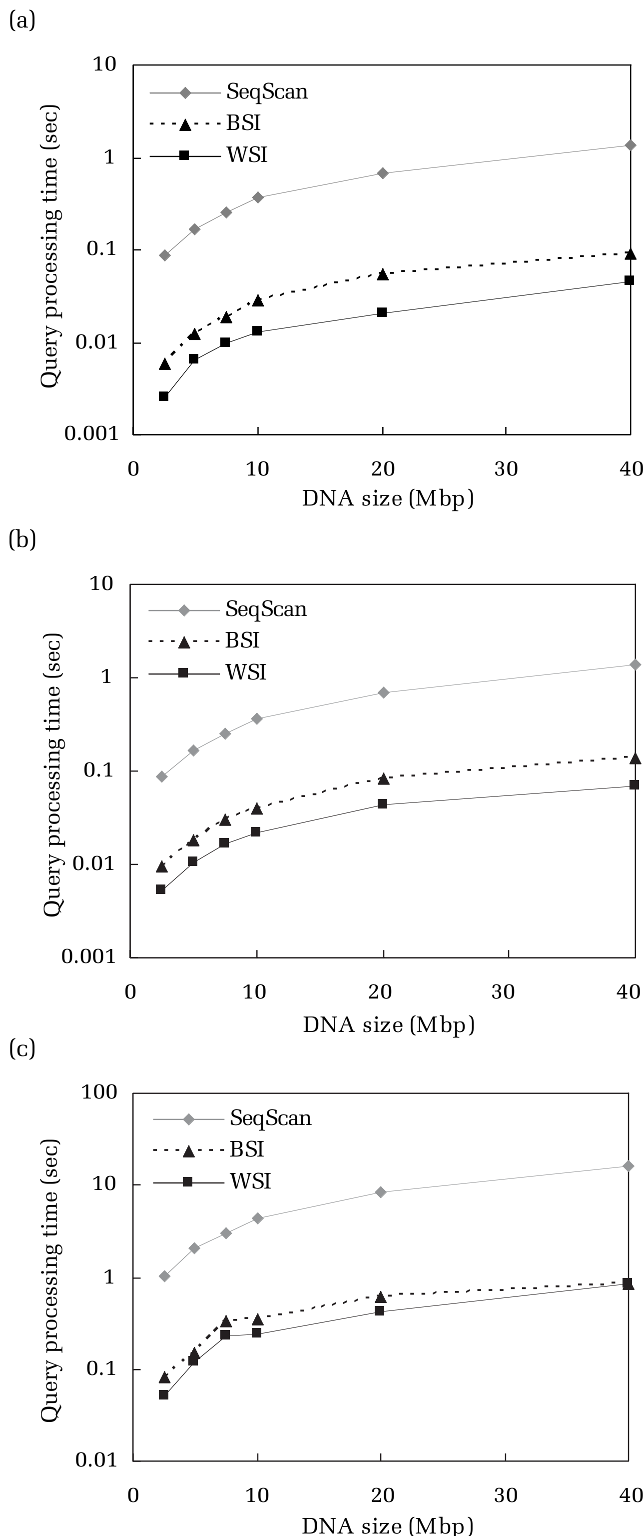

Fig. 9. Query processing time with various data sizes. Processing time of (a) exact match, (b) wildcard match and (c) k-mismatch queries.

The processing time of BSI and WSI for three kinds of queries increases almost linearly as the data size grows. WSI performs better than the other approaches in processing all kinds of queries. In exact match queries, WSI runs faster than SeqScan and BSI 25–33 times and 1.8–2.5 times, respectively. In wildcard match queries, WSI outperforms SeqScan and BSI 15–19 times and 1.7–1.9 times, respectively. Also, in *k*-mismatch queries, WSI performs better than SeqScan and BSI 13–20 times and 1.0–1.5 times, respectively.

7. Conclusions

Exact match queries, wildcard match queries, and *k*-mismatch queries are widely used in various molecular biology applications including the searching of ESTs (Expressed Sequence Tags) and DNA transcription factors. In this paper, we have discussed the method for the effective processing of such queries.

The method based on the Boyer–Moore algorithm, the method based on the Knuth–Morris–Pratt algorithm, and the method combining the Aho–Corasick algorithm and a scan vector have been proposed for exact and approximate matching problems. They focus on reducing the CPU cost needed to find the occurrences of a query pattern in data sequences. However, they have a limitation in improving search performance because they have to read the entire data sequence from a disk at the search stage. The method based on suffix trees achieves a rather good performance by adopting a filtering and refinement strategy. This method, however, incurs a large space overhead and also suffers from the high cost of traversing a large suffix tree.

In this paper, we have proposed an approach for processing exact and approximate queries that overcomes the problems mentioned above. We have suggested an effective indexing method for a set of nucleotide sequences. The method places a sliding window at every possible location of a data sequence, and extracts its signature by considering the occurrence frequency of each nucleotide character. It then stores and manages a set of signatures in a multi-dimensional index such as the R*-tree. In addition, by assigning a weight to every position of a window, it scatters the signatures over indexing space and thus reduces false alarms. Using the proposed indexing method, we have also suggested an algorithm which processes exact match queries, wildcard match queries, and *k*-mismatch queries efficiently. Experiments with real biological data sets reveal that the proposed method is at least 4.4 times, 2.1 times, and several orders of magnitude faster than the suffix-tree-based method in performing exact match, wildcard match, and *k*-mismatch queries, respectively.

Acknowledgements

This work was supported by the Korea Research Foundation Grant (KRF-2004-003-D00302), by the IT Research Center via Cheju National University and by KOSEF Basic Research Program Grant RO4-2003-000-10048-0.

References

[1] C. Gibas and P. Jambeck, *Developing Bioinformatics Computer Skills* (O'Reilly, Sebastopol, 2001).
[2] T. Smith and M. Waterman, Identification of common molecular subsequences, *Journal of Molecular Biology* 147(1) (1981) 195–7.
[3] S.F. Altschul, T.L. Madden, A.A. Schaffer, J. Zhang, Z. Zhang, W. Miller and D.J. Lipman, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *Nucleic Acids Research* 25(17) (1997) 3389–3402.
[4] S. Altschul, W. Gish, W. Miller, E. Myers and D. Lipman, Basic local alignment search tool, *Journal of Molecular Biology* 215(3) (1990) 403–0.
[5] T. Kaheci and A.K. Singh, An efficient index structure for string databases. In: P.M.G. Apers et al. (eds), *Proceedings of the 27th International Conference on Very Large Databases (VLDB01) 11–14 September 2001* (Morgan Kaufmann, San Francisco, 2001) 351–60.
[6] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* (Cambridge University Press, New York, 1997).
[7] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Communications of the ACM* 20(10) (1977) 762–72.
[8] D.E. Knuth, J.H. Morris, and V.B. Pratt, Fast pattern matching in strings, *SIAM Journal of Computing* 6(2) (1977) 323–50.
[9] A. Aho and M. Corasick, Efficient string matching: an aid to bibliographic search, *Communications of the ACM* 18(6) (1975) 333–40.
[10] G.A. Stephen, *String Searching Algorithm* (World Scientific Publishing, Singapore, 1994).
[11] U. Manber and G. Myers, Suffix arrays: a new method for on-line string searches, *SIAM Journal on Computing* 22(5) (1993) 935–48.
[12] V. Makinen, Compact suffix array: a space-efficient full-text index, *Fundamenta Informaticae* 56(1/2) (2003) 191–210.
[13] V. Makinen and G. Navarro, Compressed compact suffix

arrays. In: S.C. Sahinalp et al. (eds), *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM04) 5–7 July 2004* (Springer, Berlin, 2004) 420–33.

[14] K. Sadakane, Compressed text databases with efficient query algorithms based on the compressed suffix array. In: D.T. Lee and S.H. Teng (eds), *Proceedings of the 11th Algorithms and Computation (ISAAC00) 18–20 December 2000* (Springer, Berlin, 2000) 410–421.

[15] A. Guttman, R-Trees: a dynamic index structure for spatial searching. In: B. Yormark (ed.), *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (ACM SIGMOD) 18–21 June 2004* (ACM Press, New York, 1984) 47–57.

[16] R. Agrawal, C. Faloutsos and A. Swam, Efficient similarity search in sequence databases. In: D.B. Lomet (ed.), *Proceedings of the 4th International Conference of Foundations of Data Organization and Algorithms (FODO93) 13–15 October 1993* (Springer, Berlin, 1993) 69–84.

[17] R. Giegerich, S. Kurtz and J. Stoye, Efficient implementation of lazy suffix trees, *Software-Practice & Experience* 33(11) (2003) 1035–9.

[18] E. Hunt, M.P. Atkinson and R.W. Irving, Database indexing for large DNA and protein sequence collections, *VLDB Journal* 11(3) (2002) 256–71.

[19] R. Cole, L. Gottlieb and M. Lewenstein, Dictionary matching and indexing with errors and don't cares. In: L. Babai (ed.), *Proceedings of the 36th Annual ACM Symposium on Theory of computing (STOC04) 9–11 June 2003* (ACM Press, New York, 2004) 91–100.

[20] A. Amir, M. Lewenstein and E. Porat, Faster algorithms for string matching with k-mismatches. In: *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete algorithms(SODA00) 9–11 January 2000* (ACM/SIAM, New York, 2000) 794–803.

[21] S. Berchtold, D.A. Keim and H.-P. Kriegel, The X-tree: an index structure for high-dimensional data. In: T.M. Vijayaraman et al. (eds), *Proceedings of the 22nd International Conference on Very Large Databases (VLDB96) 3–6 September 1996* (Morgan Kaufman, San Francisco, 1996) 28–39.

[22] *The Entrez Nucleotides Database*. Available at: www.ncbi.nlm.nih.gov (accessed 13 September 2005).

[23] *The Ensembl Project*. Available at: www.ensembl.org (accessed 13 September 2005).

Appendix: derivation of the average number of disparate windows represented by the same basic signature

The average number of disparate windows which are represented by the same basic signature is computed as follows:

(1) Any one of four principal nucleotides can appear at every interior position of window W. Hence, there are $4^{|W|}$ windows whose contents are different from others.

(2) The basic signature of window W is represented by the occurrence frequency of each of four principal nucleotides and the sum of all occurrence frequencies should be $|W|$. Hence, the total number of distinct basic signatures is ${}_4H_{|W|}$. Here, H is a symbol denoting *permutation with repetition*.

(3) Combining the above two results, the average number of disparate windows which are represented by the same basic signature is expressed as $4^{|W|}/{}_4H_{|W|}$.