A DNA Index Structure Using Frequency and Position Information of Genetic Alphabet*

Woo-Cheol Kim[1], Sanghyun Park[1], Jung-Im Won[1],
Sang-Wook Kim[2], and Jee-Hee Yoon[3]

[1] Department of Computer Science,
Yonsei University, Korea
{twelvepp, sanghyun, jiwon}@cs.yonsei.ac.kr
[2] College of Information and Communications,
Hanyang University, Korea
wook@hanyang.ac.kr
[3] Division of Information Engineering and Telecommunications,
Hallym University, Korea
jhyoon@hallym.ac.kr

Abstract. Exact match queries, wildcard match queries, and k-mismatch queries are widely used in lots of molecular biology applications including the searching of ESTs (Expressed Sequence Tag) 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 overlapped with the rectangle.

Keywords: DNA database, indexing, 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 over the four-letter alphabet of A, C, G, and T known as nucleotides. *DNA sequence searching* is an operation that finds, from a DNA database, DNA (sub-)sequences whose nucleotide

* This work was supported by the Korea Research Foundation Grant (KRF-2004-003-D00302), the Basic Research Program Grant (Grant R04-2003-000-10048-0), and the IT Research Center via Cheju National University.

T.B. Ho, D. Cheung, and H. Liu (Eds.): PAKDD 2005, LNAI 3518, pp. 162–172, 2005.

arrangements are similar to a given query 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* [12], a dynamic programming approach for finding an optimal local alignment between two sequences. 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. A natural idea to resolve this kind of drawbacks is to employ the *filtering and refinement approach*. *BLAST* [4, 5] 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. [10] proposed the *MR-Index* for efficient processing of k-difference queries. A *k-difference query* is to find data subsequences that can be matched to a given query sequence by performing at most k replacing, inserting, and deleting operations.

In this paper, we proposes an approach for 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 are exactly matched to a query sequence. *Wildcard match queries* contain wildcard characters marked as '$*$' in a query sequence, and find the subsequences that are matched to a query sequence. Note that a wildcard matches with any single nucleotide. *K-mismatch queries* retrieve the data subsequences that have at most k nucleotides mismatched to those of a given query sequence. These queries are widely used in various molecular biology applications such as retrieval of expressed sequence tags and DNA transcription factors [8].

2 Definitions

The *alphabet* $\sum$ of nucleotides consists of 15 characters that can occur in DNA sequences (See Table1). 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 as *principal nucleotides*.

A *DNA sequence* $T = \langle t_1, t_2, \cdots, t_n \rangle$ is an ordered list of characters in the alphabet $\sum$. $|T|$ denotes the length of T. We use T' to denote a contiguous subsequence of T. A *window* is defined as a subsequence of a fixed length taken from a DNA sequence. W and $|W|$ denote a window and its length, respectively. The window beginning at the i^{th} position of a DNA sequence is denoted as W_i.

Any two characters s and q are said to be *matched* if the intersection of the set of characters represented by s and the set of characters represented by q is not empty. Given a DNA data sequence T and a query sequence Q, the *DNA sequence searching problem* is to find all subsequences T' of T that satisfy both of the following conditions: (1) $|Q| = |T'|$, and (2) for each i between 1 and $|Q|$, the i^{th} character of Q matches the i^{th} character of T'.

Table 1. Characters included in the alphabet of nucleotides

Code	Bases	Code	Bases	Code	Bases
A	A	Y	C or T	B	C or G or T
C	C	S	G or C	D	A or G or T
G	G	W	A or T	H	A or C or T
T	T	K	G or T	V	A or C or G
R	A or G	M	A or C	N	any base

3 Related Work

The *Boyer-Moore algorithm* [7] and the *Knuth-Morris-Pratt(KMP) algorithm* [11] have been devised for exact match queries. Their worst-case time complexity proved to be linear to the length of data sequence. These algorithms, however, should access the entire data sequences from disk because they are based on the sequential scan.

The method combining the *Aho-Corasick algorithm* [3] and the *scan vector* has been proposed for processing wildcard match queries [8]. 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 an 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 the scan vector as large as the data sequence. Also, it requires much processing time because it accesses the whole data sequences from disk.

For processing k-mismatch queries, the *suffix-tree-based method* [13] 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 from a large storage overhead and high cost for maintaining and traversing a huge suffix tree.

4 Basic Signature Index

This section proposes a new indexing method called BSI (Basic Signature Index) and also suggests a query processing method based on the proposed index structure.

To construct an index, 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 1. *Basic Signature: BS*
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.

$BS(W_i)$ is regarded as a 4-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 [9] and the X-tree [6]. The total number of windows taken from a data sequence T is $|T| - |W| + 1$. Since $|T| \gg |W|$ in most cases, the index for T could be much larger than T itself.

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 consecutive two data windows are not that different from each other and thus are located closely in the 4-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*.

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

- **Exact match query:** We construct a 4-dimensional query rectangle, ([min_A, max_A], [min_C, max_C], [min_G, max_G], [min_T, max_T]), from the query sequence.
- **Wildcard match query:** We first construct a 4-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 on the query sequence.
- **K-mismatch query:** We construct a 4-dimensional query rectangle by using the procedure for wildcard match queries. We then increase max_A, max_C, max_G, and max_T 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 each principal nucleotide in a data window is allowed to occur k times more or less than that in a query signature by k mismatches. If an adjusted minimum value becomes less than 0, we set it to 0.

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.

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 assumed $|Q| = |W|$. When $|Q| < |W|$, we generate a new query sequence Q' of length $|W|$ by appending $|W| - |Q|$ wildcard characters '$*$' to the end of Q and then apply the above query processing procedure to Q'. When $|Q| > |W|$, we first partition a query sequence Q into p sub-query sequences, $Q_1, Q_2, \cdots, and\ Q_p$, such that $p = \lceil |Q|/|W| \rceil$ and $|Q_i| = |W|$ for every i between 1 and p. Here, the last sub-query sequence Q_p can be overlapped with Q_{p-1} to make the constraint $|Q_i| = |W|$ satisfied. Next, we apply the above query processing procedure to every sub-query sequence, and then obtain the final answers by merging all the results.

5 Weighted Signature Index

Let us first mention a couple of drawbacks of BSI. First, in BSI, the signature of a window is decided only by the number of occurrences of each principal nucleotide. Therefore, there may be a great number of windows that are different from one another but are represented as the same signature. It causes a large number of false alarms, resulting in high index-searching and post-processing costs. Second, 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. The windows taken from such sequences also show similar occurrence ratios regardless of their beginning positions. Therefore, it is likely that lots of windows are represented by the signatures close to the center $(0.3 \times |W|,\ 0.2 \times |W|,\ 0.2 \times |W|,\ 0.3 \times |W|)$.

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

5.1 Basic Strategy

The simplest way to overcome the limitations of BSI is to extract more features from windows. However, this increases the dimensionality of the underlying index, and thus leads to the well-known dimension curse. To represent windows more discriminatively 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 $w(j)$ $(1 \leq j \leq |W|)$ which assigns a weight to each position i within a window. We then extract a weighted signature from each window.

Definition 2. *Weighted Signature: WS*
Let $WS(W_i)$ be a weighted signature of window W_i. $WS(W_i)$ 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.

By taking the above weighting scheme, disparate windows that were represented by the same basic signature may now be expressed by different weighted signatures. We incorporate this weighing scheme into the proposed index structure, 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 the indexing space.

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, we need to consider the positions at which mismatches may occur. The procedure to build a query rectangle for a k-mismatch query is skipped due to space limitation.

5.2 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 the indexing space by the weight function. Let us consider an MBR that covers all such weighted signatures. Larger MBR implies that the weighted signatures are scattered over larger space. However, if the weighted signatures are scattered too much, 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 a 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 j^{th} smallest weight in a window. Similarly, let $R_{max}(X, s)$ denote the maximum of all $wmax_X$ values obtained from a 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 being overlapped, $R_{max}(X, s) < R_{min}(X, s+1)$ should be satisfied for every s between 0 and $|W|-1$. Supposing $w(j) = j + C$, let us solve the inequality. Note that $sw(j)$ is identical to $w(j)$ in this case.

$$
\begin{aligned}
& R_{max}(X,s) \;<\; R_{min}(X,s+1) \\
\Leftrightarrow\; & \textstyle\sum_{j=|W|-s+1}^{|W|} sw(j) \;-\; \sum_{j=1}^{s+1} sw(j) \;<\; 0 \\
\Leftrightarrow\; & \textstyle\sum_{j=|W|-s+1}^{|W|} w(j) \;-\; \sum_{j=1}^{s+1} w(j) \;<\; 0 \\
\Leftrightarrow\; & -C \;-\; (s^2 + (1-|W|)s + 1) \;<\; 0
\end{aligned}
$$

Since the above inequality should be satisfied for every s between 0 and $|W|-1$, we obtain $C > \frac{(|W|-1)^2}{4} - 1$. Among the values of C which satisfy the inequality, we choose $|W|^2$ for the sake of simplicity. That is, we use $w(j) = j + |W|^2$ for a weight function.

6 Performance Evaluation

In our experiments, as a data sequence T, we used six sets of DNA sequences downloaded from NCBI [1]: human chromosome 3 (2.5Mbp), 17 (5Mbp), 1 (7.5Mbp), 2 (10Mbp), 10 (20Mbp), and 5 (40Mbp). As a query sequence, we used 1,000 DNA sequences of length 256 to 2,048. A half of them were randomly selected from T, and the other half were obtained from DNA sequences [2] frequently used by biologists at laboratories.

We evaluated performances of four approaches: BSI, WSI, SeqScan, and Suffix. SeqScan is the sequential scan based method, and Suffix is the method that uses the suffix tree as an index structure.

6.1 Parameter Settings

It is desirable to set the window size slightly smaller than a typical size of a query sequence. For determining a window size, we analyzed the lengths of 35,685 query sequences downloaded from [2]. From the results, we observed that 62% of them have the lengths of 256 to 2,048. Thus, we set the basic window size to 256 for further experiments.

In order to find a good value for the index compression coefficient, while changing the index compression coefficient, we evaluated the k-mismatch query processing time of BSI and WSI using human chromosome 2 of 10Mbp as a data sequence and 1% of the length of a query sequence as the value of k. As shown in Fig. 1, as the compression coefficient increases up to 80, the query processing time of both BSI and WSI decreases. From that point, however, their query processing time increases as the compression coefficient gets larger. Therefore, we set the base value for the compression coefficient to 80.

6.2 Results and Analyses

Experiment 1: Query Processing Time with Various Query Size
In this experiment, we compared query processing times of different approaches while changing the length of query sequences. We used human chromosome 2 of 10Mbp as a data sequence. Also, we set both k for k-mismatch queries and the number of wildcard characters for wildcard match queries to 10, which is 1% of

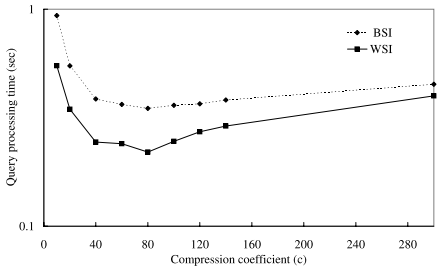


Fig. 1. Query processing time with various values for compression coefficient

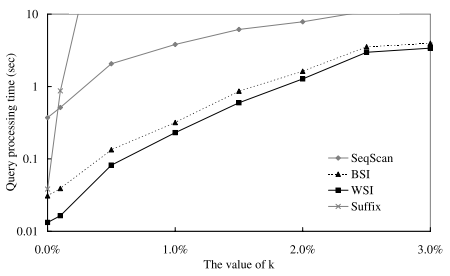


Fig. 2. Processing time of k-mismatch with various k values

the average length of query sequences. Fig. 3 depicts query processing times of all the approaches for exact match, wildcard match, and k-mismatch queries.

In exact match queries, SeqScan and Suffix show nearly constant performance regardless of the length of query sequences. In BSI and WSI, we observe that the query processing time shrinks until the length of a query sequence reaches a point (i.e., 512), and then grows gradually after that point.

In wildcard match queries, every approach spends more query processing time compared with that of exact match queries. In BSI and WSI, wildcard characters in a query enlarge the corresponding query rectangle and increase the number of candidates, which leads to a large query processing time. As a query sequence gets longer, however, the number of candidates decreases remarkably. Thus, the performance improves significantly.

K-mismatch queries take a processing time much bigger than exact match and wildcard match queries. In particular, Suffix shows performance worse than even SeqScan since the part of the index to be traversed increases explosively. In BSI and WSI, however, their performance is shown to be nearly constant, and is not that affected by the changes of the length of query sequences.

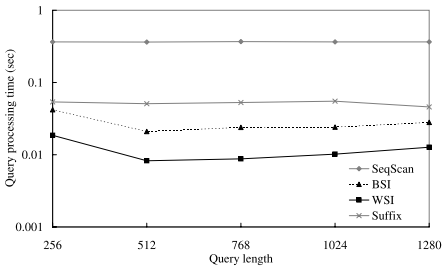
In exact match queries, the results show that WSI outperforms SeqScan, Suffix, and BSI 19 to 44 times, 2.9 to 6.1 times, and 2.2 to 2.7 times, respectively. In wildcard match queries, WSI performs better than SeqScan, Suffix, and BSI 4 to 21 times, 1.4 to 4.5 times, and 1.5 to 1.8 times, respectively. Also, in k-mismatch queries, BSI performs faster than SeqScan, Suffix, and BSI 7 to 28 times, several thousand times, and 1.3 to 1.6 times, respectively.

Experiment 2: Processing Time of k-mismatch with Various k Value
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 10Mbp as a data sequence. Fig. 2 shows an average query processing time of each approach while setting k as 0%, 1%, 2%, and 3% 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 makes the part of an index to be traversed increased, and thus increases the query processing time gradually. In Suffix, however, as k grows, the part of an index to be

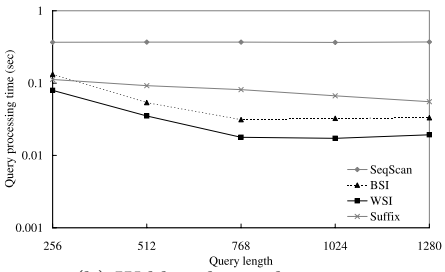
traversed becomes explosively larger, and thus, the query processing time grows abruptly. The results reveal that WSI shows the best performance, and performs better than SeqScan, Suffix, and BSI 3.6 to to 31 times, 3 to several thousand times, and 1.1 to 2.3 times, respectively.

Experiment 3: Query Processing Time with Various Lengths of Data Sequences

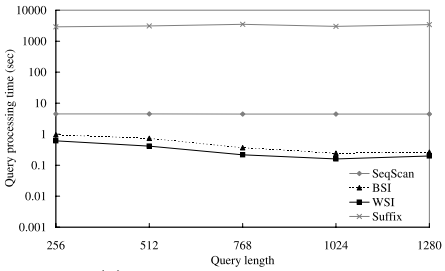
In this experiment, we measured the query processing times of different approaches with various data sizes. We excluded Suffix in this experiment since its performance degradation in performing k-mismatch queries on a large database is too serious to conduct experiments. Here, we set both k for k-mismatch queries



(a) Exact match queries

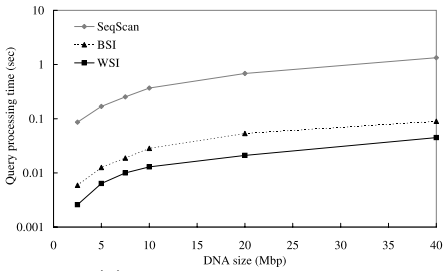


(b) Wildcard match queries

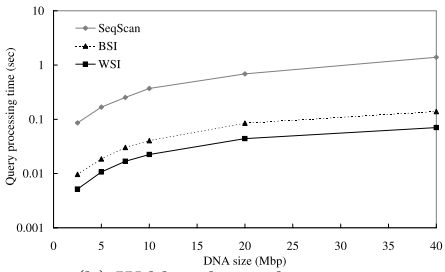


(c) K-mismatch queries

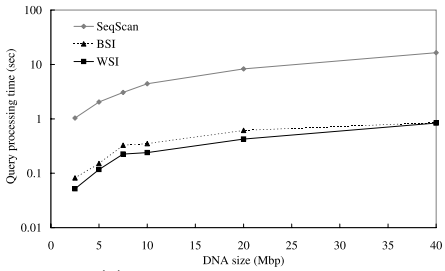
Fig. 3. Query processing time with various lengths of query sequences



(a) Exact match queries



(b) Wildcard match queries



(c) K-mismatch queries

Fig. 4. Query processing time with various data sizes

and the number of wildcard characters for wildcard match queries to 10, which is 1% of the average length of query sequences. Fig. 4 shows an average processing time of each approach for exact match, wildcard match, and 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 to 33 times and 1.8 to 2.5 times, respectively. In wildcard match queries, WSI outperforms SeqScan and BSI 15 to 19 times and 1.7 to 1.9 times, respectively. Also, in k-mismatch queries, WSI performs better than SeqScan and BSI 13 to 20 times and 1.0 to 1.5 times, respectively.

7 Conclusion

Exact match queries, wildcard match queries, and k-mismatch queries are widely used in lots of molecular biology applications including the searching of ESTs (Expressed Sequence Tag) and DNA transcription factors.

In this paper, we proposed an efficient indexing and processing technique for processing such queries on large DNA databases. The proposed indexing 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 using a multi-dimensional index, such as R*-tree. Especially, by assigning a weight to each position of a window, it scatters the signatures over the index space and thus reduces false alarms. The experiments with real biological data sets revealed that the proposed method is at least 2.9 times, 1.4 times, and several orders of magnitude faster than the suffix-tree-based method in performing exact match, wildcard match, and k-mismatch queries, respectively.

References

1. http://www.ncbi.nlm.nih.gov
2. ftp://ftp.ensembl.org
3. A. Aho, M. Corasick, "Efficient string matching: an aid to bibliographic search", Communications of the ACM, Vol. 18, pp. 333-40, 1975.
4. 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.
5. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool", Journal of Molecular Biology, Vol. 215, pp. 403-410, 1990.
6. S. Berchtold, D.A. Keim, and Hans-Peter Kriegel, "The X-tree: An index structure for high-dimensional data", VLDB, pp 28-39, 1996.
7. R.S. Boyer, J.S. Moore, "A fast string searching algorithm", Communications of the ACM, Vol. 20, pp. 762-772, 1977.
8. D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

9. A. Guttman, "R*-Trees, A dynamic index structure for spatial searching", ACM SIGMOD, pp. 47-57, 1984.
10. T. Kaheci, A. K. Singh, "An efficient index structure for string databases", VLDB, 2001.
11. D. E. Knuth, J. H. Morris, V. B. Pratt, "Fast pattern matching in strings", SIAM J. Comput., Vol. 6, pp. 323-350, 1977.
12. T. Smith and M. Waterman, "Identification of common molecular subsequences", Journal of Molecular Biology, Vol. 147, pp. 195-197, 1981.
13. G. A. Stephen, String Searching Algorithm, World Scientific Publishing, 1994.