# A Practical Method for Approximate Subsequence Search in DNA Databases

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

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

**Abstract.** In this paper, we propose an accurate and efficient method for approximate subsequence search in large DNA databases. The proposed method basically adopts a *binary trie* as its primary structure and stores all the window subsequences extracted from a DNA sequence. For approximate subsequence search, it traverses the binary trie in a breadth-first fashion and retrieves all the matched subsequences from the traversed path within the trie by a dynamic programming technique. However, the proposed method stores only window subsequences of the pre-determined length, and thus suffers from large post-processing time in case of long query sequences. To overcome this problem, we divide a query sequence into shorter pieces, perform searching for those subsequences, and then merge their results.

**Keywords:** DNA database, approximate subsequence search, suffix tree.

## 1 Introduction

Since the size of DNA databases is increasing considerably in these days, methods of fast indexing and query processing are essential for efficient DNA subsequence search. The *suffix tree* [4] has been known to be a good index structure for DNA subsequence search. Recently, there have been many research efforts on efficient construction and query processing with suffix trees [5][10][4]. The suffix tree still has the following drawbacks due to its structural features [3][4][11]: (1) high storage overhead, (2) poor locality in disk accesses, and (3) difficulty in seamless integration with DBMS.

In this paper, we propose a novel index structure that supports DNA subsequence search efficiently and also resolves the drawbacks of the suffix tree

Z.-H. Zhou, H. Li, and Q. Yang (Eds.): PAKDD 2007, LNAI 4426, pp. 921–931, 2007.
© Springer-Verlag Berlin Heidelberg 2007

mentioned above. The proposed index structure basically adopts a trie [4] as its primary conceptual structure and realizes the trie by pointerless binary bit-string representation [13]. It extracts subsequences of the pre-determined length from every possible position of a DNA sequence, and stores only those subsequences in the index. They are called *window subsequences* and their length is usually much smaller than the average length of all the suffixes within a DNA sequence. This method is devised based on the observation that the length of longest common prefixes among suffixes in a DNA sequence is fairly small.

The DNA subsequence search with the proposed method uses the dynamic programming technique [4] and finds all the similar subsequences that exist on the paths of a binary trie. By traversing the trie index in a breadth-first fashion, it accesses each related page within the index only once. However, the proposed method stores only the window subsequences of the pre-determined length, and thus suffers from large post-processing time in case of long query sequences. To overcome this problem, we divide such a long query sequence into shorter ones, and then perform subsequence search for each of them. This alleviates the problem of performance degradation even with long query sequences.

## 2 Related Work

The performance of DNA subsequence search can be improved by exploiting indexing mechanisms. The methods proposed in references [1][2][12] employ the inverted index, which has been frequently applied in the area of information retrieval. They extract *words*, fixed length intervals overlapped with one another, from every sequence, and build a posting list of $<sequence\ number, offset>$ for each word. The method proposed in reference [6] maps every subsequence into a point in multidimensional space by the wavelet transform, and then constructs a multidimensional index on those points. By using the index, it processes range queries and nearest neighbor queries. This method enjoys nice search performance owing to a relatively small size of the index.

The suffix tree [4] is an index in a form of a persistent tree, and has been widely used in DNA subsequence search. Previously, it is not easy to construct a disk-resident suffix tree whose size is larger than that of main memory. Recently, reference [5] proposed a method for suffix tree construction by using the concept of partitioned suffix trees. Also, reference [10] proposed a top-down disk-based approach for efficient construction of suffix trees. Reference [7] proposed an approach for similar subsequence search that returns the results in the similarity based order by using dynamic programming and the A*-algorithm. However, the performance of approximate subsequence search with the suffix trees deteriorates as the length of a query sequence or a tolerance increases. A query partitioning method was proposed to solve this problem [8]. It partitions a given long sequence into shorter ones, and performs subsequence search for each of them with a smaller tolerance. Then, it merges the results thus obtained from all the subsequence searches.

## 3 Indexing Method

The suffix tree, which is a compressed digital trie built on all the suffixes of given sequences, has been known to be a good index structure for DNA subsequence search [4]. The suffix tree can compress input data sequences substantially when they have a lot of common prefixes. A DNA sequence can be considered as a string from the alphabet $\sum = \{A, C, G, T\}$. Since the size of the alphabet is very small (which is 4), it is likely that there exist a considerable number of common prefixes in the suffixes of input sequences. However, *longest common prefixes*$(LCP)$ in the suffixes extracted from DNA sequences are commonly very short.

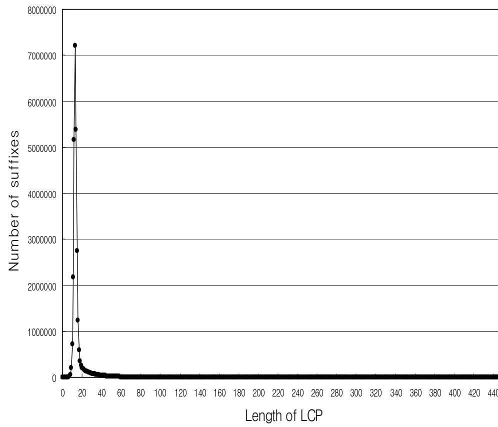


**Fig. 1.** Distribution of LCPs lengths

| Symbol | Binary Code |
|---|---|
| $ | 000 |
| A | 001 |
| C | 010 |
| G | 011 |
| N | 100 |
| T | 101 |
| S | 110 |
| Y | 111 |

**Fig. 2.** Binary codes for symbols in the alphabet

| Subsequence | Binary Representation |
|---|---|
| ACGA | 001010011001 |
| CGAC | 010011001010 |
| GACT | 011001010101 |
| ACT$ | 001010101000 |
| CT$$ | 010101000000 |
| T$$$ | 101000000000 |

**Fig. 3.** Binary representation of all the window subsequences from $S =$ 'ACGACT'

Figure 1 shows distribution of lengths of LCPs in suffixes extracted from a DNA sequence. We have used a 28.6Mbp DNA sequence in human chromosome 21 for the analysis. We have observed that the average and maximum lengths of LCPs in suffixes are 15 and 451, respectively, and that the number of suffixes that share LCPs whose length is 13 is largest(about 7.2 millions). Also, most suffixes share LCPs of a length 11 to 15, and 82.6% of LCPs have a length 1 to 15.

Based on this observation, we build an index not on all the suffixes extracted from a DNA sequence but only on their prefixes with a pre-determined length. That is, we place a sliding window of the length $|W|$ at every possible position of a DNA sequence, extract the subsequences covered by all the windows from a DNA sequence, and then insert them into the trie. We call these subsequences *window subsequences*. From our LCP analysis, we set the length $|W|$ as 15. We extract $|S|$ window subsequences from a DNA sequence $S$. The indexing with such window subsequences contributes to decrease the index size significantly and also makes the search of a leaf node simplified.

To represent all the symbols in the alphabet, we use the minimum number of bits instead of using one byte, thereby achieving high compression ratio. Figure 2 shows binary codes to represent all the symbols in DNA sequences. Here, N, S, and Y denote wild-card characters [12] and '\$' denotes a special character used for padding to make all the window subsequences have a length of $|W|$. Given sequence $S$ = 'ACGACT', we extract window subsequences whose length is 4 from $S$, allocate 3 bits for each symbol as shown in Figure 2, and represent each window subsequence into a corresponding binary bit-string as shown in Figure 3.

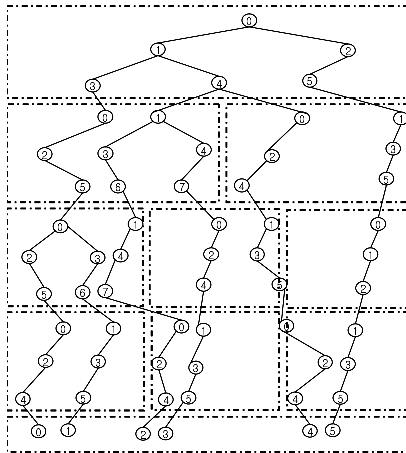

**Fig. 4.** Binary trie constructed from the window subsequences in Figure 3

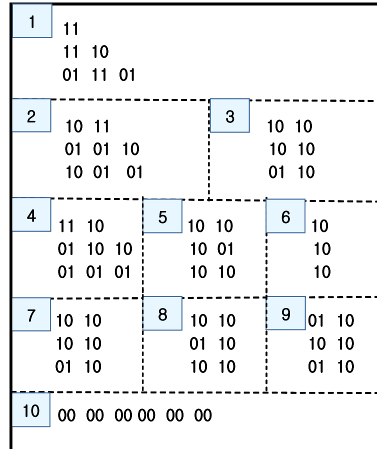


**Fig. 5.** Internal representation of the binary trie in Figure 4

In earlier work [13], we proposed a disk based index structure for efficient DNA sequence matching, exploiting the basic concept of pointerless binary tries. Pointerless binary tries require an alphabet to have only two symbols of 0 and 1. This makes every node have at most two outgoing edges. In this representation, the symbols on edges do not need to be stored explicitly if the following rules are enforced: (1) the outgoing edge labeled with 0 is assumed to connect to the left child node, and (2) the outgoing edge labeled with 1 is assumed to connect to the right child node. Our index structure basically adopts the binary trie as its primary conceptual structure. It consists of three parts: a binary trie, a page table, and a leaf table. The binary trie is an index structure storing all the window subsequences extracted from a DNA sequence. The page table stores the link information for pages within the binary trie. The leaf table stores the starting offsets of window subsequences within a DNA sequence. Figure 4 shows a binary trie constructed from the window subsequences of Figure 3. Here, node numbers in the trie are determined by the order of nodes being written into a disk page. Figure 5 shows its internal representation. The node structure is represented by a two-bit number and then is written into an appropriate page. In Figures 4 and 5, the rectangles of dotted lines represent pages stored in disk.

## 4 Query Processing Method

To discover all the subsequences similar to a query sequence $Q$, most similarity search algorithms [5][7][8][4] based on a suffix tree traverse the tree in a depth-first fashion and, during the traversal, they build a dynamic programming (DP) table [4] using $Q$ as its Y-axis and the sequence on the path from the root to the node being visited currently as its X-axis. We could apply such similarity search algorithms to binary tries. However, the proposed trie index contains only a two-bit number of each node. As a result, pointers from parents to their children, node levels, and subsequences on the paths from the root can not be extracted directly from the proposed trie index. Therefore, we need to uncover this implicit information whenever reading a new page during the traversal of the proposed binary trie. In addition, our binary trie is a disk-based index structure where the nodes on the same level are stored consecutively within a disk page. As a consequence, when we traverse all paths of the binary trie, we may access the same node multiple times within a single page and/or the same disk page more than once.

To solve the problem of accessing the same nodes and/or same disk pages multiple times, we propose to traverse the binary trie in a breadth-first order. That is, by visiting the nodes of the binary trie in a breadth-first fashion, our proposed Search-Trie() shown in Algorithm 1 effectively finds all the subsequences whose edit distances to a query sequence $Q$ are not larger than a distance tolerance $T$.

Let us explain briefly how Search-Trie() operates. The algorithm employs two queues, Qpagenumber for examining data pages sequentially and Qnode for visiting the trie nodes of a current page one by one. The whole algorithm consists of two 'while' loops, an outer loop for data pages and an inner loop for trie nodes of a current page. For each child node $CN_i$ of a current node $current_Node$, we execute the following steps (Lines 7-19). First of all, we assign TRUE to variable $moreVisit$ which indicates whether or not we need to traverse the index further downwards (Line 8). Function AppendBitString() creates $CN_i_Path$, the path from the root to node $CN_i$, by extending the path from the root to node $current_Node$ into node $CN_i$ (Line 9). If the length of $CN_i_Path$ becomes a multiple of 3, we compose a new symbol by aggregating the last 3 bits of $CP_i_Path$ and then call function AddColumn() (Line 11). Function AddColumn() adds a column for the new symbol to the DP table constructed so far (i.e., $current_DPT$), which results in a new DP table $DPT_CN_i$.

Let $dist$ be the value at the last row of the last column of DP table $DPT_CN_i$ (Line 12). If $dist$ is not larger than distance tolerance $T$, all the subsequences containing the sequence on path $CN_i_Path$ as their prefixes should be included in an answer set. Therefore, in such a case, we call function FindAnswers() where all leaf nodes under node $CN_i$ are retrieved with their sequence and offset information. After that, we assign FALSE to variable $moreVisit$ in order to indicate more extension of path $CN_i_Path$ is unnecessary (Line 15).

On the contrary, if $dist$ is larger than distance tolerance $T$, we call function FurtherVisit() which determines whether or not we have to go down under $CN_i$. Lines 18 and 19 are executed only when variable $moreVisit$ is TRUE. If node

$CN_i$ is a leaf, we cannot decide if $CN_i$ is an actual answer and thus should perform the post-processing of $CN_i$ by executing function FindCandidateAnswer() (Line 18). This step is necessary for processing query sequences longer than window subsequence $W$. If node $CN_i$ is not a leaf, we push node $CN_i$ onto Qnode by calling function CheckpageAndPush() and continue the execution of the algorithm. Note that, if the number of nodes already processed within a current page reaches the maximum number of nodes (i.e., $maxNode$) that can be stored within a single page, we locate the next data page by looking up page table $P$ and then push it onto Qpagenumber and its root node onto Qnode.

Since the binary trie has been built from a set of window subsequences of a fixed length, function FindCandidateAnswer() has to be executed when query sequences are longer than the window subsequences. However, in most cases,

**Algorithm 1.** Query processing algorithm Search-Trie

```
Input : binary trie I, query sequence Q, tolerance T, page table P, leaf table
        L, maxNode M
Output: set of answers answerSet
1  push(Qpagenumber, Root_pageNumber);
2  push(Qnode[Root_pageNumber], RootNode);
3  while notEmpty(Qpagenumber) do
4      pageNumber = pop(Qpagenumber);
5      while notEmpty(Qnode[pageNumber]) do
6          current_Node = pop(Qnode[pageNumber]);
7          for each child node CN_i of the current_Node do
8              moreVisit = TRUE;
9              AppendBitString(CN_i_Path, current_Node, CN_i);
10             if BitCount(CN_i_Path) mod 3 == 0 then
11                 DPT_CN_i = AddColumn(current_DPT, CN_i_Path);
12                 Let dist be the last row value of the new added column;
13                 if dist <= T then
14                     answerSet = answerSet ⋃ FindAnswer(CN_i, L);
15                     moreVisit = FALSE;
16                 else
                       moreVisit = FurtherVisit(DPT_CN_i);
17             if moreVisit then
18                 if terminal_Node(CN_i) then
                       answerSet = answerSet ⋃ FindCandidateAnswer(CN_i, L);
19                 else
                       CheckpageAndPush(Qpagenumber, Qnode, CN_i, P, M);
```

the number of candidate answers grows quickly as $|Q| - |W|$ becomes larger. In this paper, we propose to use a partition-based query processing [8] which circumvents this situation by decomposing a long query sequence into multiple pieces and then treating each piece as a separate query.

The proposed partition-based query processing algorithm is shown in Algorithm 2. Function Search-Trie-By-SubQuery() partitions a query sequence $Q$ into $p$ subqueries of appropriate lengths (Line 1). The number of subqueries and the length of each subquery are determined by considering how the performance of function Search-Trie() changes with respect to the length of a query sequence. For each subquery $SQ_i$ obtained in the previous step, we perform the similarity-based searching by calling function Search-Trie() of Algorithm 1 (Lines 2-3). Note that the distance tolerance of each subquery is adjusted to $\lfloor T/p \rfloor$. At last, we construct a final answer set after executing function postProcessing() with a set of candidate answers *candidateSet* (Line 4). When offset $i$ is given as a candidate answer, the post-processing step retrieves the corresponding data subsequence $S[i - |Q| - T, ..., i + |Q| + T]$ and computes its distance to $Q$ using dynamic programming.

**Algorithm 2.** Query processing algorithm Search-Trie-By-SubQuery

**Input** : binary trie $I$, query sequence $Q$, tolerance $T$, page table $P$, leaf table $L$, maxNode $M$
**Output**: set of answers answerSet

```
1 p = partitionQuery(Q, T);
2 for each subquery SQ_i do
3 └ candidateSet = candidateSet ⋃ Search-Trie(I, SQ_i, ⌊T/p⌋, P, L, M);
4 answerSet = postProcessing(candidateSet, Q, T);
5 return answerSet;
```

## 5 Performance Evaluation

In this section, we show the effectiveness of our approach via performance evaluation with extensive experiments. We compared the performances of the three approaches Search-Trie, Suffix, and SW: (1) Search-Trie represents our approach that employs the pointerless binary trie as an index structure. Note that the window size is 15 (i.e., $|W| = 15$). (2) Suffix is an existing approach based on the suffix tree. We implemented the suffix tree by utilizing the source code of the TDD (Top-Down Disk-Based) technique [10] downloaded from http://www.eecs.umich.edu/tdd. (3) SW is the Smith-Waterman algorithm [9] based on dynamic programming. As a data set, we used two *Homo sapiens chromosome sequences*, chromosome 21 (chr 21) of 28.6 Mbps and chromosome 19 (chr 19) of 56Mbps. The hardware platform is the Pentium IV 3.2GHz PC equipped with 1 Gbytes main-memory. The software platform is Redhat Linux 9 (Kernel Version 2.4.20).

| Data Size | Suffix | | | | Search-Trie | | | |
|---|---|---|---|---|---|---|---|---|
| | Tree | Leaf_ Table | Rmost_ Table | Total | Binary_ Trie | Leaf_ Table | Page_ Table | Total |
| 28.6Mbp (Chr 21) | 267.6M | 46.4M | 46.4M | 360.4M | 50.2M | 114.4M | 0.5M | 165.1M |
| 56Mbp (Chr 19) | 539M | 91M | 91M | 721M | 71.9M | 224.1M | 0.7M | 296.7M |

**Fig. 6.** Index sizes of the two approaches with increasing data set sizes

In Experiment 1, we compared Search-Trie with Suffix in the respect of an index size. Figure 6 summarizes the size of each index component of the two approaches with changing data sizes. From the experimental result, we observe that the index size increases linearly in proportion to the data size in both approaches. However, in comparison with Suffix, the proposed Search-Trie saves about 40% storage space.

In Experiment 2, we compared Search-Trie and Suffix in the respect of the elapsed time for approximate subsequence search. The total elapsed time is the time spent in finding all the subsequences whose edit distances to a query sequence are not larger than tolerance $T$. We also examined the total number of hits returned by Search-Trie and Suffix.

| Query Length \|Q\| | Data Size = 28.6Mbp | | | Data Size = 56Mbp | | |
|---|---|---|---|---|---|---|
| | Total hits | Query Processing Time(sec) | | Total hits | Query Processing Time(sec) | |
| | | Search-Trie | Suffix | | Search-Trie | Suffix |
| 10 | 4010 | 0.71 | 4.91 | 10641 | 0.8 | 7.22 |
| 20 | 137 | 2.03 | 12.81 | 2619 | 2.43 | 21.91 |
| 30 | 235 | 5.23 | 18.25 | 1815 | 9.48 | 35.42 |
| 40 | 77 | 30.96 | 28.08 | 848 | 47.32 | 48.08 |
| 50 | 52 | 182.33 | 36.26 | 712 | 279.67 | 63.16 |
| 60 | 82 | 868.85 | 57.80 | 376 | 1362.64 | N/A |

**Fig. 7.** Query processing times of the two approaches with increasing data set sizes and query lengths

| Tolerance \|T\| | Data Size = 28.6Mbp | | | Data Size = 56Mbp | | |
|---|---|---|---|---|---|---|
| | Total hits | Query Processing Time(sec) | | Total hits | Query Processing Time(sec) | |
| | | Search-Trie | Suffix | | Search-Trie | Suffix |
| 1 | 33 | 0.33 | 5.32 | 175 | 0.52 | 9.18 |
| 2 | 70 | 1.55 | 12.65 | 675 | 2.55 | 22.74 |
| 3 | 235 | 5.14 | 18.08 | 1815 | 9.51 | 35.28 |
| 4 | 490 | 23.73 | 26.58 | 3960 | 50.84 | 48.70 |
| 5 | 936 | 130.59 | 34.05 | 7614 | 300.66 | 61.84 |
| 6 | 1659 | 604.75 | 46.96 | 13596 | 1404.52 | N/A |

**Fig. 8.** Query processing times of the two approaches with increasing data set sizes and tolerance values

Figure 7 shows the elapsed times of approximate subsequence search by Suffix and Search-Trie with various query sequence lengths. The distance tolerance $T$ is set to 10% of the length of the query sequence. Search-Trie outperforms Suffix when query sequences are not so long. Search-Trie runs 4 to 9 times faster than Suffix when query sequences are shorter than 40. However, the performance of Search-trie deteriorates as query sequences get longer. This is because Search-Trie generates many candidate answers, which result in much time being spent for post-processing when $|Q| \ll |W|$. Next, Figure 8 shows the elapsed times of

approximate subsequence search by Suffix and Search-Trie with various tolerance values. The query sequence length is fixed to 30 in this experiment. Search-Trie outperforms Suffix when tolerance values are not so large. Compared with Suffix, Search-Trie is about 4 to 17 times faster when tolerance values are less than 4. However, the performance of Search-trie deteriorates as tolerance values get larger. This results from the fact that the number of candidate answers increases as tolerance values become larger.

On the other hand, Suffix shows better performance for long query sequences or large tolerance values. For large DNA sequences, however, Suffix becomes impractical when query sequences are too long or tolerance values exceed a certain threshold. This is because, in such cases, Suffix has to traverse a huge index structure in a depth-first order and therefore needs to access a large number of index pages repeatedly. The experimental result reveals that Suffix on the data sequence of 56 Mbps (i.e., chr 19) cannot handle the cases of either ($|Q| = 60$ and $|T| = 6$) or ($|Q| = 30$ and $|T| = 6$).

In Experiment 3, we compared the performance of Seach-Trie-By-SubQuery with that of Search-Suffix-By-SubQuery. Seach-Trie-By-SubQuery denotes our approximate subsequence search approach based on Search-Trie and a query partitioning method with optimal $p$ values. Search-Suffix-By-SubQuery denotes another approximate subsequence search approach based on Suffix, instead of Search-Trie, with the same query partitioning method. We also included the traditional algorithm SW in this experiment. When we apply the query partitioning method to Suffix or Search-Trie, we determine optimal $p$ values by considering both the performance of index searching and the overhead of post-processing. To select optimal $p$ values, we utilized results of Experiment 2.

Figure 9 shows the total elapsed times of the three approximate subsequence search approaches. The data sequences used in this experiment is chr 19 of 56Mbps. Also, the tolerance value $T$ is set to 10% of the length of the query sequence. The query sequences are partitioned into subquery sequences of length

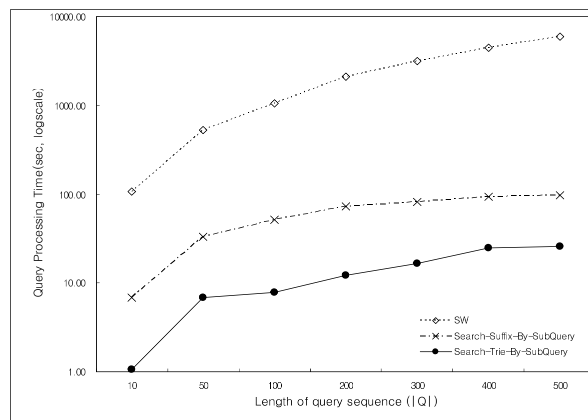


**Fig. 9.** Query processing times of the three approximate subsequence search approaches

25 in Seach-Trie-By-SubQuery. Also, the query sequences are properly partitioned into subquery sequences of length 20 or 40 in Seach-Suffix-By-SubQuery. According to our experimental results, we see that our method performs better than the other two methods and returns the answers very quickly even with large DNA data sequences. Search-Trie-By-SubQuery shows good performance regardless of the length of query sequences, and achieves 3 to 9 times speedup compared to Search-Suffix-By-SubQuery and 75 to 200 times speedup compared to SW.

## 6 Conclusions

In this paper, we have proposed an index structure and a query processing algorithm for approximate DNA subsequence search. The DNA subsequence search with the proposed index uses the dynamic programming technique, and finds all the similar subsequences stored on the paths of a binary trie. By traversing the trie index in a breadth-first fashion, it accesses just the pertinent pages within the index only once. In cases of a long sequence, it divides a query sequence into a set of shorter subsequences and retrieves actually similar subsequences by performing subsequence search for every shorter subsequence.

**Acknowledgments.** This work was supported by Korea Research Foundation Grant funded by Korea Government (MOEHRD, Basic Research Promotion Fund) (KRF-2005-206-D00015), by the Brain Korea 21 Project in 2007, by Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea Government(MOST) (No. R01-2006-000-11106-0), by the MIC of Korea under the ITRC support program supervised by the IITA(IITA-2005-C1090-0502-0009).

## References

1. A. Califano and I. Rigoutso, "FLASH: A Fast Look-up Algorithm for String Homology", *Proc. Intelligent System Conference for Morecular Biology*, pp. 56-64, 1993.
2. C. Fondrat and P. Dessen, "A Rapid Access Motif database(RAMdb) with a search algorithm for the retrieval patterns in nucleic acids or proteun databanks", *Computer Applications in the Biosciences*, Vol. 11, No.3, pp. 273-279, 1995.
3. R. Giegerich, S. Kurtz, and J. Stoye, "Efficient Implementation of Lazy Suffix Trees", *Softw. Pract. Exp.*, Vol 33, pp. 1035-1049, 2003.
4. D.Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
5. E. Hunt, M. P. Atkinson and R. W. Irving, "Database indexing for large DNA and protein sequence collections", *VLDB Journal*, Vol. 11, No. 3, pp. 256-271, 2002.
6. T. Kahveci and A. K. Singh, "An Efficient Index Structure for String Databases", *Proc. VLDB Conference*, pp. 351-360, 2001.
7. C. Meek, J. M. Patel, and S. Kasetty, "OASIS: An Online and Accurate Technique for Local-Alignment Searches on Biological sequences", *Proc. VLDB Conference*, pp. 920-921, 2003.

8. G. Navarro and R. Baeza-Yates, “A Hybrid Indexing Method for Approximate String Matching”, *Journal of Discrete ALgorithms*, Vol. 1, No. 1, pp.205-239, 2000.
9. T. Smith and M. Waterman, “Identification of Common Molecular Subsequences”, *Journal of Molecular Biology* 147, pp. 195-197, 1981.
10. S. Tata, R. Hankins, and J. Patel, “Practical Suffix Tree Construction”, *Proc. VLDB Conference*, pp. 36-47, 2004.
11. H. Wang et al., “BLAST++: A Tool for BLASTing Queries in Batches”, *Proc. Asia-Pacific Bioinformatics Conference*, pp. 71-79, 2003.
12. H. E. Williams and J. Zobel, “Indexing and Retrieval for Genomic Databases”, *IEEE TKDE*, Vol. 14, No. 1. pp. 63-78, 2002.
13. J. I. Won, J. H. Yoon, S. H. Park and S. W. Kim, “A Novel Indexing Method for Efficient Sequence Matching in Large DNA Database Environment”, *Proc. PAKDD Conference*, pp. 203-215, 2005.