A Novel Indexing Method for Efficient Sequence Matching in Large DNA Database Environment

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

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

Abstract. In molecular biology, DNA sequence matching is one of the most crucial operations. Since DNA databases contain a huge volume of sequences, fast indexes are essential for efficient processing of DNA sequence matching. In this paper, we first point out the problems of the suffix tree, an index structure widely-used for DNA sequence matching, in the respects of the storage overhead, search performance, and difficulty in seamless integration with DBMS. Then, we propose a new index structure that resolves such problems. The proposed index structure consists of the two parts: the primary part realizes the *trie* as binary bit-string representation without any pointers, and the secondary part helps fast accesses of leaf nodes of the trie that need to be accessed for post-processing. We also suggest efficient algorithms based on that index for DNA sequence matching. To verify the superiority of the proposed approach, we conduct performance evaluation via a series of experiments. The results reveal that the proposed approach, which requires smaller storage space, can be a few orders of magnitude faster than the suffix tree.

Keywords: DNA databases, DNA sequence matching, indexing.

1 Introduction

DNA sequences hold the code that determines life characteristics of every living organism. A DNA sequence is represented as a string of a four-character alphabet of A, C, G, and T known as the nucleotide bases. The DNA database contains a huge volume of DNA sequences. Historically, the database has roughly doubled in size every 14 months, and the increasing rate is growing gradually [3]. Since the size of DNA databases increases considerably as such, fast indexing is crucial for an efficient information retrieval from those databases. DNA subsequence matching is an operation that is most frequently performed on a DNA

T.B. Ho, D. Cheung, and H. Liu (Eds.): PAKDD 2005, LNAI 3518, pp. 203–215, 2005.

database [7][20]. Given a database S, a query sequence Q, and a tolerance T, it finds subsequences S' of S whose dissimilarity with some subsequences Q' of Q is less than T.

BLAST [1] is a de-facto standard tool widely used by molecular biologists to perform DNA subsequence matching. BLAST provides high performance by using a heuristic algorithm, however, does not guarantee accuracy; i.e, it may loose some true answers. The most popular algorithm that guarantees accuracy is the Smith-Waterman algorithm [16]. The Smith-Waterman algorithm uses a dynamic programming approach for finding an optimal local alignment between S and Q of the two sequences. However, it suffers from a long processing time of $O(|Q| \times |S|)$.

The suffix tree has been known to be a good index structure for efficient DNA subsequence matching [5][11]. The suffix tree is a compressed digital trie whose set of keywords comprises the suffixes of given sequences. The suffix tree shows reasonable performance in finding all the matched subsequences. Moreover, it is ready to be applied to applications that necessitate DNA subsequence matching since approximate matching algorithms for it have already been proposed [18][8]. The elapsed time of subsequence matching by using such algorithms, however, increases dramatically as the length of a query sequence and a tolerance increase. To alleviate this problem, reference [13] proposed a hybrid indexing method that divides a query sequence into multiple smaller pieces, performs their subsequence matchings with a smaller tolerance, and then integrates the results thus obtained. Also, reference [12] suggested a method that applies the best-first(A*) search method [9] in traversing a suffix tree. It shows the performance of subsequence matching comparable to that of BLAST in case of short query sequences. Moreover, it guarantees accuracy as in the Smith-Waterman algorithm.

The suffix tree still has the following drawbacks due to its structural characteristics: (1) **Storage space:** The suffix tree requires a large storage space; It is often several ten times larger than a database [10][13][6]. Hunt et al. [8] reported that a suffix tree required 19G bytes when they built it on DNA sequences of 286M bases. (2) **Search performance:** The large storage space required by a suffix tree inversely affects the search performance. In addition, the poor locality of the suffix tree causes a significant loss of efficiency in respect of disk accesses [6]. Thus, overall search performance deteriorates in DNA databases. (3) **Integration with DBMS:** DBMS uses a page as a unit for storing all kinds of data on disk. In contrast, the suffix tree has a difficulty in employing a page as a storage unit due to its structural characteristics [17][19]. Thus, the suffix tree has a problem in integrating itself with DBMS seamlessly.

In this paper, we propose a novel index structure that supports DNA subsequence matching efficiently as well as resolves the above drawbacks of the suffix tree. The proposed index adopts a trie [17] as its conceptual structure and realizes the trie by binary bit-string representation without pointers. In addition, it employs a multi-dimensional index as a secondary structure for fast accesses of the target leaf nodes when traversing the trie. With these characteristics, the proposed index successfully solves all the problems in the respects of the storage

space, search performance, and integration with DBMS. We also propose algorithms that effectively process both exact and approximate DNA subsequence matching by using the proposed index. Through extensive experiments, we quantitatively verify the effectiveness of our approach in comparison with the previous ones. The results reveal that, compared with the previous ones, our approach requires smaller storage space and achieves several times to several ten times improvement in DNA subsequence matching performance.

2 Indexing Method

2.1 Binary Suffix Trie

A trie is defined as a $|\sum|$-*ary* tree in which each edge has a symbol from the alphabet $\sum$ and symbols in each root-to-leaf path form a key. Here, $|\sum|$ is the alphabet size. A selection of subtries at level i is determined only by the i^{th} symbol of the search key, not the whole key. The most straightforward implementation of $|\sum|$-ary tries is to store $|\sum|$ pointers in each node. This method enables to select a child node in constant time. However, it is not space-efficient because trie nodes may contain lots of NULL pointers when $|\sum|$ is large. An alternative is to use dynamic data structures such as linked lists. In the linked list representation, each trie node stores two pointers, one to its leftmost right sibling and one to its leftmost child. This implementation reduces a lot of NULL pointers and therefore requires lesser storage space especially when $|\sum|$ is large. However, it cannot select a child node in constant time. In the worst case, all the child nodes have to be examined.

Shang et al. [15] suggested pointerless binary tries which attained competitive search speed with a minimal storage requirement. Pointerless binary tries require the alphabet $\sum$ to have only two symbols, 0 and 1. Therefore, every node has at most two outgoing edges. In the pointerless binary bit-string representation, the symbols on the edges do not have to be stored explicitly by enforcing the following rules: (1) the outgoing edge labeled with 0 connects to the left child node, and (2) the outgoing edge labeled with 1 connects to the right child node. More specifically, the trie node storing the two-bit data '10' has only one child which is on its left, and the node storing the two-bit data '01' has only one child which is on its right. Similarly, the trie node with '11' has both left child and right child, and the node with '00' has no child.

In this paper, we propose an index structure for efficient DNA sequence matching, exploiting the basic concepts of pointerless binary tries. Our aim is to efficiently find the subsequences matched exactly or approximately to a query sequence. Therefore, we extract all the *suffixes* from the DNA sequences and insert each one of them into the trie. Since the suffixes are the inputs to the trie construction algorithm, the resultant tree has the properties of suffix tries [17]. Suffix tries compress the input data set substantially when the input sequences have lots of common prefixes. A DNA sequence can be considered as a string from the alphabet $\sum = \{A, C, G, T\}$. Since the alphabet size is small (which

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

Fig. 1. Binary code of each symbol in the alphabet

Suffix	Binary Representation
S1: ACGT$	00 10 10 01 11 10 10 00
CGT$	01 00 11 10 10 00
GT$	01 11 01 00 00
T$	10 10 00
S2: ACT$	00 10 10 10 10 00
CT$	01 01 01 00 00
T$	10 10 00

Fig. 2. Binary representations of the suffixes from S_1 = 'ACGT' and S_2 = 'ACT'

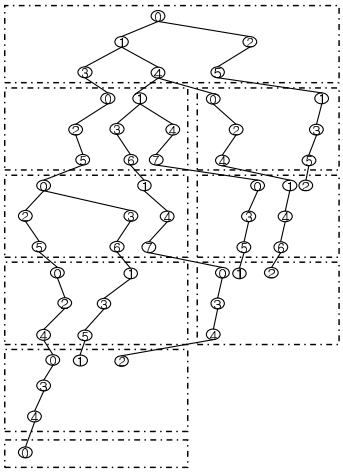

Fig. 3. Binary suffix trie constructed from the suffixes of Figure 2

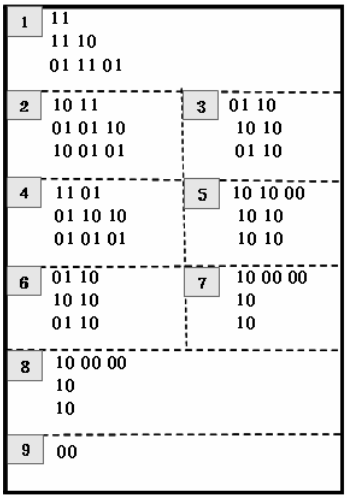

Fig. 4. Internal representations of the binary suffix trie in Figure 3

is 4), it is highly possible that there exist a considerable number of common prefixes in the suffixes of the input data set.

In this research, we use the minimum number of bits to represent each symbol rather than using a character of 8 bits, to obtain higher compression ratio. Note that DNA sequences may contain wild-card characters as well as the four typical symbols of A, C, G, and T. For example, the wild-card N denotes one from A, C, G, and T, and B denotes one from C, G, and T. Although wild-card characters do not occur frequently, we need to uniquely encode each wild-card character in addition to the typical four characters. For instance, when the number of disparate symbols occurring in the DNA sequences to be indexed is at most seven, we can use 3 bits to encode each symbol uniquely. If we construct the suffix trie from DNA sequences encoded binary, we can expect a higher compression ratio due to the increased number of common prefixes.

Let us examine the steps to build a binary suffix trie using an example. Figure 1 shows a binary code of each symbol in our alphabet. Here, '$' is a

special character used as an end marker of every suffix. Given two sequences S_1 = 'ACGT' and S_2 = 'ACT', we first convert all of their suffixes into the corresponding binary bit-string representations as shown in Figure 2. We then construct the trie through successive insertions of binary suffixes according to their lexicographic order. Insertions based on the lexicographic order make the trie grow only one direction and thus facilitate the disk-based trie construction. Figure 3 shows the binary suffix trie constructed from the suffixes of Figure 2, and Figure 4 shows its internal representation.

For the trie construction, we use a disk-based algorithm [4]. Therefore, whenever the main memory space of a predetermined size (i.e. page size) becomes occupied by a sub-trie, it is written onto a secondary storage (i.e. disk). To prevent a sub-trie larger than a page from being written onto a disk page, we precalcuate the maximum number of trie levels and the maximum number of trie nodes that can be stored within a single page. In each page, the child nodes of each level are either entirely on or entirely off that page. In other words, edges can only cross the horizontal boundaries of pages, not the vertical boundaries. This restriction is to reduce the number of disk pages to be read during query processing. Since the trie is partitioned into a set of pages, it is necessary to maintain the *page table* [15] to figure out the page connections. Each entry of the page table corresponds to a page and stores information related to that page, and each entry is filled right after the corresponding page has been written on the disk.

2.2 Storing Leaf Nodes

Each suffix is identified by the pair of the sequence identifier and the starting offset. When a suffix is inserted into the trie, its identifier is stored in the corresponding leaf node. However, every trie node is represented by a two-bit number in our indexing scheme. Therefore, suffix identifiers have to be kept separately from the trie, using, i.e., a leaf node table.

When a query sequence is given, we traverse down the trie to find a node beyond which more comparisons are meaningless. When the matching is successful, a series of labels on the path between the root node and the node visited last becomes the subsequence we are looking for in the database. To find the locations at which the subsequences matched to a query sequence start, we need to retrieve all the leaf nodes under the node visited last and get the suffix identifiers stored in these leaf nodes. When the index is large and the traversal ends at a position not deep, a large portion of the trie has to be visited.

In this work, we propose to use a multi-dimensional index to speed up the operation that retrieves all the leaf nodes under a given internal node. By regarding a binary bit-string representation of a suffix as a multi-dimensional key, we build a multi-dimensional index from a set of suffixes. Notice that suffixes do not have the same length. Therefore, we need the following scheme to convert a suffix of variable length into a set of predetermined k integers: (1) **When the binary bit-string representation of a suffix is shorter than k-integer length**, we append multiple 0s to the end of a binary bit-string to make it be of k-integer length. (2) **When the binary bit-string representation of a**

suffix is longer than k-integer length, we cut out the rightmost bits so that the resultant binary bit-string becomes of k-integer length.

3 Query Processing Method

3.1 Exact Subsequence Matching

Since each trie node is represented by a two-bit number in the proposed index, the pointers from parents to children are not stored explicitly. The information on the trie levels is not stored explicitly, either. Therefore, while traversing down the index to find the subsequences matched to a query sequence, the algorithm has to fetch the corresponding page and then extract those *implicit* information using the data in the page.

Algorithm 1. Query processing algorithm Search-Trie

Input : binary suffix trie T, query sequence Q, page table P
Output: set of answers

```
1  initialize C_0, N_{0,c}, S_0, and N_{0,f};
2  for j := 0;  j < p_Height;  j++ do
3      if j > 0 then
4          page_change(P);
5          reset C_0, N_{0,c}, S_0, and N_{0,f};
6      for i := 0;  i < n_Height;  i++ do
7          while isBefore(N_{i,c}) do
8              increase C_i;
9              update S_i;
10         if !(match(node(N_{i,c}), Q_i)) then
               return {};
11         if isLast(Q_i) then
               return find_answers();
12         get(Q_{i+1});  increase C_i;  update S_i;
13         while isBefore(N_{i,f}) do
               update S_i;
14         if i < (n_Height − 1) then
               reset C_{i+1}, N_{i+1,c}, S_{i+1}, and N_{i+1,f};
```

The algorithm Search-Trie which traverses the binary suffix trie T to retrieve the subsequences matched to a query sequence is shown in Algorithm 1. We assume that the query sequence Q has been already converted to its binary form. Remember that the information related to the page partitioning is maintained in the page table P. Let L_i denote the i^{th} trie level in the page that is being

examined. The algorithm uses the following four variables to figure out the internal structure of the page. The variable S_i stores the total number of nodes located at L_i. If a node at L_i has the value '11', it will increase S_{i+1} by one. On the contrary, if a node at L_i has the value '00', it will decrease S_{i+1} by one. The variable $N_{i,f}$ denotes the position of the rightmost node at L_i. $N_{i+1,f}$ is simply computed by summing $N_{i,f}$ and S_{i+1}. The variable $N_{i,c}$ indicates the position of the node at L_i that should be compared with the i^{th} query bit. The variable C_i stores the total number of 1 bits counted from the leftmost node at L_i to the node positioned at $N_{i,c}$. $N_{i+1,c}$ is obtained by summing $N_{i,f}$ and C_i.

The algorithm Search-Trie operates as follows. We assume that the index has p_Height page levels and each page level has n_Height node levels. First, we initialize all the variables according to the fact that the first node of the first page in the index is the root (line 1). The lines 3-5 in the external for loop (lines 2-14) replace the current page level with the next page level. The function page_change(P) in line 4 computes the location of the next page using the information in the page table P, and reads in the next page. Next, all the variables are updated before entering into the stage of traversing the nodes in the new page. The internal for loop (lines 6-14) is for handling a node level, and it consists of the following four steps. Increasing C_i and updating S_i, the first step (lines 7-9) sequentially reads the nodes positioned before $N_{i,c}$. The second step (lines 10-12) checks whether the node $N_{i,c}$ matches the i^{th} query bit Q_i or not. If not matched, the statement in line 10 is executed. If matched, the algorithm checks if there are more query bits to be examined. If there is no more query bit left, the function find_answers() is called in line 11. The function find_answers() retrieves the suffix identifiers from the leaf nodes under $N_{i,c}$. If there are more query bits to be examined, the statement in line 12 is executed where the next query bit is read and the variables S_i and C_i are updated and increased respectively. While updating the variable S_i, the third step in line 13 sequentially reads the nodes positioned before $N_{i,f}$. The final step in line 14 resets all the variables if there remain more node levels in the current page.

3.2 Direct Access of Leaf Nodes

The algorithm Search-Trie has the step to retrieve all the leaf nodes under the node $N_{i,c}$ at which the last query bit is matched successfully. This operation is mainly performed in the function find_answers(). The multi-dimensional index introduced in Section 2.2 enables direct retrieval of the leaf nodes under $N_{i,c}$. When the path p from the root to $N_{i,c}$ matches the query sequence, we take one of the following three options according to the length of p. (1) **When p has the length shorter than k-integers**: Let p_0 denote the binary bit-string of k-integer length obtained by appending multiple 0s to the end of p. And let p_1 denote the binary bit-string of k-integer length obtained by appending multiple 1s to the end of p. From the multi-dimensional index, we retrieve all the leaf nodes having the values between p_0 and p_1. (2) **When p has the length of k-integers**: From the multi-dimensional index, we retrieve all the leaf nodes having the value p. (3) **When p has the length longer than k-integers**:

Let p_k be the prefix of p with k-integer length. From the multi-dimensional index, we retrieve all the leaf nodes having the value p_k. Then, we perform the post-processing to detect and discard false matches.

3.3 Approximate Subsequence Matching

The basic method for approximate subsequence matching in DNA databases is the dynamic programming (DP) technique. Given two sequences Q and S, the DP technique finds their optimal distance by building a two-dimensional DP table of $|Q|+1$ rows and $|S|+1$ columns. The recurrence relations corresponding to the similarity measure of a target application are used to fill in each cell of the DP table. The edit distance function [8][17] is a popular similarity measure for approximate subsequence matching.

There have been several approaches [18][13][8] which employ the suffix tree as an index to speed up approximate subsequence matching. They traverse the suffix tree in the depth-first order and build-up the DP table between a query sequence and a path from the root node of the suffix tree. The proposed binary suffix trie also can be used as an index structure for approximate subsequence matching. However, since every node is represented by a two-bit number in the binary suffix trie, we need to access more than one node to append a new column to the DP table.

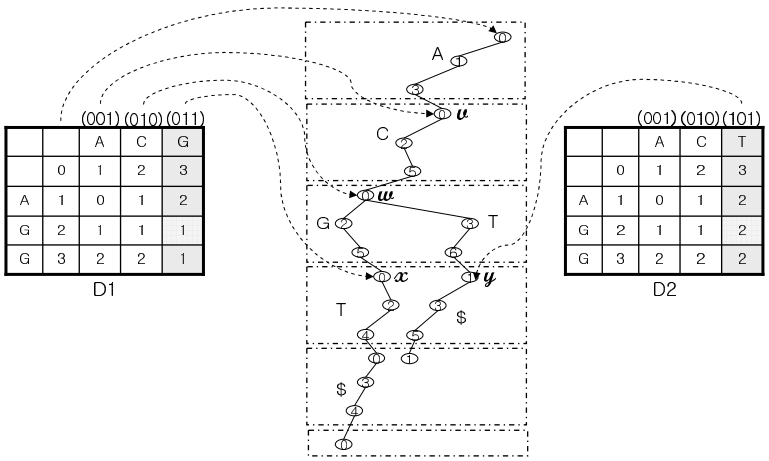

Fig. 5. DP tables constructed from the binary suffix trie of Figure 3

Let us use an example to explain the proposed approximate subsequence matching algorithm. Suppose that we want to find the subsequences whose edit distances to the query sequence 'AGG' are not larger than 1. Figure 5 shows how the DP tables are constructed during the traversal of the binary suffix trie shown in Section 2. Since every symbol is encoded by three bits, the algorithm accesses

three successive nodes to append a new column to the existing DP table. That is, the columns for the symbols 'A(001)', 'C(010)', and 'G(011)' are appended individually to the DP table when the algorithm reaches the nodes v, w, and x, respectively. D_1 in Figure 5 is the resultant DP table. Whenever a new column is added to the DP table, we check whether or not the cell at the last row of the newly added column has a value not larger than a distance threshold. If so, all the leaf nodes under the node being visited satisfy the query. We use the multi-dimensional index to directly retrieve such leaf nodes. In D_1 of Figure 5, the column for the symbol 'G(011)' is the newly added column. Since the value of the cell at its last row is 1, all the leaf nodes under the node x satisfy the query. The DP table D_2 is obtained when the node y is visited. Since all the cells in the last column have values larger than 1, the traversal stops at the node y and comes back to its parent. Note that the first two columns of D_1 and D_2 tables are identical. These two columns are shared by the two tables to save space and time.

4 Performance Evaluation

In our experiments, we have used DNA sequences of human chromosomes 18, 19 and 21 downloaded from GenBank [14]. From those data sequences, we have randomly extracted some subsequences of arbitrary lengths as query sequences. The DNA sequences used in our experiments consist of four frequent characters A, C, G, and T, and also contain some infrequent wild-card characters such as N, S, and Y. In addition, we have used a special character \$ for representing the end of a sequence. Thus, 8 different characters may appear within the DNA sequences in our experiments. The hardware platform is the Pentium IV 2GHz PC equipped with 1 Gbyte main-memory and 40 Gbyte hard disk. The software platform is the Windows 2000 Server.

In experiment 1, we have compared three approaches Trie-Rtree, Trie-Naive, and Suffix in the respect of the index size. Trie-Rtree represents our approach that employs the trie using pointerless binary bit-string representation in conjunction with a multi-dimensional index. As a multi-dimensional index, we have used the R*-tree [2], a most-widely used in the literature. Trie-Naive also represents our approach that uses just the trie using pointerless binary bit-string representation without employing a multi-dimensional index. Finally, Suffix is the previous approach based on the suffix tree. We have applied an incremental disk-based algorithm [4] for suffix tree construction, and also have allocated 32 byte memory chunk for each node in the suffix tree.

Figure 6 shows the change of the index sizes in the three approaches with different data sizes. We have set the page size for each index to 4K bytes. The suffix tree in Suffix consists of internal nodes and leaf nodes. The index in Trie-Naive consists of a binary suffix trie, a page table, and a leaf node table. The index in Trie-Rtree contains those used in Trie-Naive, and also maintains an additional R*-tree for fast accesses of leaf nodes of the trie. In the figure, we observe that the index size increases almost linearly in proportion to the data size in all the

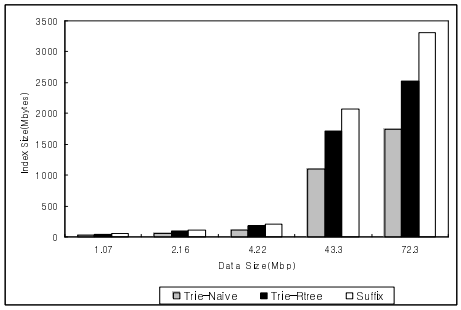

Fig. 6. Index sizes with different data sizes

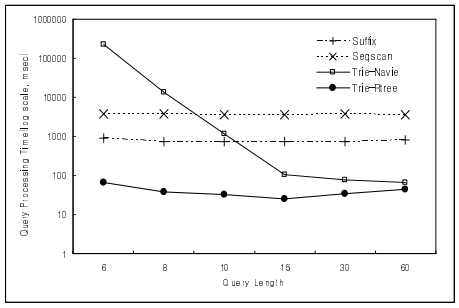

Fig. 7. Elapsed times of exact subsequence matching with different query sequence lengths

approaches. The results show that our Trie-Naive and Trie-Rtree achieve around 48% and 24% savings in storage space, respectively, in comparisons with Suffix.

In experiment 2, we have compared the three approaches along with Seqscan in terms of the elapsed time for exact subsequence matching. Seqscan is the simplest baseline method for DNA sequence matching, which is based on sequential scan. For this experiment, we have used human chromosome 21 of 43.3 Mbp as a data sequence. The total elapsed time is the time spent in finding the offsets in the DNA sequence from which subsequences exactly matched to a query sequence start.

Figure 7 shows the result. Seqscan performs poorly regardless of query sequence lengths. Trie-Navie performs well with long query sequences, but performs poorly with short query sequences due to its high overhead for post-processing. On the other hand, Trie-Rtree shows good performance regardless of query sequence lengths, and achieves 13 to 29 times speedup compared with Suffix, 54 to 145 times speedup compared with Seqscan.

In experiment 3, we have compared two approaches Trie-Rtree and Suffix in terms of the elapsed time for approximate subsequence matching. We have employed two different approaches: one is to find all the subsequences whose edit distances to a query sequence are not larger than k, which has been commonly used in many DNA subsequence matching, and the other is to find the similar subsequences using the best-first(A*) search algorithm. The data sequence used in the experiment is human chromosome 21 of 43.3Mbp.

Figure 8 shows the elapsed times of approximate subsequence matching by Suffix and Trie-Rtree for finding all the subsequences whose edit distances to a query sequence are not larger than 1. In the current experiment, we follow the behavior of reference [13], considering only short query sequences with a small tolerance. The elapsed time here is the total time required for obtaining pairs <sequence number, offset> of all the similar subsequences. The values within parentheses represent the post-processing time spent in finding leaf nodes. The result shows that Suffix has a large elapsed time for short query sequences due to a big post-processing time. On the other hand, Trie-Rtree shows better

Query Length	Total hits	Query Processing Time(msec)	
		Trie-Rtree	Suffix
6	388,321	817.4(623)	14,248.5(7446)
8	33,422	854(412.7)	3,120.2(674.9)
10	3,857	1,157.5(365)	3,055.6(109)
15	22	1,216.9(3.1)	3,456.8(0.4)

Fig. 8. Elapsed times spent in finding all the subsequences whose edit distances to a query sequence are not larger than 1

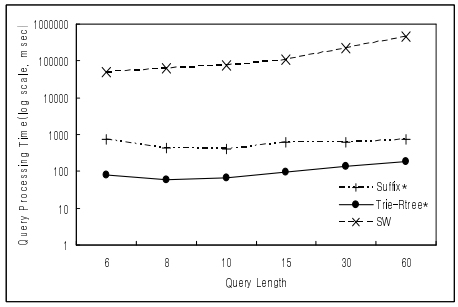

Fig. 9. Elapsed times spent in finding the subsequence most similar to a query sequence

performance due to direct accesses of leaf nodes by using the R*-tree. For long query sequences, however, a large number of bit operations increase the time for traversing the suffix trie, and subsequently enlarge the entire elapsed time.

Figure 9 depicts the result of comparing the elapsed times of Suffix*, Trie-Rtree* and SW. Here, the elapsed time is the total time required to find a set of subsequences, each of which is most similar to a query sequence in each data sequence, from a DNA database. Trie-Rtree* and Suffix* represent the elapsed time of approximate subsequence matching by Trie-Rtree and Suffix, respectively, that employ the best-first(A*) search algorithm [12]. Also, SW represents elapsed time of approximate subsequence matching by the Smith-Waterman algorithm. The result shows that Trie-Rtree* performs better than Suffix*. This is because the way for storing nodes in the suffix trie harmonizes with the level-first traversal fashion of the best-first(A*) search algorithm. That is, as mentioned in Section 2.1, all the child nodes of each level of a page are either entirely on or entirely off that page. This is quite effective in such environment where all the sibling nodes are accessed together as in the best-first(A*) search. The result shows that, compared with Suffix* and SW, Trie-Rtree* performs about 4 to 9 times and about 592 to 2,505 times better, respectively.

5 Conclusions

In this paper, we first have pointed out the problems occurring in the suffix tree for DNA sequence matching: (1) high storage overhead, (2) low search performance, (3) difficulty in seamless integration with DBMS. Then, we have proposed a novel index structure that resolves them. Our index employs a trie as its primary structure and implements it by using binary bit-string representation without pointers. Major advantages of this implementation are to reduce the storage overhead considerably and to build its structure easily in page units. Also, our index employs a multi-dimensional index as a secondary structure for

fast accesses of the target leaf nodes after traversing the trie. With the proposed index, we can successfully alleviate the three problems of the suffix tree. We also have proposed the algorithms that process DNA sequence matching effectively based on the proposed index. To verify the effectiveness of our approach, we have performed a series of experiments. The results reveal that the proposed approach, which requires smaller storage space, can be a few orders of magnitude faster than the suffix tree. In case of exact matching, Trie-Rtree, our enhanced approach, runs 13 to 29 times faster than the Suffix. In case of approximate matching, it achieves 4 to 9 times speedup over Suffix.

Acknowledgments. This work was supported by the Basic Research Program(Grant R04-2003-000-10048-0) of KOSEF, the ITRC support program (MSRC) of IITA, the Hanyang University(HY-2003-T), and the Korea Research Foundation Grant (KRF-2004-003-D00302).

References

1. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool", *Journal of Molecular Biology*, 215, pp. 403-410, 1990.
2. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles", *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 322-331, 1990.
3. D. A. Benson, M. S. Boguski, D. J. Lipman, J. Ostell, and B. F. Quellette, "Genbank", *Nucleic Acids Research*, Vol. 26, No. 1, pp. 1-7, 1998.
4. P. Bieganski, J. Riedl, and J. V. Carlis, "Generalized suffix trees for biological sequence data: applications and implementation", *Proc. Hawaii International Conference on System Sciences*, 1994.
5. A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, "Alignment of whole genomes", *Nucleic Acids Research*, 27, pp. 2369-2376, 1999.
6. R. Giegerich, S. Kurtz, and J. Stoye, "Efficient Implementation of Lazy Suffix Trees", *Softw. Pract. Exp.*, Vol 33, pp. 1035-1049, 2003.
7. R. S. C. Goble, P. Baker, and Brass, "A Classification of tasks in bioinformatics", *Bioinformatics*, Vol. 17, No. 2, pp. 180-188, 2001.
8. 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.
9. K. Kelly and P. Labute, "The A* Search and Applications to Sequence Alignment", http://www.chemcomp.com/article/astar.htm, 1996.
10. S. Kurtz and C. Schleiermacher, "REPuter: fast computation of maximal repeats in complete genomes", *Bioinformatics*, Vol. 15, No. 5, pp.426-427, 1999.
11. S. Kurtz, J. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich, "REPuter: the manifold applications of repeat analysis on a genome scale", *Nucleic Acids Research*, Vol. 29, No. 22, pp. 4633-4642, 2001.
12. 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.
13. 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.

14. http://www.ncbi.nlm.nih.gov
15. H. Shang and T. H. Merrett, "Tries for approximate string matching", *IEEE Trans. on Knowledge and Data Engineering*, Vol. 8, No. 4, pp. 540-547, 1996.
16. T. Smith and M. Waterman, "Identification of Common Molecular Subsequences", *Journal of Molecular Biology* 147, pp. 195-197, 1981.
17. G. A. Stephen, *String Searching Algorithms*, World Scientific Publishing, 1994.
18. E. Ukkonen, "Approximate string matching over suffix trees", *Proc. Combinatorial Pattern Matching*, pp. 228-242, 1993.
19. H. Wang et al., "BLAST++: A Tool for BLASTing Queries in Batches", *Proc. Asia-Pacific Bioinformatics Conference*, pp. 71-79, 2003.
20. H. E. Williams and J. Zobel, "Indexing and Retrieval for Genomic Databases", *IEEE TKDE*, Vol. 14, No. 1. pp. 63-78, 2002.